A tool for optimizing regular expressions.

By Henrik Vestermark (hve@hvks.com)

Abstract

Regular expressions (regex) are a fundamental tool in software development, used for pattern matching and data manipulation across various programming languages. However, their complexity, especially with complex patterns, can lead to inefficiency and performance issues. This paper introduces a regex optimization tool that systematically decomposes regular expressions into their core components—such as quantifiers, groups, alternation, and character classes—while applying advanced techniques for simplification and optimization. By leveraging an Abstract Syntax Tree (AST)-based approach, the tool enhances the readability and efficiency of regex patterns, helping developers avoid common pitfalls like excessive backtracking and redundant operations. In addition, this paper outlines practical strategies for optimizing regex, providing developers with the tools and knowledge to write more efficient, maintainable, and effective regular expressions.

Introduction

Regular expressions (regex) are powerful tools for pattern matching and text manipulation across various programming languages. They offer a concise way to search, extract, and modify text by specifying patterns for matching sequences of characters. However, while expressive, regex patterns can quickly become inefficient or overly complex, leading to performance issues, especially in large-scale data processing.

Regex optimization involves systematically simplifying and improving regex patterns to enhance efficiency, readability, and maintainability without altering their intended behavior. Refining complex expressions reduces the overhead caused by excessive backtracking and redundant operations, ensuring faster execution times and lower resource consumption.

This paper presents an approach to regex optimization that relies on a formal structural understanding of regular expressions. By breaking down regex into its fundamental components—such as literals, character classes, quantifiers, groups, and alternations—and applying optimization techniques, we can significantly streamline and simplify regex patterns. Using tools like Abstract Syntax Trees (ASTs) to represent and manipulate regex structures programmatically allows for a deeper analysis, paving the way for automated optimization techniques.

The following sections outline the core concepts of regex optimization, introduce a structured BNF (Backus-Naur Form) to represent regular expression syntax, and describe methods for transforming inefficient patterns into optimized versions. This paper aims to provide a robust framework for developers and automated tools to optimize and manage complex regular expressions effectively.

Change log

23-Oct-2024 Updated the documentation to reflect version 2 of the JavaScript optimizer function.

14-Oct-2024 Corrected certain inaccuracies and fixed some bugs in the code.

Contents

Abstract 1
Introduction 1
Change log 2
Core Concept of Regex Optimization 4
Optimizing regular expression 7
Optimizing Quantifiers 7
Leverage Character Classes 9
Optimize OR Conditions (|) 10
Utilize Non-Capturing Groups 11
Implement Anchors Strategically 11
Avoid Greedy Quantifiers 11
Minimize Backtracking 11
Use Lookaheads and Lookbehinds with Care 12
Lookahead 12
Lookbehind 13
Practical Example of lookbehind 13
The Regex optimizing tool 13
Why Use an AST? 14
Two-Phase Optimization of JavaScript Regular Expressions 15
Overview of the Two-Phase Optimization Approach 16
Node-Based Optimization (First Phase) 16
Literal Node Combination 16
Quantifier Simplification 17
Character Class Optimization 17
Removal of Redundant Groups 17
Removal of Empty Nodes 18
Global Optimization (Second Phase) 18
Redundant Anchor Removal 18
Alternation Simplification 18
Removal of Redundant Non-Capturing Groups 19
Global Quantifier Adjustments 19
Lookahead and Lookbehind Optimization 19
Example of optimization 20

Who is the optimizer tool for? 20
Further Improvement? 21
Conclusion 23
Other Useful online tools 23
1. Regex101 23
2. RegExr 24
3. RegexBuddy 24
4. Regex Pal 24
5. Regex Tester 24
6. Regular expression tester 25
Reference 25
Appendix 26
Function ASTtoString() 26
Function ASTtoRegex() 27
Function parseRegex2AST() 28
Function optimizeAST() 32

Core Concept of Regex Optimization

Regex optimization involves dissecting a regular expression into its basic elements and structures, such as literal characters, character classes, quantifiers, and groups. This process is crucial for developers who must debug or modify complex regex patterns. It clarifies what each part of the regex does and how it affects the matching behavior. Since regular expression syntax is complex and not always logical, it helped me by first establishing a formal BNF description of the syntax.

BNF syntax for regular expression

```
<regexp> ::= "/" <pattern> "/" [<flags>]
<pattern> ::= <concatenation> ("|" <concatenation>)*
<concatenation> ::= <element>*
<element> ::= <group>
            | <character-class>
            | <quantified>
            | <anchor>
            | <escaped-character>
            | <character>
<group> ::= "(" <pattern> ")"
          | "(?<identifier>" <pattern> ")"
```

```
        | "(?:" <pattern> ")"
        | "(?=" <pattern> ")"
        | "(?!" <pattern> ")"
        | "(?<=" <pattern> ")"
        | "(?<!" <pattern> ")"
<character-class> ::= "[" [ "^" ] <character-class-body> "]"
<character-class-body> ::= ( <character-range> | <character> )*
<character-range> ::= <character> "-" <character>
<quantified> ::= <element> <quantifier> [ <lazy-modifier> ]   // Separating quantifier and lazy
modifier
<quantifier> ::= "*" | "+" | "?" | "{" <digits> [ "," <digits> ] "}"
<lazy-modifier> ::= "?"                                  // Lazy quantifier modifier
<anchor> ::= "^" | "$" | "\b" | "\B"
<escaped-character> ::= "\\" <character>
<character> ::= any single character except special characters
            | <escaped-character>
<identifier> ::= <letter> (<letter> | <digit> | "_")*
<flags> ::= <flag>*
<flag> ::= "g" | "i" | "m" | "s" | "u" | "y"
```

It is now easier to see how a regular expression can be broken down into simpler components. The JavaScript function I developed for the breakdown is listed in the appendix.

The main group for decomposition consists of:

- Escaped characters are those preceded by a backslash \, which changes the usual meaning of the character following it. For example, \n represents a new line, and \d matches any digit. They are used to enable the inclusion of special characters in the regex pattern as literal characters or to signify special regex functions (like \b for word boundaries).

- Character sets, enclosed within square brackets [], match any one character from a set of characters. For instance, [a-z] matches any lowercase letter, and [^a-z] matches any character that is not a lowercase letter. They allow the regex to be more flexible and concise by enabling the matching of characters from a defined set, enhancing pattern-matching capabilities within strings.

- Quantifiers determine how many instances of a preceding element (like a character, group, or character class) are needed for a match. Standard quantifiers include * (0 or more), + (1 or more), and ? (0 or 1). Quantifiers are critical for specifying the number of occurrences in the string that match the preceding element, allowing regex patterns to match varying text lengths. In its core form, the quantifiers are greedy, meaning they try to consume as much input as possible for matching. Its counterpart, lazy, means it will consume the least input to fulfill the match. Laziness is indicated by adding an extra ? to the beforementioned quantifiers.

- Braces are used as quantifiers to specify the number of times a pattern element must appear. They can define a fixed number {n}, a range {n,m}, or a minimum number of times {n,} a pattern must occur. This type of quantifier helps match a precise number of repetitions in a pattern, allowing for precise control over the occurrences of a component. This is particularly useful in matching specific formats of data like dates, serial numbers, or parts of codes. The simple quantifiers can also be written using braces as: {0,} for *, {1,} for +, and {0,1} for ?

- Alternation, denoted by the pipe symbol |, acts like a boolean OR. It matches the pattern before or after the |. For instance, cat|dog matches cat or dog. Alternation allows for matching one of several possible parts, making it worthwhile to include multiple options within a single regex pattern, enhancing flexibility and scope.

- Groups in regular expressions treat multiple characters as a single unit. They are enclosed in parentheses (). There are two forms of groups: capturing and non-capturing groups.

    - Capturing groups save the part of the string matched by the part of the regex inside the parentheses. This allows the user to extract information from strings or to re-use parts of the pattern in the same regex (backreferences). There are two capturing groups: unnamed groups (…) and named groups (?<name>…). The named group was an addition to the unnamed groups that could only be backreference via their relative position in the regex. The named group allows us to give a group a more self-explanatory name referenced by that name. Any regex inside plain parentheses () is used for capturing. For instance, (abc) captures the sequence abc, and (?<year>\d{4}) captures four digits as a year.
    - There are five non-capturing groups. The first one is (?:…), which matches the group, but they do not save the text by the group. They are used when you need the grouping functionality without the overhead of capturing. Prefix the group with ?:, like (?:abc) to group it without capturing. It treats "abc" as a single unit without remembering the match.
      The second and third ones are the two lookahead groups (positive and negative lookahead). Positive Lookahead (?= ...) matches a group after the main expression without including it in the result. Negative Lookahead (?! ...) asserts that the group specified does not follow the main expression.
      The fourth and fifth are the two look-behind groups (positive and negative look-behind). Positive Look-behind (?<= ...) asserts that the group precedes the main expression and must be matched, but it does not consume any characters. Negative Lookbehind (?<! ...) asserts that the specified group does not precede the main expression.

Each component is crucial in constructing complex and efficient regular expressions for pattern matching and text processing tasks.

A tool for optimizing regular expressions

Optimizing regular expression

Regular expressions are great tools for pattern matching and text processing, but they can quickly become inefficient if not carefully crafted. Optimizing your regular expressions boosts performance and enhances their readability and maintainability. Here's a breakdown of some essential best practices and tips for optimizing your regular expressions. First, let's look at the syntax for Regex. From a top perspective, you can break it down into the following list.

Symbol	Naming
^	Anchor Start
$	Anchor end
[…]	Char class
\d,\D,\w,\W,\s,\S,\b,\B	Predefined class
Abc…	Literals
+,*,?,{…},+?,*?,??,{…}?	Quantifiers
(…),(?<name>…)	Capturing groups
(?:…),(?=…),(?!...),(?<=…),(?<!...)	Non-capturing groups
Cat|dog|bird	Alternation

Let's start by looking at the Quantifiers.

