Episode Book

EPISODE #5938

One Regex Rule Worked, One Was Impossible

PUBLISHED	13 October 2026 at 18:07 UTC
DURATION	27:19
LISTEN	myweirdprompts.com/episode/regex-limits-android-whisperian-54308/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.4
SOURCES	20 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:41 UTC

SYNOPSIS

A listener wrote one text-replacement rule in an Android dictation app and it erased every "um" and "uh" from his transcripts. Then he wrote a second rule about capitalization, and it failed — not because it was fiddly, but because it's provably impossible. This episode uses that gap as a case study in what regular expressions actually are: a 1951 notation from Stephen Cole Kleene, first implemented by Ken Thompson in 1968, now a family of dialects (POSIX, Perl, PCRE2, RE2) that disagree about what the word even means. Along the way: why the app's Accessibility-service architecture constrains everything you can write on Android, why JavaScript's own reference admits its regexes aren't regular, and why LLMs haven't made any of this obsolete.

TRANSCRIPT



Corn

Most people hear "regular expressions" and their eyes glaze over. They think it's a syntax error waiting to happen, some arcane punctuation soup that only sysadmins and Perl programmers ever loved. Herman: And yet it's running inside almost everything they touch. Every search box, every text editor, every log parser, every form validator. Corn: Which is strange, because the notation is seventy-five years old. Herman: Older than the transistor being commercially viable, older than the hard drive. Corn: So Daniel sent us something. He found an Android dictation app called Whisperian, says he tried literally every app in the category before landing on this one, and he's not selling anything, he just thinks it's good. It decouples the speech recognition from the post-processing, so you can pick your ASR model, pick your cleanup model, write your own prompt. But the feature that got him writing in is the text replacement rules, which run on device and support regex. Herman: And he wrote one rule. Corn: One rule that killed every disfluency. Um, uh, uh-huh, er, ah, all of it, gone in a single line. Herman: Then he tried to write a second rule, and it didn't work. Corn: The second one was about capitalization. He dictates with an overlay button, he pauses mid-sentence to gather his thoughts, the model comes back and capitalizes the first word of the new transcript because it doesn't know the sentence is still going. He wanted a rule that said, if the character isn't the first or second since the last period, make it lowercase. Herman: And that's where it fell apart. Corn: That's where it fell apart. So he wants to know what regex can and can't do, how running it on Android compares to running it on Linux, and whether LLMs have made the whole thing obsolete. Herman: The answer to the last one is no. Corn: I know. But the reason is interesting.



Herman

Here's the tension in the prompt, and it's a good one. Same tool, same week, same person. One rule works like a charm, the other is provably impossible. Corn: Provably. Not "hard," not "fiddly." Herman: Provably. That's a mathematical claim, and it holds. Corn: So the episode is really about that gap. What's mechanically matchable versus what requires understanding. Herman: Twenty years ago you'd have said "what a computer can do versus what a person can do," and that line has moved enormously since. But this particular line hasn't moved at all, because it's not a performance limit. It's a structural one. Corn: Which we'll get to. Tell me about the app first, because the app is the case study. Herman: Whisperian, version 0.26.0, about forty-four and a half megabytes, a thousand-plus installs, on Google Play since April, developer Duplex Limited. Corn: A thousand installs. That's nothing. Herman: It's early access, all features free, no sign-up. And their own website is styled as a mock Victorian newspaper, which tells you something about the developer's sense of humor. Corn: "Issue Number 801." Herman: And it does the thing Daniel said. Profiles let you configure the transcription provider, the post-processing model, a custom prompt, and text replacement rules, all separately. Local Parakeet models or a cloud provider through OpenRouter. Corn: The decoupling is the important part. Herman: It is. And the reason it can work system-wide, in any app, is that it uses an Accessibility service. That's how it detects keyboard visibility, starts recording, and reads the active text field and cursor position to insert text. Corn: Which is also the constraint we're going to spend the back half of this episode on. Herman: Exactly. Everything about what rules you can write on Android flows from that one architectural fact. Corn: Then let's earn it. Where did this notation come from?



