Meeting Questions

1. **What is the motivation behind your approach?**
   because existing regex matching techniques have limitations that we aim to address with this work: -state explosion in DFA-based methods, -catastrophic backtracking in NFA-based methods, -and expressions growth in derivative-based methods.

   DFA,Q... 4

   NFA,Q... 4

   Derivatives,Q... 5

2. **How does the marked approach help?**
   In the marked approach, the regular expression size remains fixed. It keeps the structure of the regex 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.

3. **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 construct a DFA from a regular expression, that might explode the state space, with worst case 2^n states. It is not that clear-cut.

   Ultimately, we are looking at POSIX value extraction, longest-leftmost values *which requires a disambiguation strategy, 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???* 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 relatively straightforward in the derivative but not so in DFA-based engines. We are not yet sure it is straightforward in the marked approach; this requires further investigation. It is not straightforward to produce a DFA for the complement. *maybe add something about the correspondence between derivatives marks???*

Not just the negation constructor but also bounded repetitions and intersections.

4. **Why not just use existing regex engines?**
   Existing engines still face size-related limitations.

   -DFA-based engines such as Rust's regex engine attempt to avoid constructing the full automaton by starting at the initial state and building states only as needed for the given input string. However, for patterns such as r^n, construction may effectively duplicate r multiple times (up to n in the worst case). 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 a linear number of states ($n + 1$), it can still require exploring many——or even all——states in practice. 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.

5. **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.

6. **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).

7. **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.

8. **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.

9. **(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.