Optimizing Quantifiers

Quantifiers appear after a <pattern> where any quantifiers can be followed by an optional ? to indicate a lazy quantifier. E.g.

<pattern>+	meaning one or more occurrences of the pattern
<pattern>*	meaning zero or more occurrences of the pattern
<pattern>?	meaning zero of one occurrence of the pattern.

For clarity, a <pattern> here means a single pattern like: /d, [A-Z], b, etc. A quantifier can also be a bracket quantified. e.g. {n} or {n,} or {n,m}. A bracket quantifier can also be followed by an optional ? Indicating a lazy bracket.

<pattern>{n}	n occurrence of the <pattern>
<pattern> {n,}	n or more occurrences of the <pattern>
<pattern> {n,m}	from n to m occurrence of the <pattern>

Efficiency can often be achieved by simplifying how repetitions are handled:

```
Replace {1,}    → +   # to match one or more occurrences.
Replace {0,}    → *   # for zero or more occurrences.
Replace {0,1}   → ?   # to match zero or one occurrence.
```

```
Simplify {n,n}      → {n}
Simplify {n,n}?     → {n} # lazy quantifier ? is redundant
```

As I mentioned, a quantifier can be followed by the lazy quantifier symbol ? e.g.

<pattern>+?, <pattern>*?, <pattern>?? Or pattern>{…}?

We can also optimize the regex under certain conditions with the lazy quantifier.

If we encounter <pattern>+? where the lazy quantifier is applied, it is redundant to keep the +? because the pattern itself will be sufficient to capture the required behavior. We can, therefore, optimize it to:

<pattern>+? → <pattern>

For example: \d+? → \d

You could extend this optimization to other lazy quantifiers if needed, such as:

<pattern>*? → potentially equivalent to an empty pattern in some cases.
<pattern>?? → reduces to just the pattern itself if applicable.

However, these optimizations would need to be carefully applied depending on the context of the regex. Here is a classic example where it does not work. Consider the regex:

/<.*?>/

This regex is commonly used to match HTML tags like , , etc. The pattern .*? matches zero or more characters lazily, which matches as few characters as possible until the closing > is found. E.g. Hello . Matches , , ,

Here, the *? doesn't simply match an empty string; instead, it matches the shortest possible sequence of characters (between < and >) inside an HTML tag. If we did not put the lazy quantifier ? then it will match the entire string: which is not what you want to do.

There are some specific scenarios where a lazy quantifier follows the pattern *? (i.e., <pattern>*?) can be simplified by removing it. These cases occur when the pattern is redundant because of the surrounding context or because it doesn't contribute to the overall match. However, detecting these cases generally requires analyzing the structure of the regex and its context.

1. When the Pattern is at the End and Followed by an Anchor or Another Character That Enforces a Match. If the *? pattern is placed just before an anchor (^, $, \b) or a mandatory literal, and it’s non-contributory. The pattern can be removed. The lazy quantifier allows zero

pattern occurrences to be matched, and the anchor or literal will determine the match.
Example 1: a*?$ Since the *? allows zero matches and the $ enforces the end of the string. We can simplify this to just: $ In this case, the pattern a*? can be safely removed because the match will always end at the end of the string ($), whether or not there are any 'a' characters.
Example 2: \w*?\b Again, we can simplify; since \w*? can match zero characters, the word boundary will always be satisfied. This simplifies to just: \b The \w*? here is redundant because the word boundary will match even if there are no preceding word characters.

2. When the Pattern is part of a larger optional sequence. It can be removed if the pattern *? is part of an already optional larger sequence (e.g., due to an alternation or another quantifier). Example: (a|b*?). Simplification can be done to (a|) and simplified to a?. Here, b*? can be safely removed because it doesn't contribute anything different from the empty alternative in the alternation.
3. When the Pattern is Repeated Within an Already Optional Group If *? is applied within a group already repeated with a quantifier (like ? or {0,}), the *? can be redundant and may be removed.
   Example 1: (a*?)? The outer ? already marks the group optional, so *? Is redundant, and the regex simplifies to (a)? or just a?
   Example 2: (a*)*? Where the outer *? makes the whole group redundant because a* already allows zero matches. This simplifies to a*
4. In Alternation with an Empty Pattern. If *? appears in alternation (|) with an empty pattern or another pattern allowing zero matches, the *? can be removed.
   Example: (a*?|b), which can be simplified to just (a|b).

When simplifying patterns like <pattern>*? (i.e., zero or more occurrences of a pattern lazily), you can apply the optimization by removing the <pattern>*? if it's followed by certain anchors that make the lazy quantifier redundant. Anchors typically don't require any match themselves and dictate positions in the input (like the start or end of a string), so *? can be safely removed if the anchor sufficiently restricts the match. The following anchors can be used to simplify. ^, $, \b, \B, lookahead assertion: (?=…), Negative lookahead assertion: (?!...), Lookbehind assertion: (?<=…) and Negative lookbehind assertion (?<!...)

Leverage Character Classes

Predefined character classes can significantly simplify your expressions:

- \d can be used instead of [0-9], matching any digit.
- \w is a substitute for [a-zA-Z0-9_], matching any alphanumeric character plus an underscore.
- \s effectively matches any whitespace character, streamlining patterns and avoiding lengthy alternatives. \s substitutes [\t\n\t\f\v]
- A charset using multiple predefined character classes like [\d\w] can be simplified to [\w] since \d is a subset of \w.

Optimize OR Conditions (|)

When using alternation (|), placing the most likely conditions first can save time, especially if subsequent conditions are subsets of previous ones. Also, simplifying alternations by extracting common prefixes or suffixes can reduce complexity. For instance, we perform the following optimization of the OR conditions.

Transforming patterns like:

```
abc|abd              → ab[cd]
abc|ab               → ab[c]?
[0-9]|[a-z]|[A-Z]|[_] → \w
a|b|c                → [abc]
a|b|                 → [ab]?
```

A special optimization is when meeting a regular expression like:

```
/apple|applecare/
```

It looks innocent, but this regex will never work in real life. The issue is that every alternation takes place from left to right, and as soon as the regex engine has found a match, it moves to whatever follows the alternation.

If you try the above, you get this with a regex engine:

Input	Match
apple	apple
applecare	apple

Applecare is never a match because the first alternation takes precedence.
Since Applecare will never match, you could optimize it away.

```
apple|applecare      → apple
```

However, it is unlikely that this was the intention of the above regex. The intention was to match either apple or applecare; for this to happen, you must put the more extended version before the shorter version.

```
/applecare|apple/
```

If you try the above, you get the right solution:

Input	Match
apple	apple
applecare	applecare

Instead of removing the redundant alternation, the optimization rearranges it by moving the more extended version in front of the shorter version. However, the order will remain intact if there is no common subset. For example:

```
apple|applecare|grape|banana|bananacake  → applecare|apple|grape|bananacake|banana
```

Utilize Non-Capturing Groups

Non-capturing groups (?:...) are preferable when you do not need to save the captured group for later use. This approach reduces the overhead since the regex engine does not need to track the contents of these groups. Sometimes, a non-capturing grouping is unnecessary or doesn't change the behavior. For example, when a single pattern (a literal, char class, or predefined escapes) follows the non-capturing group, then the non-capturing can be removed. Also, if a non-capture group precedes by ^ (ANHOR_START) or $(ANCHOR_END), then the non-capturing group is redundant.

```
(?:a)              → a
(?:a|b|c)?         → [abc]?
^(?:abc)           → ^abc
(?:abc)$           → abc$
(?:abc)+           → (?:abc)+      #can't be optimized
(?:a)+             → a+
(?:[a-z])+         → [a-z]+
(?:world)|hello → world|hello
```

Implement Anchors Strategically

Using anchors like ^ (ANCHOR_START) for the start of a string and $ (ANCHOR_END) for the end of a string can prevent unnecessary matching attempts across the entire string when the pattern is known to be fixed or limited to specific positions.

Avoid Greedy Quantifiers

Greedy quantifiers (*, +) attempt to match as much text as possible, which can lead to performance issues. Opt for their lazy counterparts (*?, +?), which match the smallest possible string, enhancing efficiency. However, it depends on what you are trying to match.

Minimize Backtracking

To prevent excessive backtracking, which can slow down your regex, utilize atomic groups (?>...) and possessive quantifiers like *+ or ++. These constructs tell the regex engine to avoid revisiting previously matched characters, improving performance in complex patterns.

Since JavaScript does not support atomic groups or possessive quantifiers, you often need to rely on other methods to prevent inefficient backtracking:

1. **Non-Capturing Groups with Greedy Quantifiers**: You can sometimes rewrite parts of your regex using non-capturing groups (?:...) combined with greedy quantifiers, being careful about the overall pattern design to minimize backtracking.
2. **Lookahead Assertions**: Use lookahead assertions to check for conditions without actually consuming characters, which can sometimes help manage backtracking.
3. **Specific Character Exclusions**: Instead of using a . (dot), which matches any character (often leading to excessive backtracking), specify more explicitly which characters to match or not match. For example, use something like [^,]+ to match strings up to a comma without including it.

For example, if you're trying to match a sequence that ends with a specific suffix, instead of writing something like .*suffix, which can lead to excessive backtracking, you might specify what shouldn't be matched before the suffix, like [^]*suffix (assuming the suffix is preceded by spaces).

Use Lookaheads and Lookbehinds with Care

Lookaheads and look behinds are potent tools for assertions in regex, but they should be used sparingly. If overused, they can impact performance. Employ these when necessary to assert conditions without consuming characters.

Lookaheads and lookbehinds are advanced regex features that allow you to specify additional conditions for a match without including those in the match itself. These "zero-width assertions" do not consume any characters on the string; they just assert whether a match is possible.