Herman

1951. A mathematician named Stephen Cole Kleene, working at RAND, wrote a memo called "Representation of Events in Nerve Nets and Finite Automata." It later became a chapter in a Princeton volume, Automata Studies, in 1956. Corn: Kleene was building on McCulloch and Pitts' work on neural nets. Herman: Yes, the 1943 paper. He was asking what kinds of pattern a network of simple threshold units could recognize, and he named the answer "regular events." Corn: And he described them with three operations. Herman: Union, which is the pipe character. Concatenation, which is just putting symbols next to each other. And star, meaning zero or more repetitions. Corn: That's it. Herman: That's the whole notation. And those three operations are exactly enough to describe any language a finite automaton can recognize. That's the Kleene theorem, and it's the reason regex and finite automata are the same subject wearing different clothes. Corn: Seventeen years before anybody implemented it. Herman: Seventeen years. Kleene writes it down in 1951, and nobody builds it into software until Ken Thompson puts it into the QED editor on MIT's CTSS, and publishes "Regular Expression Search Algorithm" in the Communications of the ACM in June of 1968. Corn: First implementation. Herman: First one. And he did it with a technique we now call Thompson's construction, which compiles a pattern into a little state machine on the fly. Corn: So the pattern isn't interpreted line by line. Herman: No, it's compiled. That's why it's fast. Then it moves into the ed editor on Unix in 1971, and in 1973 Thompson pulls one command out of ed and makes it standalone. Corn: And that command, for the record, is not an acronym. Herman: It is not. It's the literal ed command characters. g, slash, re, slash, p. Global, regular expression, print. Corn: Global regex print. Herman: That's the name. It's a keystroke sequence that got promoted to a word. Corn: I have heard people expand it as "global regular expression parser" or "print." Herman: Wrong. There's no expansion. It's just the command.



Corn

Then the dialects start. Herman: Then the dialects start. POSIX standardizes it twice in 1992, basic and extended, which is why you sometimes have to escape a parenthesis and sometimes don't, depending on which mode you're in and how old the tool is. Corn: A gift to humanity. Herman: Larry Wall's Perl 1.0 arrives in December 1987 and adds lazy quantifiers, lookarounds, backreferences, named groups, and the shorthand classes, backslash d, backslash w, backslash s. Corn: Which everyone now assumes are universal. Herman: They're not universal, they're Perl's. Then PCRE, written by Philip Hazel in 1997, becomes the de facto standard for anything that isn't POSIX, and PCRE2 in 2015 is the current line. And then in 2010, Russ Cox at Google releases RE2, which does linear-time matching and deliberately drops backreferences and recursion. Corn: Deliberately. Herman: Deliberately. Because those are the features that make matching blow up. Corn: So "regex" today is a family, not a thing. Herman: That's the first misconception to kill. When you say "regular expression," you might mean the formal language Kleene described, or you might mean the PCRE dialect with backreferences that is provably not regular at all. Corn: And the MDN documentation says it outright. Herman: "JavaScript regular expressions are in fact not regular, due to the existence of backreferences." That's a sentence in the official JavaScript reference. Corn: Which is a wonderful thing to find in a reference manual.



Herman

Now the mechanism, because this is where Daniel's second rule dies. A finite automaton does not have memory apart from the state it's in. Corn: Say more about that. Herman: It has a fixed number of states, decided when the pattern is compiled, and all it knows at any moment is which state it's in and what character it just read. It can't count. It can't remember that it saw something earlier and keep a running tally. Corn: So "balanced parentheses" is the classic example. Herman: The canonical one. No regular expression can match arbitrarily nested parentheses, because you'd need to count depth, and counting depth needs unbounded states. You can write a regex that matches two levels or five levels, but not any number of levels. Corn: And the same reason kills backreferences. Herman: Backreferences reference a captured group, and that's storage. The expression open bracket a b close bracket star, then backslash one, matches a word followed by the same word again, which is a language that isn't even context-free. Corn: Not even context-free. Herman: That's the strongest way to put it. It's beyond the whole hierarchy below, not just the regular tier. And once you allow that in your engine, you're no longer running a finite automaton, you're running a backtracker. Corn: Which is why RE2 won't do it.



