TREX-Terminal RegEx Engine

Sathvik Reddy Bhavanam

1 PROBLEM STATEMENT

Build a **Regular Expression (Regex) Engine** that can parse, compile, and execute regular expressions against input strings, supporting a rich subset of standard regex syntax (similar to Python or JavaScript).

The engine should:

- Accept a regex pattern and an input string.
- Compile the pattern into an internal representation (e.g., NFA or DFA).
- Execute the pattern efficiently to determine matches.
- Return match results and group captures.

2 FEATURES AND SYNTAX SUPPORT

Feature	**Syntax**	**Description**
Literal Match	`abc`	Matches string "abc"
Wildcard	`.`	Matches any single character
Zero or More	`a*`	0 or more repetitions of "a"
One or More	`a+`	1 or more repetitions of "a"
Optional	`a?`	"a" occurs 0 or 1 time
Alternation	`a|b`	Matches "a" or "b"
Grouping	`(ab)`	Group "ab" together
Character Class	`[abc]`	Any of "a", "b", or "c"
Range	`[a-z]`	Any lowercase letter
Negation	`[0-9]`	Not a digit
Anchors	`^`, `$`	Start or end of string
Escapes	`\d`, `\w`	Digits, word characters, etc.

Table 1: Regex Syntax Features Supported

3 TEST CASES WITH EXPLANATIONS

Each of the following cases should be supported and validated by your engine:

Test 1. Literal Match
Pattern: `abc`
Input: `abc` ⟶ Match
Explanation: Exact match of characters.

Test 2. Wildcard
Pattern: `a.c`
Input: `abc` ⟶ Match
Explanation: "." matches any character.

Test 3. Kleene Star
Pattern: `ab*c`
Inputs: `ac`, `abc`, `abbbc` ⟶ Match
Explanation: "b*" matches zero or more "b".

Test 4. Plus Quantifier
Pattern: `ab+c`
Inputs: `abc`, `abbbbbc` ⟶ Match, `ac` ⟶ No match

Test 5. Optional Character
Pattern: `colou?r`
Inputs: `color`, `colour` ⟶ Match

Test 6. Alternation
Pattern: `cat|dog`
Inputs: `cat`, `dog` → Match, `cow` → No match

Test 7. Grouping and Repetition
Pattern: `(ab)+`
Inputs: `ab`, `abab` → Match

Test 8. Character Classes
Pattern: `h[ae]llo`
Inputs: `hello`, `hallo` → Match, `hollo` → No match

Test 9. Range
Pattern: `[a-z]+`
Input: `abcxyz` → Match, `ABC` → No match

Test 10. Anchors
Pattern: `^abc$`
Input: `abc` → Match, `xabc`, `abcx` → No match

Test 11. Shorthand Classes
Pattern: `\d+`
Input: `1234` → Match, `abcd` → No match

Test 12. Group Captures
Pattern: `(a)(b)(c)`
Input: `abc` → Match with Groups 1 = a, 2 = b, 3 = c

4 THEORY BEHIND THE ENGINE

4.1 REGULAR EXPRESSIONS AND AUTOMATA

Regular expressions define regular languages and can be represented using finite automata. Your engine will use this principle to convert patterns into automata.

4.2 COMPONENTS

- **Lexer**: Tokenizes the regex pattern.
- **Parser**: Builds an Abstract Syntax Tree (AST) from the tokens.
- **NFA Generator**: Uses Thompson's construction to build a Non-deterministic Finite Automaton (NFA).
- **NFA Simulator**: Simulates the NFA using epsilon-closure and state transitions.
- **(Optional) DFA Optimizer**: Converts the NFA to a Deterministic Finite Automaton (DFA) for faster matching.

4.3 MATCHING PROCESS

1. Compile regex into an NFA.
2. Use simulation to check if the input string can reach an accepting state.
3. Track positions and groups while traversing transitions.

5 DELIVERABLES

- Full implementation of the regex engine.
- Support for all features described.
- Match results with group information.
- Unit tests covering all listed cases.
- Optional: DFA optimization, named groups, or lookahead/lookbehind.