Lookahead

Positive Lookahead (?=) asserts that what immediately follows the current position in the string must match the pattern specified inside the parentheses but does not include that in the match.

Example: X(?=Y) matches X only if X is followed by Y. In the string XYP, X would match because it's followed by Y.

Optimizing positive lookahead by removing redundant constructions.

(?=abc)abc	→ abc
(?:abc|def)(?=ghi)	→ (?:abc|def)

Negative Lookahead (?!) asserts that what immediately follows the current position in the string must not match the pattern specified inside the parentheses.

Example: X(?!Y) matches X only if X is not followed by Y. In the string XZP, X would match because Y does not follow it.

Negative lookaheads (?!...) can sometimes be rewritten more efficiently, especially when applied to simple character classes or patterns. E.g.

```
(?!a)b → [^a]b
```

This pattern matches b, but only if not preceded by a. Depending on the context, this can often be replaced with a negated character class for simplicity.

Lookbehind

Positive Lookbehind (?<=) asserts that what immediately precedes the current position in the string must match the pattern specified in the parentheses but not include that in the match.

Example: (?<=Y)X matches X only if Y precedes X. In the string YXP, X would match because Y precedes it.

Negative Lookbehind (?<!) asserts that what immediately precedes the current position in the string must not match the pattern specified inside the parentheses.

Example: (?<!Y)X matches X only if X is not preceded by Y. In the string ZXP, X would match because it's not preceded by Y.

Practical Example of lookbehind

Consider a scenario where you need to find dollar amounts not preceded by the word Refund. The regex could use a negative lookbehind to ensure the amounts we match aren't part of a refund: /(?<!Refund)\$[0-9]+/ with the target string “Refund $100, Earned $200" will match $200 because it is not preceded by "Refund ".

Lookaheads and lookbehinds are incredibly powerful for conditions where you need to check the context of a substring without including that context in the final matched result. They are often used in data validation, parsing complex string formats, and conditional processing within lines of text.

It can be challenging to implement any optimization of regular expressions. A few can be done without knowing the purpose of the regular expression. However, most are based on how you want to extract information from the regular expression. For example, changing a capturing grouping to a non-capturing requires that you don’t plan to use the grouping later on. It will be impossible for an optimizer to know that; therefore, it can only be suggested that any grouping you don’t plan to refer back to can be changed to a non-capturing group.

The Regex optimizing tool

Now, with all the optimization potential in place, we can turn our attention to the actual coding for the tool. At first, I was planning to analyze the regular expression to find a pattern that can convert to a simpler version or is redundant. However, I quickly ran into limitations using that approach. Instead, I realized that to get through, I needed to build an abstract syntax tree (AST) for a regular expression and then use compiler techniques to rearrange, simplify, and remove redundant elements in the regular expression.

An Abstract Syntax Tree (AST) represents the syntactic structure of a source code or other formal language, such as a regular or mathematical expression. The AST abstracts away the actual syntax used (like parentheses, brackets, or commas), focusing instead on the structural and logical elements of the code. Each node in the tree represents a construct occurring in the code.

Key Characteristics of a General AST for Programming Language are nodes, edges, abstraction, and hierarchy:

1. Each node represents a syntactic construct, such as an operator, literal, function, or code block. Nodes are often categorized by their "type" (e.g., EXPRESSION, STATEMENT, FUNCTION_CALL, etc.).
2. The edges represent relationships between nodes. For example, a function call might be a parent node with child nodes representing the function name and arguments.
3. An AST simplifies the source code, abstracting away details like punctuation or specific symbols that are not necessary to understand the code's logic.
4. Hierarchy reflects the program's hierarchical structure, showing which constructs are "inside" or "related to" others.

Example:

Consider a simple arithmetic expression found in countless books about compilers:

```
2 + 3 * 4
```

The AST for this expression would look something like this:

```
  +
 / \
2   *
   / \
  3   4
```

- The root node is the + operator because addition is the main operation.
- The * operator is a child of the +, showing that multiplication happens before addition due to precedence rules.
- The numbers 2, 3, and 4 are leaf nodes (the operands of the operations).

Why Use an AST?

1. Compilers and interpreters use ASTs to analyze and manipulate code, such as for syntax checking, optimization, or translation into machine code.
2. During compilation, an AST analyzes the structure and dependencies between operations to identify areas where the code can be optimized.
3. Once an AST is constructed, a compiler can easily traverse the tree to generate the equivalent machine code or bytecode.

4. Many code analysis tools (like linters or formatters) work by manipulating ASTs because it's easier to analyze a structured tree than raw source code.

The purists would properly argue that for my simple regex project, it is more like a parse tree than an AST?

An AST is often confused with a parse tree (or concrete syntax tree), but they are different:

- A parse tree contains all the details of the grammar and syntax of the source code, including all tokens, parentheses, and punctuation.
- An AST abstracts unnecessary grammar details and focuses on the semantic structure.

We're working with a parse tree if your structure retains all the details (like distinguishing + from +?. It would be more accurately called an AST if it simplifies or optimizes such cases. So, we have a parse tree that we optimized into an AST. For simplicity, I will call it an AST.

For our regular expression project, we have Nodes like (only a subset shown below):

ANCHOR_START	^
ANCHOR_END	$
ALTERNATION	|
CAPTURING GROUP	(…), (?<name> …)
NON-CAPTURING Group	(?: …), and others
LITERAL.	A…Z , a…z, 0… 9
PREDEFINED.	\d,\w,\D,\W,\s,\S,\b,\B
QUANTIFIER.	+,*,?, {…}

For our Project, we need four different functions. (JavaScript code in Appendix)

1. Convert a regular expression into an AST. ParseRegextoAST()

2. Convert it back to a regular expression. ASTtoRegex()

3. Optimized the regular expression. OptimizeAST()

4. Decoded the AST for debugging purposes. ASTtoString()

All four different functions include recursive functions to descend into the tree and optimize it.

Two-Phase Optimization of JavaScript Regular Expressions

Now, we have most of the stuff ready for our optimizer. I divided the optimization into two steps to have a structured approach. The first step is a simple node optimizer, in which we only look at the node itself and try to optimize it. A couple of nodes are relevant for this optimization. This is the:

- QUANTIFIER node
- CHAR_CLASS node.
- Empty NON-CAPTURING nodes.
- Empty Internal CAPTURING nodes

We traverse the AST tree and try to optimize the node directly. We have previously described the simple replacement of quantifiers and char classes to simplify the node. We look for redundant empty groups for the non-capturing and capturing groups that can be removed. For example, an empty non-capturing group has no children, which is easy to detect in the first step.

The second step is global optimization, where we look across several nodes to find optimizations we can perform. As mentioned, some optimizations rely on what comes before and or after a particular node. Again, we traverse the AST tree with a bottom-up approach to identify and perform global optimization.

Overview of the Two-Phase Optimization Approach

The optimization strategy is divided into two distinct stages:

1. **Node-Based Optimization**: This phase focuses on optimizing individual nodes of the AST.
2. **Global Optimization**: The second phase looks at patterns across groups of nodes, applying optimizations that can only be achieved when considering the broader context.

Each stage is designed to address specific types of inefficiencies and redundancies, outlined in more detail below.

Node-Based Optimization (First Phase)

Node-based optimization is the initial step in the optimization process. This phase involves processing each node in the AST independently to apply local optimizations. The AST represents the regular expression structure, where each node corresponds to a particular component, such as literals, quantifiers, or groups.

Literal Node Combination

One of the first optimizations performed is combining consecutive literal nodes. In many regular expressions, adjacent literals can be grouped to reduce the overall complexity of the regex.

For example, consider the following regex:

```
Abcd  (as four different internal literal nodes)
```

This can be simplified by combining the individual literal nodes:

```
Abcd      (as one internal literal node)
```

Instead of having four separate nodes representing each literal character in the AST, the optimizer merges them into a single node. This reduces the AST's complexity and the regex pattern's size.

Quantifier Simplification

Quantifiers in regular expressions indicate how often a character or group should be matched. Some quantifiers can be simplified for better readability and performance. The node-based optimization phase identifies and simplifies such quantifiers.

Examples of Quantifier Simplification:

- {0,1} is simplified to ?, indicating that the preceding element is optional.
- {1,} is simplified to +, indicating that the preceding element must appear at least once.
- {0,} is simplified to *, meaning that the element can appear any number of times, including zero.

Consider the regex:

```
a{1,} -> a+
```

This transformation is applied directly at the node level by examining the quantifier node and replacing it with a simpler equivalent.

Character Class Optimization

Character classes define sets of characters that can be matched. In many cases, character classes can be simplified by replacing them with predefined character classes such as \d for digits, \w for word characters, and \s for whitespace.

For instance, the following character class:

```
[0-9]-> \d
```

Replacing the longer form with the predefined shorthand makes the regex easier to read and more efficient.

Removal of Redundant Groups

In regular expressions, groups (both capturing and non-capturing) are often used to control how parts of the pattern are matched. However, not all groups are necessary. The optimizer removes redundant non-capturing groups that do not affect the logic of the regex.

For example, a regex with an unnecessary non-capturing group:

```
(?:abc)-> abc
```

This optimization identifies and eliminates such groups, reducing the AST's complexity and the regex itself.

Removal of Empty Nodes

Empty nodes can be introduced during parsing or writing regexes, particularly when dealing with quantifiers or groups. The optimizer removes these empty nodes to streamline the AST. For example, a group or quantifier node with no children is removed, as it does not contribute to the overall pattern.

Global Optimization (Second Phase)

Global optimization occurs after node-based optimization has been completed. In this phase, the optimizer examines the regular expression's broader structure and applies optimizations that span multiple nodes.

Redundant Anchor Removal

Anchors such as ^ (beginning of string) and $ (end of string) match specific positions in the input. However, it is possible to have redundant anchors that do not add value to the pattern, especially when there are multiple anchors.