Herman

So look at Daniel's first rule. Um, uh, uhm, umm, eh, ehm, erm, er, ah, ahem, each with a plus quantifier, wrapped in a non-capturing group, anchored by word boundaries on both ends. Corn: That's a finite list of alternatives. Herman: Finite list, bounded repetition on each, and a word boundary is a zero-width assertion that only looks at the character before and after. There's no counting anywhere in it. The compiled automaton has some number of states, probably under a hundred, and it never needs to remember anything beyond which one it's in. Corn: Which is why one rule did the whole job. Herman: Which is why one rule did the whole job. And this is the right tool for that job, even though the academic literature on disfluency removal has mostly moved to transformers. There's a whole line of work arguing that detection-only strategies are insufficient because just deleting the token often breaks the grammar of the sentence around it. Corn: But "um" and "uh" aren't content. Herman: Right. The literature is about the hard cases, the repairs, the restarts, the false starts where someone abandons a sentence and begins again. Those need semantics. Filler words don't. You can strip them mechanically and lose nothing, and regex does it in microseconds with no model, no network call, and no cost. Corn: And then the second rule. Herman: And then the second rule. Daniel's heuristic is: if the character is not the first or second since the last period, lowercase it if it's uppercase. Corn: Which sounds simple. Herman: It is simple, and it's impossible. The rule has to know whether a period has appeared earlier in the string, and then keep track of how many characters have gone by since, and only act if that count exceeds two. That's a counter, and a counter is exactly what a finite automaton doesn't have. Corn: What about lookbehind? Herman: Lookbehind is a zero-width assertion. It says "the thing before this position matches this pattern," but it doesn't store anything and it can't count. You can write a lookbehind that says the previous character isn't a period, or the previous two aren't a period and a space. The moment you need to say "the most recent period was at least three characters back," you're asking for arithmetic on positions, and the automaton doesn't have positions. Corn: And Java and ICU actually let you do bounded repetition in lookbehind. Herman: Bounded. You can say up to ten characters back. You can't say "any number," and you can't express "count since the last occurrence of a thing that could be arbitrarily far away." Some engines will let you fake it with a greedy lookahead that scans to the end of the string, but that's not a regular operation, it's a backtracking trick that happens to work on short inputs and turns into a nightmare on long ones. Corn: This is the



Herman (continued)

ReDoS territory. Herman: This is exactly ReDoS territory. Cloudflare had an outage in July of 2019 from a single regular expression in a web application firewall that went catastrophic on certain inputs. The matching time went exponential, the CPU pinned, and the whole edge network started failing health checks. Corn: One pattern. Herman: One pattern. That's the cost of the features that make an expression non-regular. You get power, and you pay with unpredictable time. RE2 refuses to pay, and that's why it drops them. Corn: So the boundary isn't "the engine isn't good enough yet." Herman: It's not. It's a theorem. You can add more memory to the machine, you can add more states, but a finite automaton is finite by definition, and the capitalization rule needs unbounded counting. It will never be a regex. Corn: The character is not the first or second since the last period. Herman: Never. And that's the thing Daniel ran into in his own dictation workflow, which I think is the perfect illustration, because he wasn't doing theory, he was trying to fix his emails.



