Interview Notes

Meshal Binnasban

October 29, 2025

1 Questions

Q1: What is the motivation behind your approach?
to address the limitations of existing regex matching techniques, including state explosion in DFA-based methods, catastrophic backtracking in NFA-based methods, and expression growth in derivative-based methods.

DFA$\cdots$ **??** NFA$\cdots$ **??** Derivatives$\cdots$ **??**

Q2: How does the marked approach help?
In the marked approach, the regular expression size remains fixed. It keeps the structure of the regex fixed and moves marks through it as matching progresses. These marks represent how far the matcher has advanced inside each subexpression. When a character is read, the marks shift through the regex to indicate progress.

Q3: if you translate a regular expression into an automaton, then the matching is linear, why not just use earlier work instead?

yes, on the DFA, it might be linear time for matching, but if you translate a regular expression into a DFA, that might explode the state space, with worst case 2^n states. it is not that of a clear cut.

ultimately, we are looking at posix value extraction, longest leftmost values *which requires that we have a strategy put in place for disambiguation, currently we are using lists for disambiguation purposes to maintain the order of marks based on remaining suffixes but we are aiming to use sets???*

In the other hand, DFA based engines do that too but they are not straightforward and clear on the number of states and lack formal correctness proofs. we aim to provide correctness proofs for lexing as well as acceptance which we are almost done with.

There is also the case of extending constructors such as the negation constructor, which is relativly straightforward in the derivative but not so in DFA based engines. we are not completely sure it is straightforward in the marked approach but that needs to be investigated. but it is not a straightforward with producing DFA for the complement language. *maybe add something about the correspondence between derivatives marks???* not just the negation but also bounded repetitions and intersections.

Q4: Why not just use existing regex engines?
Existing engines still face size-related limitations, DFA-based engines such as Rust attempt to avoid constructing the full automaton by starting from the initial state and building DFA states only as needed for the given input

string. However, for patterns such as r^n, the engine still needs to produce n copies of r's automaton. Although Rust uses size-checking to decide which regexes are "safe," this behaviour is not formally guaranteed, and some cases still slip through, approaching memory limits and running slowly.

Python, on the other hand, uses an NFA and does not generate a DFA. This design supports additional features like backreferences. While an NFA has linear size of states ($n + 1$ states), it can still require exploring many or even all states in implementation. Python uses depth-first search, which is fast when a match is found quickly, but if not, the engine might need to explore a large number of paths—or even the entire tree—before reaching a decision. If the input is incorrect, the search may still traverse the full tree, leading to catastrophic backtracking.

Q5: What about derivative-based matchers?

Derivative-based matchers are elegant and easy to reason with, and they have been proven correct for both acceptance and lexing. However, the Achilles' heel of the derivative approach is expression growth. Each derivative reconstructs new parts of the expression, particularly for SEQ and STAR, which can expand rapidly. Even with aggressive simplifications, the size can still grow, since equivalent subexpressions may appear at different levels of the expression tree and cannot be merged.

We performed a runtime test to compare the difference between derivatives with aggressive simplifications and Fischer's mark-based algorithm. Although, the test is naive,but it demonstrate the impact of the growth of derivatives, even under aggressive simplifications whereas marks don't grow the size of the regular expression. there is a considerable difference in behaviour. the executing of derivatives ran out of memory after 10,000, with the last successful case taking almost 22 seconds, while the mark-based algorithm finished executing all inputs with each case under 0.0035 seconds. Figure 5 in the report.

Q6: What exactly does POSIX mean in your work?

In our work, POSIX means that among all possible matches, the longest-leftmost match. It follows the definition of Tan and Urban (2023).

Q7: Why is value extraction important?

Value extraction records how the regex matched, not just whether it matched. It is useful in several areas, for example:

- Tokenization: for instance, splitting a source file into identifiers, numbers, and punctuation before parsing.

- Syntax highlighting. regexes are often used to detect keywords, comments, and string literals; reconstructing the match (e.g., a value like Left(Right(Empty))) tells the highlighter which span of text to colour.
- Lexing in compilers.

Q8: What is the difference between your work and Fischer's? Up to this stage, we have developed an algorithm that extends Fischer's marked approach so that marks carry strings, beginning with the full input string. ultimately, we are aiming at extending the algorithm to include POSIX value extraction. Fischer's work focused on matching only while our work extends it to also perform value extraction with POSIX disambiguation.

(Summary of contribution) This approach aims to avoid the limitations of state explosion, backtracking, and derivative size growth, while providing not only matching but also value extraction with POSIX disambiguation.

2 Notes

2.1 proofreaded notes

String matching is usually done by constructing an NFA from a regular expression, then compiling it into a DFA, and finally matching character by character. This process can be efficient for small patterns but may lead to a state explosion in the DFA, reaching up to 2^n states for expressions like r^n. Engines that rely on these constructions use various strategies to manage this growth, such as building DFA states lazily or exploring nondeterministic paths, each with its own trade-offs.

2.1.1 Flow

The classical path is regex $\rightarrow$ NFA $\rightarrow$ DFA. We construct an NFA from a regex. Then we apply subset construction to obtain a DFA. This "regex $\rightarrow$ NFA" stage is a helpful stepping stone before determinization.