For example:

```
^^abc$ -> ^abc$
```

By identifying and removing redundant anchors, the optimizer ensures the regex is as efficient as possible.

Alternation Simplification

Alternation allows a regex to match one pattern out of several options, denoted by the pipe | symbol. Sometimes, alternations can be simplified by combining them into character classes or factoring out common prefixes or suffixes.

Example: Combining Alternations into a Character Class

Consider the following regex:

```
a|b|c -> [abc]
```

This alternation can be simplified into a character class:

This reduces the complexity of the regex and makes it more efficient.

Example: Factoring Out Common Prefixes

For a regex such as:

```
cat|car -> ca(t|r)
```

where the common prefix ca can be factored out.

This optimization reduces the number of operations needed to match the pattern, leading to performance improvements.

Removal of Redundant Non-Capturing Groups

Non-capturing groups, denoted by (?:...), are often used to group parts of a regex without storing the matched value. While they are helpful in some instances, they can also be redundant if their removal does not affect the behavior of the regex.

For example:

```
(?:a|b|c) -> a|b|c -> [a-c]
```

The optimizer scans for such groups and removes them when they do not alter the meaning of the regex.

Global Quantifier Adjustments

In the node-based optimization phase, quantifiers were simplified on a per-node basis. However, in the global optimization phase, **quantifiers are further adjusted** based on the context of surrounding nodes. For example, a lazy quantifier (e.g., `*?`) may be replaced with its greedy counterpart (`*`) if the surrounding pattern allows it without altering the matching behavior.

This adjustment improves performance by reducing the need for backtracking, typically introduced by lazy quantifiers.

Lookahead and Lookbehind Optimization

Lookahead (`(?=...)`) and lookbehind (`(?<=...)`) assertions allow a regex to match patterns based on what comes before or after a particular position. While these assertions are powerful, they can also introduce inefficiencies if not used carefully.

During global optimization, the optimizer identifies redundant lookahead or look-behind assertions and removes them. Additionally, consecutive assertions are combined when possible, reducing the regex's overall complexity.

The two-phase optimization process—combining node-based and global optimizations—ensures that regular expressions are simplified locally and globally. Node-based optimization reduces

complexity within individual nodes, such as merging literals, simplifying quantifiers, and removing unnecessary groups. Global optimization, on the other hand, looks at the broader context of the regex to apply more comprehensive changes, including the removal of redundant anchors, alternation simplification, and global quantifier adjustments.

By applying these optimizations, regular expressions become more efficient in terms of runtime performance, more maintainable, and more accessible for developers to understand.

This two-phase approach provides a powerful method for improving the efficiency of regular expressions, leading to faster execution and reduced complexity in pattern matching.

Example of optimization

Using the regular expression /[a-zA-Z_][0-9a-zA-Z_]*/g
you can optimize it to /[A-Z_a-z]\w*/

```
The details of the optimization are listed below.
Original 1st Regex:
/[a-zA-Z_][0-9a-zA-Z_]*/g
Prior Optimization:
1: ROOT: Child[2]
2:    CHAR_CLASS: Value=[a-zA-Z_], Parent=1
3:    QUANTIFIER: Value=*: Child[1], Parent=1
4:       CHAR_CLASS: Value=[0-9a-zA-Z_], Parent=3

Suggested optimization:
[a-zA-Z_] to [A-Z_a-z] to simplify regex
Replaced [0-9a-zA-Z_] with \w to simplify regex
New Optimize regular expression:
/[A-Z_a-z]\w*/
After Optimization
1: ROOT: Child[2]
2:    CHAR_CLASS: Value=[A-Z_a-z], Parent=1
3:    QUANTIFIER: Value=*: Child[1], Parent=1
4:       PREDEFINED_CLASS: Value=\w, Parent=3
```

Who is the optimizer tool for?

In optimizing regular expressions, it's important to recognize the different needs of users at various skill levels. An optimizer provides significant advantages for beginners and intermediate users by automating best practices and avoiding common inefficiencies. Novices often write patterns prone to excessive backtracking or redundancies, while intermediate users may overlook subtle optimizations like simplifying quantifiers or removing unnecessary capturing groups. An optimizer helps bridge this gap, ensuring performance improvements, reducing errors, and

serving as a learning tool by demonstrating more efficient patterns. In contrast, advanced users are typically as proficient as an optimizer in crafting highly optimized regex. Their deeper understanding of regex engines allows them to make fine-tuned optimizations based on specific context, balancing performance with readability and maintainability. While advanced users may not always require automated optimizations, they can still benefit from an optimizer's consistency and convenience, particularly for catching repetitive or easily overlooked improvements.

Further Improvement?

There is only so much you can do for optimization when you don't know the programmer's intent and use of the regex. For example, the optimizer can see when a grouping is necessary. Therefore, we need to finalize the optimizer with some general guides for the programmer.

1. Backtracking Control and Avoidance

- **Backtracking** is a common source of inefficiency in regex, mainly when patterns involve ambiguous quantifiers like `*` or `+`. Patterns that lead to excessive backtracking can cause performance degradation.
- **Optimization Techniques**:
  - **Greedy vs. Lazy Quantifiers**: Although lazy quantifiers (`*?`, `+?`) reduce backtracking, they may not always be necessary. Optimizers can convert lazy quantifiers to greedy ones where possible.
  - **Atomic Groups**: Introducing atomic groups `(?>...)` can help prevent unnecessary backtracking by preventing the regex engine from retrying previous matches when subsequent patterns fail.
- **Example**:
  - A regex pattern like `(a+)+` can cause excessive backtracking in certain inputs. Atomic groups can mitigate this:
    - `(?>a+)+` would avoid excessive backtracking by preventing the regex engine from reconsidering partial matches.

2. Anchored vs. Unanchored Expressions

- **Anchored Regex**: Using anchors like `^` and `$` can significantly improve regex performance by restricting the possible positions where matches are attempted.
- **Optimization**: If a regex is known to always match at the start or end of a string, the optimizer can introduce anchors if they are not already present.
- **Example**:
  - A regex like `abc` can be optimized to `^abc` if it is always expected at the start of a string.

3. Optimizing Lookahead and Lookbehind

- While we have covered the removal of redundant lookaheads and lookbehinds, We can suggest **optimizing complex lookaheads** by merging multiple lookaheads into a single assertion or simplifying them where possible.
- **Example**:
  - A regex like `(?=a)(?=b)` can be optimized by merging the two lookaheads into a single assertion when they apply to the same position in the string.

4. Greedy vs. Lazy Quantifier Balancing

- While we have covered quantifier simplifications, it is worth noting the importance of balancing **greedy** and **lazy** quantifiers, particularly in cases where the optimizer can predict patterns that could lead to catastrophic backtracking with excessive greediness or laziness.
- **Example**:
  - The optimizer could choose `.*` instead of `.*?` when it detects that a lazy quantifier may add unnecessary complexity without significant benefit in matching.

5. Unrolling the Loop

- This technique involves replacing **loops** or repetitive patterns with **explicit enumerations** to avoid backtracking and improve performance.
- **Example**:
  - A regex like `(abc)+` could be rewritten to `abcabcabc` when the repetition is known to be limited.

6. Lazy Matching Prevention

- **Lazy matching** is sometimes misused, leading to inefficient patterns. The optimizer can replace lazy matching with greedy ones if the surrounding context makes it safe without affecting the correctness.
- **Example**:
  - In a pattern like `.*?abc`, if `abc` is expected to occur frequently, lazy matching could be replaced with greedy matching for better performance.

7. Inline Comments and Group Names for Readability

- Though not directly related to performance, we can suggest or apply inline comments or **named groups** to improve the **readability** and **maintainability** of complex regular expressions.
- **Example**:
  - Transforming `(a(b(c)))` into `(?<outer>a(?<middle>b(?<inner>c)))` would improve readability without affecting performance.

8. Optimizing for Specific Regex Engines

- Depending on the **regex engine** (e.g., JavaScript's `RegExp` vs. Python's `re`), different optimizations may be more beneficial.
- **Example**:
  - Some engines do not support lookbehind assertions. Alternatives that are outside the scope of this optimizer need to be considered.

9. Regex Compilation and Caching

- Compiling and caching regex patterns for performance-sensitive applications can significantly improve runtime performance, primarily when the same pattern is used repeatedly.
- **Optimization**: If you can compile the regex patterns ahead of time and reuse rather than recompile them with each execution,

10. Nested Group Optimizations

- Regex patterns often use nested groups, which can sometimes be flattened or merged if they don't contribute to backtracking or capture results.
- **Example**:
  - In a pattern like `((abc))`, the outer and inner groups can sometimes be merged into a single capturing group if there's no difference in their role.

Conclusion

The four tools listed are a crucial part of optimizing a regular expression through decomposition, and optimization not only aids in debugging and development but also enhances the regular expression to be the most optimal version and improves one's ability to write more efficiently and error-free regexes. The Optimizing tool set plays an important role in this educational process. The Author's regular expression tool is the sixth tool mentioned. It has a unique feature that allows you to test two regular expressions simultaneously, decompose a regular expression, or optimize a regular expression. This becomes useful when trying to build a regular expression for a given problem, and then dynamically, the expression can be built to match more of the needed pattern successively.

Other Useful online tools

There are several online tools for testing and debugging regular expression. Here are some widely-used regex tools and their relevant details that you can reference in your article:

1. Regex101

- **Website**: Regex101
- **Description**: Regex101 is a powerful online tool for testing and debugging regular expressions. It supports multiple programming languages, including JavaScript, Python,

PHP, and Go. The tool provides a detailed explanation of each regex part as you type, along with a quick reference guide and a library of user-submitted regex patterns.

- **Key Features**:
  - Real-time regex parsing and testing.
  - Detailed explanations of regex constructions.
  - Code generator for various languages.
  - User pattern library.