Corn

Let's get to the phone, because the engine is the easy part. On Linux, what does he actually have? Herman: He has everything. He can pick PCRE2, he can pick RE2, he can use GNU grep, he can use sed, he can drop into Python or Perl or Rust or Go. And more importantly, he owns the text buffer. Corn: Meaning. Herman: Meaning if he writes a script, the script reads the whole file, has access to every character, can maintain arbitrary state, can count, can look at the entire document, can rewrite it, can save it. The regex library is a component in a program he controls. Corn: On Android. Herman: On Android, Whisperian only sees what the OS exposes through the Accessibility service. The active text field, and the cursor position. It doesn't reliably get the whole document. It can't rewrite arbitrary parts of it. It can insert text at the cursor, and in some field types it can do more, but the fundamental view is partial. Corn: So the constraint isn't "Android's regex engine is weaker." Herman: The engine is actually quite capable. It's java.util.regex.Pattern, backed by ICU, International Components for Unicode. ICU's regex engine is Perl-derived. It supports lookahead, lookbehind, backreferences, named capture, atomic groups. The things it doesn't support are recursion and balancing groups. Corn: So it's closer to PCRE than to RE2. Herman: Much closer. If Daniel's problem were an engine limitation, he'd have options. It isn't, and he doesn't. The problem is that the app can't see enough text to make the decision, and even if it could see it, the decision itself is beyond regex. Corn: Two separate walls. Herman: Two separate walls, and they both happen to be in the same place.



Corn

What are the Android-specific pitfalls, then? Herman: The one I'd lead with is the Unicode-aware shorthand classes. In Android's ICU engine, backslash w matches letters, digits, and underscore, but the letters it matches include accented letters and letters from other scripts. On desktop Java, depending on how the pattern is compiled, it may only match ASCII by default. Corn: So the same pattern behaves differently. Herman: Same pattern, same Java syntax, different result, because one runtime says "letter" means a to z and the other says "letter" means any Unicode letter. The example that gets cited is a string of capital letters with an accented I in it, run through a replace that strips non-word characters. On Android the accented letter survives, because it's a word character. On



Corn (continued)

desktop Java it may not. Corn: And you'd never notice. Herman: You'd never notice until the day it matters, and then you'd spend an afternoon trying to figure out why your rule works on your laptop and doesn't work on your phone. Which, for a text-replacement rule that runs on every dictation you type, is a real problem. Corn: What else? Herman: The architecture. The Accessibility service is how the app gets into other apps' text fields, and that means it lives with every quirk of the Android input stack. Field types. Whisperian skips password and number fields, which is sensible, but it means your rule doesn't run there and you might not know why. IME behavior. Battery-saving. Their own site lists detailed exemption instructions for devices with aggressive battery-saving features, which tells you how much of the work is fighting the OS instead of writing patterns. Corn: On Linux you don't have an exemption list. Herman: On Linux there's no OS between your script and the text. You call the function, you get the string, you return the string. The platform doesn't have opinions.



Corn

Here's the question Daniel flagged and couldn't answer. Do the text replacement rules run before or after the LLM post-processing? Herman: I went looking, and it isn't documented anywhere I can find. Not on their site, not on the Play listing. Corn: So it's ambiguous. Herman: It's ambiguous, and it matters a lot. If your rules run first, they see raw ASR output, so um and uh and erm are actually in the text, and your disfluency regex does its job. If they run after the cleanup model, the model may have already stripped the fillers, and your rule is mostly idle. Corn: And if you're relying on a rule for something the model changes, order determines whether the rule fires. Herman: Order determines everything. If the model capitalizes something and then your rule tries to lowercase it, you win or lose based on sequence. Corn: Did you ask him? Herman: I didn't, and I don't think he knows either. It's the kind of thing you find out by experiment. But the fact that the documentation doesn't say is itself a design signal. Corn: It suggests the pipeline isn't a fixed order, or it's configurable, or it's an oversight. Herman: Any of those. But for someone writing rules, the safe assumption is that you can't assume. Write rules that are idempotent where you can, and test both orders if you care.