2.1.2 How string matching is done (DFA route)

We first produce an NFA. We then build a DFA. Finally, we match character by character on the DFA. In a DFA, the running time is naturally linear in the input length: each character triggers a single transition, and we can tell if we end in an accepting state. The main limitation is automaton size.

2.1.3 Rust's regex engine (claim and scope)

Rust's engine does not support lookaround or backreferences. In return, it offers a linear-time guarantee for the regex and the text, but only for approved patterns. With patterns like R^n, the situation deteriorates: translating R^n to a DFA effectively chains n copies of the DFA for R. The guarantee is tied to a size heuristic. It holds only within the approved set. Some hard cases slip the heuristic, can approach memory limits, and run slowly.

2.1.4 Automata size and lazy construction

When generating a DFA, the number of states can reach 2^n in the worst case, with n proportional to the regex size. Rust does not build a complete DFA up front. It constructs states lazily and on demand, often yielding a small DFA on typical inputs. For adversarial inputs, one may still end up generating essentially all 2^n states.

2.1.5 Python/PCRE engines (NFA with backtracking)

Python does not generate a DFA. It executes an NFA with backtracking. A standard construction yields an NFA with size linear in the regex. For R^n, this means n sequential copies of the fragment for R. Because NFAs are nondeterministic, matching may need to explore many paths. Two strategies are common:

- Depth-first search (DFS): follows one path, often fast when acceptance is nearby, but susceptible to catastrophic backtracking.
- Breadth-first search (BFS): explores level by level; in the worst case memory grows exponentially.

A practical reason for this design is feature support: Python/PCRE include backreferences, which preclude a clean DFA route.

2.1.6 Backreferences and complexity

Backreferences push matching beyond regular languages. The general matching problem with backreferences is NP-complete. Backreferences enforce equality of arbitrarily long substrings, so there is no general subset construction to a finite DFA.

2.1.7 Derivatives

Derivative-based matchers do not support backreferences. They are typically fast in common cases. They are elegant and proof-friendly, with correctness proofs for acceptance and lexing. The Achilles' heel is potential expression growth, especially through SEQ (product). We apply simplification to curb this growth. There are no worst-case guarantees. Duplicated subexpressions can appear at different levels or positions, where local simplifications cannot merge them.

2.1.8 Example note (backtracking)

To illustrate catastrophic backtracking in Python-style engines, (${}^{(}a^{*}) + b$ on an all a input).

2.2 raw notes

$1 - Regex- > NFA- > DFA$. constructing from REGEX to NFA is better stepping stone to then convert into DFA. how is string matching doen? String matching is done by producing NFA then DFA then match character by character. in DFA, some claim matching is linear to input? at first, it might make sense since you have the string and you always know where to transition in a DFA, to know if you are in an accepting state. rust claims "they dont have lookaround and backreferencing in exchange to linear time with respect to the size of the regex and the search text. however, if you feed it a specific regex such as one with power to n (r^n) it doesnt. if we translate regex of a power of n to DFA, it will copy the r DFA into n copies and put them together. rust then says the it is linera for approved regex, limited by size which they determine using an algorithm. they give this strong property but only in the case they deem ok. ofcourse some cases slips thier algorithm, it might not exhaust the memory as some regex do, but just about exhausting it and it is slow. - note : when generating DFA, size can grow 2^n. rust dont generate complete automata (because a small regex could exhaust memory) they generate DFA essentially dynamically (they generate the starting state and depending on the input string, usually relativly small DFA) but of given the incoreect string , you still have to generate all the 2^n states. python why perform bad on example of a**b? in python, they dont generate DFA, instead they generate NFA. there is an NFA proprety which is that the number of states is $n + 1$ if you have a regex with size n in the worst case, unfortunately you will have to generate n copies for r^n. if the matching is done in NFA, you might have to explore all states to find accepting states. because, in nfa, each state you dont know where exactly to go unlike DFA. to implement this (meaning matching regex using nfa), you can use 1- depth first search or 2- bredth first search. 1- in depth, explore one path and count on finding accepting state very quickly, which is fast most of the time. 2- in breadth search, to explore every level of the tree at the same time, sometimes you may need to explore the full tree, and the size grow exponentially for the memory. Downfall of python, they use this to include back-referencing, so that is why they use nfa, downfall is if the string don't match, you may need to explore the full tree, tricking the engine with depth first causes this. note- if backreference is added , then the matching problem could become np complete problem (or equivalent). most efficient algorithms for solving np problems are exponential algorithm (maybe an example such as backtracking heuristic algorithm), backreference changes the problem and there is no posibility of subset construction (explain why here). how about derivative: no backreferencing. fast in usual (such as examples given in lecture) but also has cases where it can explode in size. achilies heel of derivative that it can grow quickly and make it slower. to try and prevent this, simplification is done. unfortunately no gurantees that simplification works all the time, copies to be simplified might not be possible to find, if r and its copies are in different levels and positions in the tree and next to eachother, then simplification cannot be applied. as elegent and beautifully designed derivatives are, and easy to reason about and implement (and proofs exisit of it and its lexer as well which is something some other matcher implementations lacks), it still has this problem.