2. RegExr

- **Website**: RegExr
- **Description**: RegExr is another popular online tool for learning, building, and testing regular expressions. It offers a clean interface and provides real-time visual feedback on your regex pattern matching.
- **Key Features**:
  - Real-time results and highlighting.
  - Extensive community patterns and examples.
  - Detailed help and cheat sheets.
  - Provide a history of your regex tests for easy backtracking.

3. RegexBuddy

- **Website**: RegexBuddy
- **Description**: RegexBuddy is a downloadable tool for Windows that acts as your regex assistant. It helps you create and understand complex regexes and implements them in source code.
- **Key Features**:
  - Detailed analysis of regular expressions.
  - Test regexes against sample texts.
  - Integration with various programming environments.
  - Regex building blocks for easier assembly.

4. Regex Pal

- **Website**: Regex Pal
- **Description**: Regex Pal is a straightforward, web-based tool for quickly testing JavaScript regular expressions. It provides immediate visual feedback but is more straightforward and less feature-rich than Regex101 or RegExr.
- **Key Features**:
  - Quick testing with real-time highlighting.
  - Minimalistic and fast.
  - Sidebar with regex tokens and short descriptions.

5. Regex Tester

- **Website**: Regex Tester and Debugger Online - Javascript, PCRE, PHP

- **Description**: Regex Tester from RegexPlanet supports testing and debugging regex for multiple programming languages, including Java, .NET, and Ruby. It offers a unique environment to test regex in different programming contexts.
- **Key Features**:
    - Support for several programming languages.
    - Regex library and community examples.
    - Advanced options for regex testing and results.

6. Regular expression tester

- **Website**: Testing Regular expression for matching specific text patterns (hvks.com)
- **Description**: Regular Expression Tester supports testing and debugging regex for JavaScript programming languages. It offers a unique environment to test regex in different programming contexts.
- **Key Features**:
    - Support for most programming languages.
    - Samples of regular expression for most common day coding problems
    - Offer decomposing of regular expressions for easier debugging and understanding.
    - Offer optimization of regular expressions
    - Offer execution analysis
    - Offer multiline matching.
    - Can check and test two regular expressions simultaneously.
    - Offers printing and emailing of results.

Reference

1. J. Goyvaerts & S. Levithan, Regular Expression Cookbook, O'Reilly May 2009
2. Regular Expression Tester. Testing Regular expression for matching specific text patterns (hvks.com)
3. Decoding regular expression. Decomposing regular expression (hvks.com)
4. Jeffrey E.F. Friedl, Mastering Regular Expression 3rd edition. August 2006, O'Reilly Media, Inc.

Appendix

All code fragments in this appendix are based on JavaScript. Go to my website, hvks.com, to download the latest version. [Testing Regular expression for matching specific text pattern]

A JavaScript class handler binds the 4 JavaScript functions together in a simple class structure.

```
// A handler for the RegexAST
class RegexASTHandler {
    constructor(regex) {
        this.regex = regex; // Original regex string
        this.ast = parseRegextoAST(regex); // Parse regex into AST
    }

    toString() {
        return ASTtoString(this.ast); // Convert AST to string
    }

    toRegex() {
        return ASTtoRegex(this.ast); // Convert AST to RegExp object
    }

    optimize() {
        this.ast = OptimizeAST(this.ast); // Optimize the AST
    }

    createIterator(input, modifiers) {
        return createRegexAstIterator(this.ast, input, modifiers); // Create an iterator
    }
}
```

Function ASTtoString()

```
// Helper function to convert an AST node to a string representation
function ASTtoString(rootNode) {
    if (!rootNode) return 'Invalid AST';

    function ASTtoStringInternal(node, indent = 0, line = 1, parentLine = null) {
        const indentation = '   '.repeat(indent);
        let result = [];

        // Add line number, node type, and parent information to the result
        result.push(`${line}: ${indentation}${node.type}`);
        if (node?.value) result.push(`: Value=${node.value}`);
        const childCount = node.children.length;
        if (childCount > 0) result.push(`: Child[${childCount}]`);
        if (parentLine !== null) result.push(`, Parent=${parentLine}`);
        if('seq' in node && node.seq!=null)
            result.push(`, Seq=${node.seq}`);  // Debug
        result.push('\n');

        let currentLine = line + 1;

        // Process each child recursively
        for (const child of node.children) {
```

```
            const childResult = ASTtoStringInternal(child, indent + 1, currentLine,
line);
            result.push(childResult.result);
            currentLine = childResult.line;
        }

        return { result: result.join(''), line: currentLine };
    }

    return ASTtoStringInternal(rootNode).result;
}
```

Function ASTtoRegex()

```
// Convert AST to a regex string
// Notice we don't need the parent link
function ASTtoRegex(node) {
    let regex = '';

    switch (node.type) {
        case 'ROOT':
        case 'GROUP':
        case 'NON_CAPTURING':
        case 'CAPTURING':
        case 'NAMED_CAPTURING':
        case 'LOOKAHEAD':
        case 'NEGATIVE_LOOKAHEAD':
        case 'LOOKBEHIND':
        case 'NEGATIVE_LOOKBEHIND':
            // Handle different types of groupings with specific regex syntax
            const groupPrefixes = {
                'NON_CAPTURING': '(?:',
                'NAMED_CAPTURING': `(?<${node.value}>`,
                'CAPTURING': '(',
                'LOOKAHEAD': '(?=',
                'NEGATIVE_LOOKAHEAD': '(?!',
                'LOOKBEHIND': '(?<=',
                'NEGATIVE_LOOKBEHIND': '(?<!'
            }
            if (groupPrefixes[node.type])
                regex += groupPrefixes[node.type]

            // Process each child
            node.children.forEach((child) =>
                regex += ASTtoRegex(child)
            )

            // Close the group if necessary
            if (['NON_CAPTURING', 'CAPTURING', 'NAMED_CAPTURING','LOOKAHEAD',
'NEGATIVE_LOOKAHEAD', 'LOOKBEHIND', 'NEGATIVE_LOOKBEHIND'].includes(node.type))
                regex += ')'
            break
        case 'ALTERNATION':
            // If alternation is inside a group, enclose it in parentheses
            const needsGrouping = node.parent && node.parent.type == 'GROUP'
            if (needsGrouping) regex += '('

            node.children.forEach((child, childIndex) => {
                if (childIndex > 0)
                    regex += '|'
```

```
                regex += ASTtoRegex(child)
            })

            if (needsGrouping) regex += ')'
            break
        case 'QUANTIFIER':
            // Assume the quantifier applies to the last segment of the regex
            if (node.children.length > 0)
                regex += ASTtoRegex(node.children[0])
            regex += node.value
            break
        case 'CHAR_CLASS':
        case 'PREDEFINED_CLASS':
        case 'LITERAL':
        case 'ANCHOR_START':
        case 'ANCHOR_END':
             regex += node.value;
            break
        case 'ESCAPED_CHAR':
            regex += '\\' + node.value;
            break
        default:
            throw new Error(`Unknown node type: ${node.type} (Node value:
${node.value})`)
    }

    return regex
}
```