Corn

So does the LLM make any of this obsolete? Herman: The evidence says no, and it's better than I expected. There's a paper on log parsing, DeepParse, that uses an LLM to mine reusable regex patterns from log formats, then applies those patterns deterministically at runtime. Corn: So the model does the reasoning, and the regex does the execution. Herman: That's the phrase from the paper. "Separating the reasoning phase from execution." And the result is ninety-seven and a half percent average parsing accuracy, thirty percent fewer false alarms, and thirty-six percent lower latency than the LLM-only baseline. Corn: Lower latency is the obvious benefit. Running a compiled pattern is essentially free compared to a model call. Herman: And the accuracy is the interesting part. The model is better at figuring out what the pattern should be. The pattern is better at applying it consistently, thousands of times, without drifting. Corn: There's a financial extraction paper too. Herman: Same shape. Regex matching combined with an LLM as judge, dropping apparent subject hallucination from sixty-five percent to under two percent. Corn: That's a huge gap. Herman: It's the difference between asking a model to



Corn (continued)

both find the fields and confirm them, and asking a model to confirm what a pattern has already found. The pattern narrows the space, the model judges the candidates. Corn: And the counter-evidence. Herman: A paper called "When the Ruler is Broken" makes the other side of the case. A strict regex parser reported zero percent threat accuracy where a fuzzy parser recovered seventy-six percent. Corn: Zero. Herman: Zero, because the input format had drifted slightly and the strict parser matched nothing, and it failed silently. That's the danger with regex as an evaluation tool. It's a precision instrument, and it doesn't tell you when the thing it's measuring has moved.



Corn

The show's own pipeline does this. Herman: It does. There's a regular expression pass in our script editing that runs before text-to-speech, and it catches a handful of things before the review agent ever looks at the text. Corn: Which is the same pattern. Herman: Same pattern as DeepParse. The model reasons, the regex executes, and the order matters because the cheap deterministic pass handles the mechanical cases and frees the expensive semantic pass for the judgment calls. Corn: This is not a novelty. Herman: It's mainstream. It's what the research is converging on, and it's what production pipelines were already doing by intuition. The point of all of these papers is not "regex beats LLMs" or "LLMs beat regex." It's that they're different tools that solve different parts of the same problem, and the interesting question is which layer owns which job.



Corn

Which brings it back to Daniel's capitalization. Herman: It does. And I want to say something specific about it, because the answer isn't "impossible, give up." The answer is "put it in the right layer." Corn: Which layer. Herman: If the app can see enough text, a small local model could do it. A sentence-boundary model, or a tiny transformer that looks at a window around the cursor and decides whether the sentence is ending or continuing. That's a semantic decision, and semantics is what models are for. Corn: Or a change to how the app segments the dictation. Herman: Also plausible. If the app knew that the pause was a continuation and not a sentence end, it could just not capitalize. That's arguably the cleanest fix, because it's upstream of everything. Corn: Or a post-processing prompt. Herman: Or that. If the cleanup model gets the pause marker in its context, it can decide case. That's the least invasive option and it uses infrastructure Daniel already has. The point is that all three options live above the regex layer, and none of them replaces regex, they just take the one job regex can't do.



Hilbert

The man at the stall on Agrippas had a book. Corn: A book. Hilbert: Forty years of patterns, handwritten, indexed by what went wrong. Every customer who ever brought him a problem, he wrote the pattern down, and the name of the customer, and the date. Corn: And you read this book. Hilbert: I copied page two hundred and eleven. That's the one for the postal codes. He wouldn't sell me the book, but he'd sell me the page, four hundred



Hilbert (continued)

shekels, and I still have it. Corn: Four hundred shekels for one page. Hilbert: It's a good page. Herman: What was on it? Hilbert: The pattern. And a note about the man from the post office who came in 1998 and never paid. Herman: Did the pattern work? Hilbert: It works. It's the one I use for the invoices. The trick in it is that you never trust the digit that comes after a dash. Corn: You never trust the digit after a dash. Hilbert: Never. You put it in a capture group and you check it against the total. If the total doesn't match, the pattern is lying to you. Corn: Herman, he's describing checksum validation. Herman: He is describing checksum validation with a four-hundred-shekel piece of paper.



Hilbert

The other thing is the stall itself. It's not a stall anymore. He moved it to a kiosk near the shuk, and he still has the book, and he still won't sell it. The page I have is a photocopy of a photocopy, and the third-generation copy has a smudge where the post office man's name used to be. Herman: Does the smudge matter? Hilbert: No, the pattern is above it. Corn: So the rule survives the damage. Hilbert: The rule survives everything. That's the point of writing it down.



