Emacs regex compilation

future directions for expressive pattern matching

Danny McClanahan

EmacsConf 2024

Outline

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer

Who are you?

Danny McClanahan:

- failed an independent study course in undergrad attempting to make a C compiler because (as my prof *specifically* warned me against!) I got stuck on the parser.
- spent the next several years realizing that was actually what I should have been doing the whole time.

Why are you here?

I have a lot of feelings about text and I'm making it everyone else's problem!

I ♥ emacs-devel

- spent a lot of time learning much of this from `emacs-devel` 🤓
  - was super confused about. . . a lot 😅
- thought this would be a matter of just updating the regex engine to use modern techniques 🧬 ⚛ ⚕
  - . . . turns out the emacs regex engine has features that don't exist in other engines![1] 🔥 🔮
- then i learned about other larger goals for the regex engine! 🧩
  - . . . which happened to overlap a lot with my own research interests 😈

[1] especially non-contiguous input, syntax-aware matching, and multibyte encoding

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer[2]

What is a regular expression?

No one's really sure!

When and how does implementation match formal theory?

Formal theory is mostly invoked to post-hoc justify design decisions instead of expanding expressive power!

[2] This section is adapted from "The Four Eras of Regex" by Prof. Jamie Jennings at NCSU: `https://jamiejennings.com/posts/2021-09-23-dont-look-back-2/`.

How did this happen?

Regex began as an investigation into theories of *formal languages*.

- Many features were added on to implementations to improve the practical user experience.
  - While the people adding these features were often academics, they were still very interested in building practical tools!
- This led to feature development beyond the range of formal theory (awesome!).
  - However, this also means that functionality becomes poorly-specified and implementation-defined.
  - Formal theory couldn't keep up!

The 1980s: Moonwalking and Backtracking

1983 Michael Jackson demonstrates the *moonwalk* in the music video for "Billie Jean".

1986 *Backtracking* is developed to simulate `egrep`-style regular expressions.

Backtracking Away from Formal Theory

- Backtracking ends up being a fantastic way to implement many more features!
  - . . . but this is really where the break from formal theory begins.
- Perl in particular adds a whole host of functionality!
  - . . . but this locks people into the specifics of perl as a language to perform many text manipulation tasks.[3]

[3] Much bioinformatics code still uses perl!

Conjecture

Formal theory remains largely concerned with incremental improvements to artificial benchmarks, and much less with *expanding* models to cover actual user needs.[4]

[4] This means they're also much slower than they could be if they *listen closely*.

Backtracking Away from Backtracking

By the 1990s, non-backtracking engines are created.

- Extant ones include `RE2`[5], `hyperscan`[6], and rust `regex`[7].
- These make use of the earlier automaton models with linear runtimes for well-specified search tasks.
  - However, they intentionally do not cover anything beyond regular linguistic complexity.

[5] See `https://docs.rs/re2` for the only rust wrapper that will ever be blessed by its maintainer.

[6] See `https://docs.rs/vectorscan-async` for the only rust wrapper that builds it for you.

[7] Of which more will be said later.

Complexity Theory vs Complexity Practice

So what happens if you need to do more?

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer

What are regexps used for?

All variety of text search and parsing tasks!

How do Emacs users use them?

As an auxiliary form of logic, to construct the user-level grammar for human thought that Emacs provides: text as input and output.

Why is Text Powerful?

The reason text programming languages are successful is because text is both input (readable) and output (writable).

- This makes text an extremely empowering and accessible framework to *navigate* and *manipulate* program code.

Why is Text Powerful?

[2: Hidden Dependencies]

If you are unable to modify or deploy your code without employing an opaque external system, then you have a hidden dependency.

- This opaque external system can then exert arbitrary control over your programming output.

Why is Text Powerful?

[3: Locality]

If you cannot reproduce a system *locally*, it becomes an opaque external system.

- e.g. GUI IDEs, cloud services, Large Language Models[8]

[8] all Microsoft products

Why is Text Powerful?

Text is local.

Emacs ❤ Text

- Emacs is a text editor which implements much of its own logic and user interface via text.
  - This is why we have elisp, a language tightly integrated with text operations from the editor.
- Because text forms UI, *parsing* and *text search* can be employed not just to edit code, but to construct a user interface from text input.
  - This means that language-level mechanisms for text such as the regex engine can be extended into the user interface.

Who Says Text is Empowering?

[1: You're Not Smart Enough]

Not everyone thinks text is empowering!

- Formal theory thinks nobody should be allowed to parse text without their tools!

Who Says Text is Empowering?