Function parseRegex2AST()

```
// Here is the Regex node for parsing and optimizing regex
// type is 'ANCHOR_START', 'ANCHOR_BEGIN', 'PREDEFINED_CLASS', 'ESCAPED_CHAR',
//          'CHAR_CLASS', 'CAPTURING', 'NON_CAPTURING', 'NAMED_CAPTURING',
//          'LOOKAHEAD', 'NEGATIVE_LOOKAHAED', 'LOOKBEHIND", 'NEGATIVE_LOOKBEHIND',
//          'ALTERNATION', 'GROUP', 'LITERAL', 'QUANTIFIER', 'ROOT'
// value is the value or undefined or null. fx. QUANTIFIERS can have the vlaue
//          of '*','+','?','*?','+?','??'
//          for CAPTURING it is the grouping number
//          for LITERAL it is the literal itself (character)
//          for ANCHOR_START it is '^', ANCHOR_END it is '$'
//          for PREDEFINED+CLASS it is one of theses: 'b', 'B', 'd', 'D', 'w', 'W', 's',
'S'
//          for the other captures it is the string version e,g, '(?:' ...
//
class RegexNode {
    constructor(type, value, parent=null) {
        this.type = type;       // Node type
        this.value = value;     // Node value
        this.parent= parent;    // Parent link
        this.children = [];     // Siblings
        this.seq=null;          // For debug purpose
    }
    addChild(node) {
        node.parent = this;     // Set the parent link
        this.children.push(node);
    }
}

// Parse a Regex into an AST.
```

```
// The regex can be the regex itself or the string representation of it.
function parseRegextoAST(regex) {
    const regexString = regex instanceof RegExp ? regex.source : regex

    function parseRegex2AST(regexString, capturingGroupCount = { count: 0 }, parent =
null) {
        const rootNode = new RegexNode('ROOT', null, parent) // Set parent as null for
root
        let i = 0

        // Handling quantifiers and laziness
        function handleQuantifier(rootNode, regex, currentIndex, quantifier) {
            let isLazy = (currentIndex + quantifier.length < regex.length &&
regex[currentIndex + quantifier.length] === '?')
            let quantifierValue = quantifier + (isLazy ? '?' : '')
            const lastNode = rootNode.children.pop()
            const quantifierNode = new RegexNode('QUANTIFIER', quantifierValue,
rootNode)
            quantifierNode.addChild(lastNode)
            rootNode.addChild(quantifierNode)
        }

        // Parse group types (e.g., capturing, named capturing, non-capturing)
        function handleGroup(type, regex, startIndex, name = '') {
            let depth = 1
            let i = startIndex
            let groupNode = new RegexNode(type, name, rootNode)
            while (i < regex.length && depth > 0) {
                if (regex[i] === '(' && !(regex[i - 1] === '\\')) depth++
                if (regex[i] === ')' && !(regex[i - 1] === '\\')) depth--
                i++
            }
            let content = regex.substring(startIndex, i - 1)
            let parsedContent = parseRegex2AST(content, capturingGroupCount, groupNode)
            if (parsedContent.type === 'ROOT') parsedContent.type = 'GROUP'

            if (parsedContent.type === 'GROUP') {
                if (parsedContent.children.length === 1)
                    groupNode.addChild(parsedContent.children[0])
                else if (parsedContent.children.length != 0)
                    groupNode.addChild(parsedContent)
            } else
                groupNode.addChild(parsedContent)
            return { node: groupNode, newPosition: i }
        }

        // Main paarsing loop through each regex character
        while (i < regexString.length) {
            const char = regexString[i]

            switch (char) {
                case '^':
                    rootNode.addChild(new RegexNode('ANCHOR_START', '^', rootNode))
                    break
                case '$':
                    rootNode.addChild(new RegexNode('ANCHOR_END', '$', rootNode))
                    break
                case '\\':
                    i++
                    if (['b', 'B', 'd', 'D', 'w', 'W', 's',
'S'].includes(regexString[i]))
```

```
                        rootNode.addChild(new RegexNode('PREDEFINED_CLASS', '\\' +
regexString[i], rootNode))
                    else
                        rootNode.addChild(new RegexNode('ESCAPED_CHAR', regexString[i],
rootNode))
                    break
                case '[':
                    let endIdx = regexString.indexOf(']', i + 1)
                    rootNode.addChild(new RegexNode('CHAR_CLASS',
regexString.substring(i, endIdx + 1), rootNode))
                    i = endIdx
                    break
                case '(':
                    let result
                    if (regexString.substring(i, i + 3) === '(?:') {
                        i += 3
                        result = handleGroup('NON_CAPTURING', regexString, i)
                    } else if (regexString.substring(i, i + 3) === '(?<' &&
regexString.indexOf('>', i) !== -1) {
                        let nameEndIdx = regexString.indexOf('>', i)
                        let name = regexString.substring(i + 3, nameEndIdx)
                        i = nameEndIdx + 1
                        result = handleGroup('NAMED_CAPTURING', regexString, i, name)
                    } else if (regexString.substring(i, i + 3) === '(?=' ||
regexString.substring(i, i + 3) === '(?!') {
                        let type = regexString.substring(i + 2, i + 3) === '=' ?
'LOOKAHEAD' : 'NEGATIVE_LOOKAHEAD'
                        i += 3
                        result = handleGroup(type, regexString, i)
                    } else if (regexString.substring(i, i + 4) === '(?<=' ||
regexString.substring(i, i + 4) === '(?<!') {
                        let type = regexString.substring(i + 3, i + 4) === '=' ?
'LOOKBEHIND' : 'NEGATIVE_LOOKBEHIND'
                        i += 4
                        result = handleGroup(type, regexString, i)
                    } else {
                        i++
                        let currentCaptureCount = ++capturingGroupCount.count
                        result = handleGroup('CAPTURING', regexString, i)
                        result.node.value = currentCaptureCount
                    }
                    rootNode.addChild(result.node)
                    i = result.newPosition - 1
                    break
                case '|':
                    let endIndex
                    if (rootNode.type !== 'ALTERNATION') {
                        const altNode = new RegexNode('ALTERNATION', null, rootNode)
                        const leftSideGroup = new RegexNode('GROUP', null, rootNode)
                        leftSideGroup.children = [...rootNode.children]
                        rootNode.children = []
                        altNode.addChild(leftSideGroup)
                        rootNode.addChild(altNode)
                        endIndex = regexString.length
                        const alternationContent = regexString.substring(i + 1)
                        const rightSideAST = parseRegex2AST(alternationContent,
capturingGroupCount, altNode)

                        if (rightSideAST.children[0] && rightSideAST.children[0].type
=== 'ALTERNATION')
```

```
                            rightSideAST.children[0].children.forEach(child =>
altNode.addChild(child))
                        else {
                            const rightSideGroup = new RegexNode('GROUP', null, altNode)
                            rightSideGroup.children = [...rightSideAST.children]
                            altNode.addChild(rightSideGroup)
                        }
                    } else {
                        endIndex = regexString.length
                        const alternationContent = regexString.substring(i + 1)
                        const additionalAST = parseRegex2AST(alternationContent,
capturingGroupCount, rootNode)

                        if (additionalAST.children[0] && additionalAST.children[0].type
=== 'ALTERNATION')
                            additionalAST.children[0].children.forEach(child =>
rootNode.addChild(child))
                        else {
                            const additionalGroup = new RegexNode('GROUP', null,
rootNode)
                            additionalGroup.children = [...additionalAST.children]
                            rootNode.addChild(additionalGroup)
                        }
                    }
                    i = endIndex - 1
                    break
                case '*':
                case '+':
                case '?':
                    handleQuantifier(rootNode, regexString, i, char)
                    if (regexString[i + 1] === '?')
                        i++
                    break
                case '{':
                    let closeBraceIndex = regexString.indexOf('}', i)
                    if (closeBraceIndex !== -1) {
                        handleQuantifier(rootNode, regexString, i,
regexString.substring(i, closeBraceIndex + 1))
                        if (regexString[closeBraceIndex + 1] === '?')
                            ++closeBraceIndex
                        i = closeBraceIndex
                    }
                    break
                default:
                    rootNode.addChild(new RegexNode('LITERAL', char, rootNode))
                    break
            }
            i++
        }
        return rootNode
    }

    function combineLiteralNodes(astNode) {
        for (let i = 0; i < astNode.children.length; i++) {
            let node = astNode.children[i];
            if (node.type === 'LITERAL') {
                while (i + 1 < astNode.children.length && astNode.children[i + 1].type
=== 'LITERAL') {
                    node.value += astNode.children[i + 1].value;
                    astNode.children.splice(i + 1, 1);
                }
```

```
            }
            if (node.children && node.children.length > 0)
                combineLiteralNodes(node);
        }
    }

    function removeUnnecessaryGroups(astNode) {
        for (let i = 0; i < astNode.children.length; i++) {
            let node = astNode.children[i];
            if (node.type === 'GROUP' && node.value == null && node.children.length ===
1) {
                let child = node.children[0];
                astNode.children[i] = child;
                child.parent = astNode;
            }
            if (node.children && node.children.length > 0)
                removeUnnecessaryGroups(node);
        }
    }

    let seq=1;
    function sequence(astNode) {
        astNode.seq=seq;
        for (let i = 0; i < astNode.children.length; ++i) {
            let node = astNode.children[i];
            node.seq=++seq;
            if (node.children && node.children.length > 0)
                sequence(node);
        }
    }

    let ast = parseRegex2AST(regexString)
    combineLiteralNodes(ast);
    removeUnnecessaryGroups(ast);
    //sequence(ast);  // Debug purpose. Can be removed
    return ast
}
```