Corn

So if regex can't do the capitalization, and it can't be made to, then where does that job go? Herman: My guess is a small local model, running on the phone, looking at a short window around the cursor. Corn: Because the decision depends on semantics. Herman: Because it depends on whether the sentence is ending, and that's a judgment, not a match. Corn: And as on-device models get cheaper, the boundary moves. Herman: The boundary moves, but it doesn't disappear, because the regular and non-regular distinction is a theorem, not a hardware limit. You can put a bigger model on the phone, you can put a faster one, and the counting problem stays exactly where it is. Corn: So the practical lesson is to stop trying to make one tool do everything. Herman: Use the pattern for the mechanical work, use the model for the judgment, and know which is which before you write the rule.



Corn

That's the show. Thanks to Hilbert Flumingtop for producing, and for the page. Herman: If you enjoyed this one, try episode ten, How ASR Went From Frustration To … Whisper Magic; episode fifty-seven, From Lawyers in Limousines to Developers in Their PJs; and episode eight, Building Your Own Whisper. This has been My Weird Prompts. Corn: If you want to send us your own prompt, you can do it on Telegram at t dot me slash MWP listener bot, and if you enjoyed this one, leave us a review wherever you're listening. Herman: We'll be back soon. Corn: See you then.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **Whisperian official site, Issue N° 801, dated 2026-03-12. [primary]**
   https://whisperian.app

2. **Android Pattern API reference. [primary]**
   https://developer.android.com/reference/java/util/regex/Pattern

3. **Android libcore Pattern source (ICU-backed). [primary]**
   https://android.googlesource.com/platform/libcore/+/refs/heads/master/ojluni/src/main/java/java/util/regex/Pattern.java

4. **Whisperian Play Store listing mirror, updated 2026-09-23.**
   https://apppage.net/preview/app.whisperian.client

5. **AppBrain analytics, data updated 2026-10-10, version 0.26.0.**
   https://www.appbrain.com/app/whisperian-ai-voice-to-text/app.whisperian.client

6. **WhisperianHQ GitHub org, created 2026-02-05.**
   https://github.com/WhisperianHQ

7. **Mart Traagel, seventy-five-year history of the regular expression, 2026-05-22 (updated 2026-07-05).**
   https://mart.traagel.dev/blog/what-does-grep-stand-for/

8. **Wikipedia, regex origins (Kleene 1951).**
   https://en.wikipedia.org/wiki/regular_expression

9. **Russ Cox, Regular Expression Matching Can Be Simple And Fast.**
   https://swtch.com/~rsc/regexp/regexp1.html

10. **engine feature comparison (ICU, Java, RE2, PCRE).**
    https://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines

11. **finite-automaton memory limits.**
    https://stackoverflow.com/questions/133601/can-regular-expressions-be-used-to-match-nested-patterns

12. **backreferences and non-regularity.**
    https://cs.stackexchange.com/questions/2557/how-to-simulate-backreferences-lookaheads-and-lookbehinds-in-finite-state-auto

13. **JavaScript regular expressions are in fact not regular.**
    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions

14. **Android Unicode-aware `\w`.**
    https://stackoverflow.com/questions/30185139/java-regex-working-differently-on-android-than-in-java

15. **Tabular PDF Information Extraction with Local LLMs, hybrid regex+LLM.**
    https://arxiv.org/abs/2604.00003v2

16. **DeepParse: Hybrid Log Parsing with LLM-Synthesized Regex Masks.**
    https://arxiv.org/abs/2604.20553v1

17. **LLM-based Triplet Extraction from Financial Reports, regex + LLM-as-judge.**
https://arxiv.org/abs/2602.11886v1

18. **When the Ruler is Broken, parsing-induced suppression.**
https://arxiv.org/abs/2605.07293v1

19. **Mind the Pause, LLM disfluency correction (ACL 2026).**
https://arxiv.org/abs/2605.12242v1

20. **End-to-End Speech Recognition and Disfluency Removal.**
https://arxiv.org/abs/2009.10298v3