[2: Don't Parse HTML with Regex]

"Everyone knows" not to parse HTML with regex, because regex (alone!) isn't sufficiently powerful to parse HTML. But:

1. Nobody is parsing HTML with a single massive regex!
2. Regex + mutable state can achieve arbitrary linguistic complexity!
3. Regex search for a specific substring is much faster than parsing everything up front!

Who Says Text is Empowering?

[3: C Lexer Hack]

Turns out those tools aren't *too* powerful, they're actually not powerful *enough* for practical inputs!

- This is why I got stuck on the parser in that independent study course!

Emacs Says So!

This isn't remotely a concern for Emacs code, which regularly uses regexps to parse HTML and other programming languages! How?

text properties write state to the text itself[9]

syntax parsing regex engine is aware of this via syntax classes[10]

jit-lock-mode use smart heuristics to only reparse what's changed[11]

[9] Not unlike the tape of a turing machine!

[10] \b, \<, etc.: see `https://www.gnu.org/software/emacs/manual/html_node/elisp/Regexp-Backslash.html`.

[11] This might just be fontification, as opposed to the work done in `syntax-ppss`.

Formal Theory: Right for the Wrong Reasons

There *are* actually reasons to avoid using regexps to parse text!

- Regexps may have extremely non-obvious dependencies on parse context.
  - A non-greedy match may be correct when invoked in a restricted context, but may become subtly incorrect when used more generally.[12]
- While text properties and buffer-local variables can retain the state necessary to parse non-regular languages, coordinating that state can be error-prone.
  - Since there are no existing formalisms to link regex with external state[13], it can become extremely difficult to reproduce the precise internal state which generates a logic bug in an elisp mode.

[12] For example, `(_<.*?):` could match a symbol before a `:` (like `a:` in JavaScript), but could unintentionally match string properties like `"a:b":` as `a:` too!

[13] Composing automata with other parse state is one of the subjects of my research.

tree-sitter

In fact, `tree-sitter` (since Emacs 29) was created to solve this problem *for well-specified language definitions*.

- It is a highly constraining formal tool!
- And it means you now depend on:
  - The tree-sitter grammar for your language.[14]
  - The `tree-sitter` library.[15]
- So I don't like it!
  - But for the specific task of parsing a programming language, it happens to solve a lot of other problems at once.

[14] obnoxious to read and write

[15] does not have universal uptake within distros

So Why Use Regex?

So why are we talking about regex here? Mainly:

- Parsing programming languages is a very small subset of all text search/matching tasks!
- Regex can be directly manipulated by the user!

Regex as I/O

For the interactive experiences that Emacs excels at, regex provides a powerful language *for both input and output*:

- It can be synthesized hygienically from elisp code via `rx`, either statically at load time or dynamically at run time!
- It can be received or transformed from user input to specify powerful queries over complex data![16]

[16] See `helm-rg` and `telepathygrams` at end.

We Can Go Further...

...but this might require going beyond "regex" alone!

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer[17]

What is the Emacs regex engine?

It's a backtracking engine over multibyte codepoints, defined in `src/regex-emacs.c`.

How is it invoked?

In two ways:

- over a single contiguous string input,
- over the two halves of the gap buffer.

[17] This section is an unfortunately brief walkthrough through the current regex logic.

The compiled pattern is stored as an `re_pattern_buffer` struct from `src/regex-emacs.h`.

- In particular, `unsigned char *buffer` holds the instructions!
- Case folding uses the char table in `Lisp_Object translate`.

regex-emacs

[2: Match Loop]

The matching loop in `re_match_2_internal()` in `src/regex-emacs.c` goes vaguely as follows:

1. extract current and next char
   - perform multibyte varint decoding to iterate bytes
   - translate input characters via the `translate` case-folding char-table
2. read instruction from instruction pointer
3. big switch statement for the next instruction
   - if instruction uses syntax, read the syntax class[18] for the current character from the current syntax table
4. increment the instruction pointer[19]
5. if we've concluded a capture, write the end position to the C-level array `re_nsub`

[18] `https://www.gnu.org/software/emacs/manual/html_node/elisp/Syntax-Class-Table.html`

[19] unless instruction was a jump

Non-Contiguous Matching

Non-contiguous matching over the two halves of the gap buffer is supported by checking at each point whether we have progressed to the end of the first half, and then switching over to the second half.

- This allows the same code to be used for single-string search, as it simply avoids checking a NULL second pointer and only checks if we've reached the end of the first input.

Multibyte

- It turns out this actually isn't terribly relevant to the regex engine!
  - Or at least, it doesn't really differ from "standard"[20] Unicode regex matching.
- Emacs reads in data from whatever encoding into multibyte,[21] and the regex engine only acts upon this normalized encoding.

[20] There is no (real) standard (yet): `https://jamiejennings.com/posts/2021-09-07-dont-look-back-1/`.