Function optimizeAST()

```
// Optimize AST
function optimizeAST(node, changes = new Set()) {
    let changeDesc = '';

    // Find the next sibling or return null if not found or end of siblings
    function nextSibling(currentNode) {
        if (!currentNode.parent)
            return null; // No parent means no siblings
        const siblings = currentNode.parent.children; // Get all siblings (children of
the parent)
        const index = siblings.indexOf(currentNode);  // Find the index of the current
node
        if (index >= 0 && index < siblings.length - 1)
            return siblings[index + 1];  // Return the next sibling if it exists
        return null; // No next sibling (end of siblings or node not found)
    }

    // Utility function to find the previous sibling of the current node
    function previousSibling(node) {
        if (!node.parent)
```

```
            return null; // No parent means no siblings
        const siblings = node.parent.children; // Get all siblings from the parent
        const index = siblings.indexOf(node);  // Find the index of the current node
        if (index > 0)
            return siblings[index - 1];  // Return the previous sibling if it exists
        return null; // No previous sibling if the node is the first child
    }

    // Compare two nodes structurer
    function CompareNodes(node1, node2) {
        // Step 1: Check if both nodes are null (both are identical in this case)
        if (!node1 && !node2) return true;

        // Step 2: If one is null and the other is not, they are not similar
        if (!node1 || !node2) return false;

        // Step 3: Compare the current nodes (type and value)
        if (node1.type !== node2.type || node1.value !== node2.value)
            return false;

        // Step 4: Compare the number of children
        if (node1.children.length !== node2.children.length)
            return false;

        // Step 5: Recursively compare all children
        for (let i = 0; i < node1.children.length; i++) {
            if (!CompareNodes(node1.children[i], node2.children[i]))
                return false;  // If any child comparison fails, the entire structure is
different
        }
        // Step 6: If all checks passed, the nodes and their children are similar
        return true;
    }

    function normalizeCharacterClass(value) {

        function preprocessCharacterClass(value) {
        // [] has been strip before calling
        // Replace shorthand character classes with their expanded forms or remove if
redundant
        const hasWordChar = /\\w/.test(content);
        const hasNonWhitespace = /\\S/.test(content);
        const hasDigit = /\\d/.test(content);

        // Remove redundant escape sequences and deduplicate
        content = content.replace(/(\\w)+/g, hasWordChar ? '\\w' : '')
                         .replace(/(\\S)+/g, hasNonWhitespace ? '\\S' : '')
                         .replace(/(\\d)+/g, hasDigit ? '\\d' : '');

        if (hasNonWhitespace)
            // If \S is present, remove \d, \w, and potentially more as they are
redundant
            content = content.replace(/\\d|\\w/g, '');
        else if (hasWordChar)
            // If \w is present, remove \d as it's redundant
            content = content.replace(/\\d/g, '');

        // Replace shorthand character classes with their expanded forms
        content = content.replace(/\\d/g, '0-9');
        content = content.replace(/\\w/g, 'a-zA-Z0-9_');
        content = content.replace(/\\D/g, '^0-9');
```

```
        content = content.replace(/\\W/g, '^a-zA-Z0-9_');
        // Convert \D and \W within character classes to their negated \d and \w forms
        content = content.replace(/\[\\D\]/g, '^0-9');
        content = content.replace(/\[\\W\]/g, '^a-zA-Z0-9_');
        content = content.replace(/\[\\s\]/g, ' \t\n\r\f\v');
        content = content.replace(/\[\\S\]/g, '^ \t\n\r\f\v');
        return content;
    }

    function expandCharacterRange(range) {
        const result = [];
        const start = range.charCodeAt(0);
        const end = range.charCodeAt(2);
        for (let i = start; i <= end; i++) {
            result.push(String.fromCharCode(i));
        }
        return result;
    }

    function postProcessCharacterSet(characters) {
        const digitRange = '0123456789';
        const wordRange =
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_';
        const spaceRange = ' \t\n\r\v\f';

        let charSet = new Set(characters);

        // Helper function to check coverage of range
        function isFullRangeCovered(range) {
            return [...range].every(char => charSet.has(char));
        }
        // Helper function to create ranges from consecutive characters
        function createCharacterRanges(chars) {
            let ranges = [];
            let lastChar = chars[0];
            let rangeStart = lastChar;
            for (let i = 1; i < chars.length; i++) {
                let currentChar = chars[i];
                if (currentChar.charCodeAt(0) !== lastChar.charCodeAt(0) + 1) {
                    if (rangeStart === lastChar)
                        ranges.push(rangeStart);
                    else
                        ranges.push(`${rangeStart}-${lastChar}`);
                    rangeStart = currentChar;
                }
                lastChar = currentChar;
            }
            if (rangeStart === lastChar)
                ranges.push(rangeStart);
            else
                ranges.push(`${rangeStart}-${lastChar}`);
            return ranges;
        }

        let result = [];
        let processedChars = new Set();

        // Check and process space range
        if (isFullRangeCovered(spaceRange)) {
            result.push('\\s');
            [...spaceRange].forEach(char => processedChars.add(char));
```

```
        }

        // Check and process word range
        if (isFullRangeCovered(wordRange)) {
            result.push('\\w');
            [...wordRange].forEach(char => processedChars.add(char));
        }
        else
            { // Check and process digit range
                if (isFullRangeCovered(digitRange)) {
                    result.push('\\d');
                    [...digitRange].forEach(char => processedChars.add(char));
                }
            }

        // Collect unprocessed characters
        const remainingChars = [...charSet].filter(char => !processedChars.has(char));
        if (remainingChars.length > 0) {
            remainingChars.sort(); // Sort to ensure correct range detection
            let ranges = createCharacterRanges(remainingChars);
            result.push(ranges.join(''));
        }

        return result.join('');
        }

       // Remove the brackets for easier processing
    let content = value.replace(/^\[|\]$/g, '');   // Remove the outer brackets
    content = preprocessCharacterClass(content);  // Preprocessing
    const isNegated = value.startsWith('^');
    let characters = new Set();

    // Step 1: Capture valid ranges like a-z, 0-9
    let ranges = content.match(/[^\\]-[^\\]/g) || [];
    // Step 2: Replace the matched ranges with placeholders, then capture the remaining
characters and escaped sequences
    let withoutRanges = content.replace(/[^\\]-[^\\]/g, ''); // Remove ranges
temporarily
    let remainingParts = withoutRanges.match(/\\.|[^\\-]+|-/g) || [];
    // Step 3: Combine the ranges and remaining parts back into the result
    let parts = [...ranges, ...remainingParts];

    parts.forEach(part => {
        if (part.length === 3 && part[1] === '-' && !part.startsWith('\\')) {
            // Correctly handle character ranges
            expandCharacterRange(part).forEach(char => characters.add(char));
        } else if (part === '\\w') {
            // Handle \w by adding all alphanumeric characters and underscore

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_'.split('').forEach(char
=> characters.add(char));
        } else {
            // Add individual characters and other escaped characters
            part.split('').forEach(char => characters.add(char));
        }
    });

    // Convert set to sorted string and the postprocessing
    let normalized = postProcessCharacterSet(Array.from(characters).sort().join(''));
    return { normalized, isNegated };
    }
```

```
    // Traverse each child and optimize recursively
    node.children.forEach(child => optimizeAST(child, changes));

    switch (node.type) {
        case 'CHAR_CLASS': // Optimize [] class
            const { normalized, isNegated } = normalizeCharacterClass(node.value);
            switch (normalized) {
                case '\\d':
                case '0123456789':
                    node.type = 'PREDEFINED_CLASS';
                    changeDesc = `Replaced ${node.value} with`;
                    node.value = isNegated ? '\\D' : '\\d';
                    changes.add(`${changeDesc} ${node.value} to simplify regex`);
                    break;
                case '\\w':
                case 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_':
                    node.type = 'PREDEFINED_CLASS';
                    changeDesc = `Replaced ${node.value} with`;
                    node.value = isNegated ? '\\W' : '\\w';
                    changes.add(`${changeDesc} ${node.value} to simplify regex`);
                    break;
                case '\\s':
                case ' \t\n\r\f\v':
                    node.type = 'PREDEFINED_CLASS';
                    changeDesc = `Replaced ${node.value} with`;
                    node.value = isNegated ? '\\S' : '\\s';
                    changes.add(`${changeDesc} ${node.value} to simplify regex`);
                    break;
                default:
                    let content = node.value;
                    if (content.replace(/^\[|\]$/g, '') != normalized) {
                        changes.add(`${node.value} to [${normalized}] to simplify
regex`);
                        node.value = '[' + normalized + ']';
                    }
            }
            break;
          case 'QUANTIFIER': // Optimize quantifiers
            let optimizationDone = false;  // Flag to track if optimization was
performed

            // Helper function to optimize bracket quantifiers like {n}, {n,}, {n,m}
            // {0,} -> *, {1,} -> +, {0,1} -> ?, {n.n} -> {n}, {n}? -> {n}
            function optimizeBracketQuantifier(quantifier) {
                const regex = /\{(\d+)(?:,(\d*))?\}(\?)?/;
                const match = regex.exec(quantifier);
                let optimizedQuantifier = quantifier;
                let changeDesc = '';

                if (match) {
                    const n = parseInt(match[1], 10);
                    const m = match[2] ? parseInt(match[2], 10) : null;
                    const isLazy = !!match[3];

                    if (m === null) {
                        if (match[0].includes(',')) {
                            if (n === 0) {
                                optimizedQuantifier = isLazy ? '*?' : '*';
                                changeDesc = `Optimized {0,} to ${optimizedQuantifier}`;
                            } else if (n === 1) {
```

```
                                optimizedQuantifier = isLazy ? '+?' : '+';
                                changeDesc = `Optimized {1,} to ${optimizedQuantifier}`;
                            }
                        } else {
                            optimizedQuantifier = `{${n}}`;
                            if(isLazy)
                                changeDesc = `Simplified {${n}}${isLazy ? '?' : ''} to
{${n}}`;
                        }
                    } else {
                        if (n === m) {
                            optimizedQuantifier = `{${n}}`;
                            changeDesc = `Simplified {${n},${m}}${isLazy ? '?' : ''} to
{${n}}`;
                        } else if (n === 0 && m === 1) {
                            optimizedQuantifier = isLazy ? '??' : '?';
                            changeDesc = `Optimized {0,1} to ${optimizedQuantifier}`;
                        } else {
                            optimizedQuantifier = `{${n},${m}}${isLazy ? '?' : ''}`;
                            changeDesc = `Retained quantifier {${n},${m}}${isLazy ? '?'
: ''}`;
                        }
                    }
                }
                return { optimizedQuantifier, changeDesc };
            }

            if (node.children.length === 0) {// Empty quantifier
               // Now, remove the empty quantifier node from the parent's children
               node.parent.children = node.parent.children.filter(child => child !==
node);
               // Log the change for debugging or tracking purposes
               changes.add(`Removed empty Quantifier`);
               break;
            }
            let bracket = optimizeBracketQuantifier(node.value);
            node.value = bracket.optimizedQuantifier;
            if (bracket.changeDesc !== '') changes.add(bracket.changeDesc);

            let isLazy = node.value.slice(-1) === '?' && node.value.slice(-2, -1) !==
'?';
            let baseValue = /*isLazy ? node.value.slice(0, -1) :*/ node.value;
            let newQuantifier = '';
            changeDesc = '';

            // 1. Handle Simple Optimizations First
            switch (baseValue) {
                case '{1}':
                case '{1}?':
                    // {1} or {1}? means exactly one repetition, so it's redundant
                    newQuantifier = '';  // The quantifier becomes redundant and can be
removed
                    if (node.children.length > 0) {
                        // Get the first (and possibly only) child of the quantifier
node
                        const childNode = node.children[0];
                        // Update the quantifier node to "become" the child node
                        node.type = childNode.type;  // Change the quantifier node's
type to the child's type (e.g., 'LITERAL')