[21] See `https://www.gnu.org/software/emacs/manual/html_node/elisp/Text-Representations.html`.

Is that all?

How much time[22] do you have?

[22] and space

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer[23]

How could we do regex better in Emacs?

- introspection
- optimization

How could Emacs do regex better than anywhere else?

- explicit control over linguistic complexity
- libraries of composeable patterns

[23] This section will describe several potential paths we might investigate, paraphrasing discussion from emacs-devel .

Separately-Compiled Regexps

Precompile regexps to enable more powerful compilation techniques.

Problem Emacs currently uses a fixed-size global compile cache.

Solution Create native elisp objects for regexps and match data.[24]

- Use `(make-regexp "...")` to explicitly compile a pattern string.
- All supported[25] methods in `search.c` can accept either a compiled regexp or bare string.

[24] I have demonstrated this in a test branch: `https://github.com/cosmicexplorer/emacs/tree/lisp-level-regex`.

[25] Literal search methods do not use the regex engine.

Separately-Compiled Regexps

Precompile regexps to enable more powerful compilation techniques.

Results Artificial benchmarks[26] show an improvement![27]

- . . . but I haven't been able to produce an apples-to-apples comparison yet.
- Syntax highlighting would be the most appropriate, but caching these compiles currently breaks fontification.[28]

[26] Using `test/manual/perf.el`.

[27] Using the native match data object produces no improvement over consing a list.

[28] Current guess is that it relies on buffer-local state not available when precompiled.

Match Over Bytes, not Chars

Compile patterns to byte-level automata, then iterate over bytes.

Problem Char-by-char varint decoding of multibyte[29] is comparatively slow.

- This is the reason go's "RE2" is much much slower than the C++ `RE2` library.[30]

Solution We can do this work at compile time instead![31]

- Generates a larger automaton in order to be able to think in terms of byte ranges.[32]

[29] No worse than UTF-8 in general, but we may be able to pipeline decoding somehow.
[30] Source: a very dear friend.
[31] This is already what we do for e.g. char-folding.
[32] This is a necessary prerequisite for SIMD instructions.

Explicit Control over Linguistic Complexity

Break apart the monolithic regex pattern interface into subroutines for specific inputs.

Problem[1] There's no way to validate that a given pattern isn't more complex than expected.

- This requires careful escaping to avoid accidentally triggering regex behavior.

Problem[2] There's no way to ensure Emacs uses faster algorithms[33] for less complex patterns.[34]

- This results in difficult-to-understand performance characteristics.

Problem[3] There's no way to specify different search semantics.[35]

- Instead, we have a single type of input and a single type of output.

[33] See `https://github.com/BurntSushi/rebar` for a fantastic discussion of techniques for regex performance.

[34] We perform a heuristic check for literal patterns, but only in some code paths.

[35] Such as allowing false positives, or matching against a set of patterns.

Explicit Control over Linguistic Complexity

Break apart the monolithic regex pattern interface into subroutines for specific inputs.

Solution[1] Single or multiple literals[36] can employ specialized SIMD algorithms to avoid reading every single byte one by one.[37]

Solution[2] Non-capturing patterns or patterns without backrefs[38] can use faster automata.

Solution[3] Collecting a sequence of matches for the same pattern can be done all at once.[39]

Solution[4] Matching against a set of patterns can be done more efficiently and ergonomically than combining with `\|`.

---

[36] Multiple literals is especially helpful for matching a set of keywords.

[37] This is used as a "prefilter" optimization in modern engines like `RE2` to avoid reading each byte one-by-one.

[38] These have recently been formalized: `https://jamiejennings.com/posts/2023-10-01-dont-look-back-3/`.

[39] Overlapping matches can be supported for ambiguous cases, instead of choosing longest or shortest only.

Lisp Regexp Library

Expose a lisp-level library for regexp matching.

Problem The compiled form of the regexp in `re_pattern_buffer` can be *executed*, but not really *introspected*.

- No form of "IR": this also contributes to the difficulty of composing patterns together.

Solution[40] We have `libgccjit` now: why not implement the regex engine itself in lisp?[41]

Results *(postulated)*:

- Integration into `pcase` could achieve a form of type safety along with interleaving lisp-level matching logic.
- Biggest issue for optimization: lisp code (or native modules) can't access or operate on the separate halves of the gap buffer.

[40] Proposed by Pip Cet on `emacs-devel`.

[41] Alternatively, translate the regexp into lisp which we can then JIT.

Question

1. Who are you? Why are you here?
2. What is a regular expression? When and how does implementation match formal theory?
3. What are regexps used for? How do Emacs users use them?
4. What is the Emacs regex engine? How is it invoked?
5. How could we do regex better in Emacs? How could Emacs do regex better than anywhere else?
6. Do you have any concrete examples? Or are you just posturing?

Answer

Do you have any concrete examples?

Yes!

Or are you just posturing?

My posture is terrible![42]

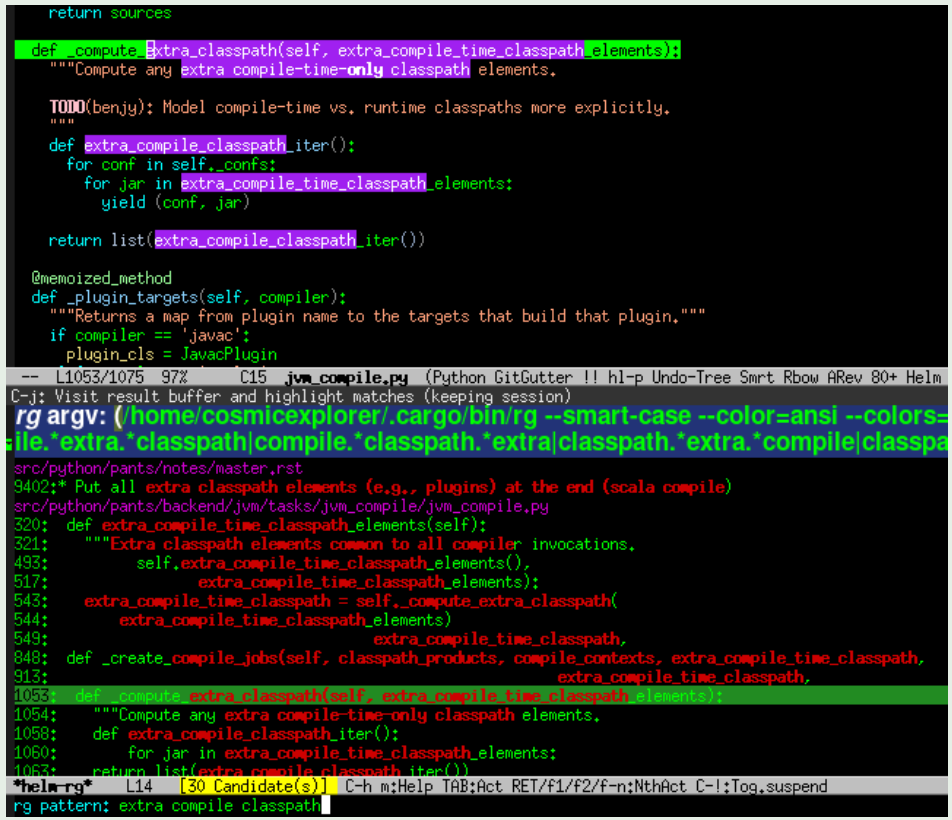
[42] Other translations have also been suggested by modern scholars, including *formidable*, as well as *awe-inspiring*.

helm-rg[43]

A code search tool similar to `M-x grep`, using `ripgrep`[44].

- Generates regexps from input:
  - `"a b"` => `"a.*b|b.*a"`.
  - Translates from PCRE to elisp regexps to highlight matches in the helm buffer.
- `M-b` enters "bounce mode", where matched lines can be edited directly.

Example (pattern generation)



[43] https://github.com/cosmicexplorer/helm-rg

[44] https://blog.burntsushi.net/ripgrep/

`telepathygrams`[48]

A (WIP) code search tool that precompiles a database to execute NFAs against.

I want to beat `ripgrep` by "cheating"[45] with a precompiled index.[46]

- n-gram indices have been done,[47] but I don't want to just find where to *start*–I want to execute the *entire search* against the index!
- This requires virtualizing NFA state so that it may be distributed:

  across time in parallel / across machines,

  & space in terms of *offsets* vs directly against the input data.
- This may fail, but it will be fun!

[45] There are other ways to cheat here too, like precompiling known queries.

[46] Inspired by `etags`, but with a more complex index for more general queries.

[47] e.g. Kythe: `https://kythe.io/docs/kythe-overview.html`

[48] `https://github.com/cosmicexplorer/telepathygrams`

RIP @junyer[49]

Paul Wankadia taught me everything.

[49] https://github.com/google/re2/issues/502

(point-max)

call me beep me if you wanna reach me[50]

text-mode

fedi @hipsterelectron@circumstances.run[a]

IRC @cosmicexplorer in #emacsconf[b] on irc.libera.chat

email dmc2@hypnicjerk.ai[c]

[a] also (sporadically) twitter & bluesky
[b] also (sporadically) elsewhere
[c] also dmcC2@ (still deciding which poetic license to use)

prog-mode

codeberg @cosmicexplorer

github @cosmicexplorer

[50] or hire me