                        node.value = childNode.value;  // Set the quantifier node's
value to the child's value
```

```
                        // Replace the quantifier node's children with the child's
children
                        node.children = childNode.children;
                        // Ensure the new children (if any) have their parent updated to
this node
                        if (node.children.length > 0)
                            node.children.forEach(child => child.parent = node);
                        // Log the optimization
                        changes.add(`Removed redundant quantifier ${baseValue} and
simplified to its child node ${childNode.type}`);
                    } else {
                        // In case the quantifier node has no children, just clear its
value and type
                        node.type = '';
                        node.value = '';
                        node.children = [];
                        changes.add(`Removed redundant quantifier ${baseValue} (no
children to move up). THAT SHOULD NOT BE POSSIBLE`);
                    }
                    optimizationDone = true; // Mark that optimization has been done
                    break;
                case '{0}':
                case '{0}?':
                    // Handle the case where the quantifier is {0} or {0}?
                    newQuantifier = '';  // Simplifying the quantifier
                    // Check if the node has children and remove them (because {0} means
no children)
                    let removeNode = node.children.pop();  // Pop removes the last child
                    if (removeNode) {
                        // Now, remove the quantifier node from the parent's children
                        node.parent.children = node.parent.children.filter(child =>
child !== node);
                        // Log the change for debugging or tracking purposes
                        changes.add(`Removed node with zero quantifier ${baseValue}`);
                    }
                    optimizationDone = true; // Mark that optimization has been done
                    break;
                case '?':
                case '*':
                case '+':
                    break;
                case '+?':
                    {
                        let childNode = node.children[0];
                        node.parent.children = node.parent.children.map(child => child
=== node ? childNode : child);
                        childNode.parent = node.parent;
                        changes.add(`Removed lazy +? quantifier and replaced with
${childNode.type}`);
                        optimizationDone = true; // Mark that optimization has been done
                    }
                    break;
                case '*?':
                    {
                        // Updated decision table for lazy quantifier optimization
                        const optimizationTable = {
                            // No prevNode (null) cases
                            'null-null-literal': 'removeLazy',
                            'null-null-nonLiteral': 'keep',
                            'null-literal-literal': 'removeLazy',
                            'null-literal-nonLiteral': 'keep',
```

```
                            'null-nonLiteral-literal': 'removeLazy',
                            'null-nonLiteral-nonLiteral': 'keep',
                            'null-assertion-literal': 'removeLazy',    // Lazy
quantifier can be removed if nextNode is an assertion
                            'null-assertion-nonLiteral': 'removeLazy',

                            // PrevNode is a literal
                            'literal-null-literal': 'removeLazy',
                            'literal-null-nonLiteral': 'keep',
                            'literal-literal-literal': 'keep',
                            'literal-literal-nonLiteral': 'keep',
                            'literal-nonLiteral-literal': 'removeLazy',
                            'literal-nonLiteral-nonLiteral': 'keep',
                            'literal-assertion-literal': 'removeLazy',  // Lazy
quantifier can be removed if nextNode is an assertion
                            'literal-assertion-nonLiteral': 'removeLazy',

                            // PrevNode is a nonLiteral
                            'nonLiteral-null-literal': 'removeLazy',
                            'nonLiteral-null-nonLiteral': 'keep',
                            'nonLiteral-literal-literal': 'keep',
                            'nonLiteral-literal-nonLiteral': 'keep',
                            'nonLiteral-nonLiteral-literal': 'removeLazy',
                            'nonLiteral-nonLiteral-nonLiteral': 'keep',
                            'nonLiteral-assertion-literal': 'removeLazy',  // Lazy
quantifier can be removed if nextNode is an assertion
                            'nonLiteral-assertion-nonLiteral': 'removeLazy'
                        };

                        // Helper function to classify node types and create a key for
the table
                        function classifyNode(node) {
                            if (node === null) return 'null';
                            if (node.type === 'LITERAL' ) return 'literal';
                            if (node.type === 'PREDEFINED' && node.value != '\b' &&
node.value != '\B' ) return 'literal';
                            if (node.type === 'CHAR_CLASS' ) return 'literal';
                            if (node.type === 'ANCHOR_START' || Node.type ===
'ANCHOR_END')  return 'assertion'; // Treat both as "assertion"
                                if (node.type === 'PREDEFINED'&& (node.value === '\\b' ||
node.value === '\\B' ) )  return 'assertion';
                            return 'nonLiteral';
                        }
                        const nextNode = nextSibling(node);  // Get next sibling
                        const prevNode = previousSibling(node);  // Get previous sibling
                        // Classify the node types
                        const prevNodeType = classifyNode(prevNode);
                        const nextNodeType = classifyNode(nextNode);
                        const quantifierNodeType = classifyNode(node.children[0]);

                        // Generate the key for the decision table
                        const decisionKey = `${prevNodeType}-${nextNodeType}-
${quantifierNodeType}`;

                        // Look up the action in the optimization table
                        const action = optimizationTable[decisionKey];

                        const isAnchorOrLookaround = nextNode && ['ANCHOR_START',
'ANCHOR_END', 'LOOKAHEAD', 'NEGATIVE_LOOKAHEAD', 'LOOKBEHIND',
'NEGATIVE_LOOKBEHIND'].includes(nextNode.type);
```

```
                        const isWordBoundary = nextNode && nextNode.type ===
'PREDEFINED_CLASS' && (nextNode.value === '\\b' || nextNode.value === '\\B');

                        // Execute the corresponding action
                        switch (action) {
                            case 'remove':
                                // Fully remove the lazy quantifier (replace *? with an
empty string)
                                let childNode = node.children[0];
                                node.type = childNode.type;
                                node.value = childNode.value;
                                node.children = childNode.children;
                                node.children.forEach(child => child.parent = node);
                                changes.add(`Removed redundant lazy quantifier entirely
followed by ${childNode.type}`);
                                optimizationDone = true; // Mark that optimization has
been done
                               // changes.add(`Removed lazy quantifier *? entirely for
scenario: ${tableKey}`);
                                break;
                            case 'removeLazy':
                                // Remove only the lazy quantifier (replace *? with *)
                                node.value = '*';
                                changes.add(`Removed lazy quantifier and kept greedy *
for scenario: ${decisionKey}`);
                                optimizationDone = true; // Mark that optimization has
been done
                                break;
                            case 'keep':
                                // Retain the lazy quantifier (*?) as it ensures minimal
matching
                                changes.add(`Kept lazy quantifier *? for scenario:
${decisionKey}`);
                                optimizationDone = true; // Mark that optimization has
been done
                                break;
                            default:
                                changes.add(`No action for scenario: ${decisionKey}`);
                                break;
                        }
                    }
                    break;
                case '??':
                    let childNode = node.children[0];
                    node.parent.children = node.parent.children.map(child => child ===
node ? childNode : child);
                    childNode.parent = node.parent;
                    changes.add(`Removed lazy ?? quantifier and replaced with
${childNode.type}`);
                    optimizationDone = true; // Mark that optimization has been done
                    break;
            }

            // 3. Final Step: Only apply newQuantifier if no prior optimizations were
done
            if (!optimizationDone && (newQuantifier )) {
                node.value = newQuantifier + (isLazy ? '?' : '');
                changes.add(changeDesc + node.value);
            }
            break;
        case 'NON_CAPTURING':
```

```
            // Check if the non-capturing group is unnecessary
            if (node.children.length === 0) {// Empty Non-capture
               // Now, remove the empty non-capture node from the parent's children
               node.parent.children = node.parent.children.filter(child => child !==
node);
               // Log the change for debugging or tracking purposes
               changes.add(`Removed empty non-capture`);
            }
            if (node.children.length === 1) {
                let child = node.children[0];
                // Simple literals, predefined classes, or quantifiers
                if (['LITERAL', 'PREDEFINED_CLASS',
'CHAR_CLASS','GROUP'].includes(child.type) || (child.type === 'QUANTIFIER' &&
!/[\{\}]/.test(child.value))) {
                    node.type = child.type;
                    node.value = child.value;
                    node.children = child.children;
                    node.children.forEach(c => c.parent = node);
                    changes.add(`Removed unnecessary non-capturing group
`+(child.value?`around ${child.value}`:``));
            }  else if (node.children.every(child => child.type === 'ALTERNATION') &&
                       (nextSibling(node)==null||nextSibling(node).type==='ANCHOR_END'))
{
                // Handle simple alternation: (?:a|b) -> a|b
                node.type = 'ALTERNATION';
                node.value = ''; // No value needed for alternation
                node.children = node.children[0].children;  // Flatten the alternation
group
                // Update the parent reference for the alternation's children
                node.children.forEach(c => c.parent = node);
                // Log the change
                changes.add(`Removed unnecessary non-capturing group around
alternation.`);
            }
            }

            // Handle non-capturing group at start or end of regex
            if (node.parent && ['ANCHOR_START',
'ANCHOR_END'].includes(node.parent.type)) {
                node.type = node.children[0].type;
                node.value = node.children[0].value;
                node.children = node.children[0].children;
                node.children.forEach(c => c.parent = node);
                changes.add(`Removed unnecessary non-capturing group at start(^)/end($)
of regex.`);
            }
            break;
        case 'ALTERNATION': {
            // Helper functions for prefix and suffix calculation
            function commonPrefix(a, b) {
                let minLength = Math.min(a.length, b.length);
                for (let i = 0; i < minLength; i++) {
                    if (a[i] !== b[i])
                        return a.substring(0, i);
                }
                return a.substring(0, minLength);
            }

            function commonSuffix(a, b) {
                let minLength = Math.min(a.length, b.length);
                for (let i = 0; i < minLength; i++) {
```

```
                    if (a[a.length - 1 - i] !== b[b.length - 1 - i])
                        return a.substring(a.length - i);
                }
                return a.substring(a.length - minLength);
            }

            // Helper function to identify the scenario for alternation
            function identifyAlternationScenario(children) {
                // Check if all children are either character classes, predefined
classes, or single-character literals
                const isCharClassOrPredefined = children.every(child => {
                    // Check for character classes or predefined classes directly
                    return ['CHAR_CLASS', 'PREDEFINED_CLASS'].includes(child.type) ||
                           (child.type === 'LITERAL' && child.value.length === 1); //
Single-character literal
                });

                // Check if all children are single-character literals
                const isLiteralGroups = children.every(child =>
                    child.type === 'LITERAL'
                );

                // Check if all children are single-character literals
                const isSingleLiteralGroups = children.every(child =>
                    (child.type === 'LITERAL' && child.value.length === 1)
                );

                // Return the identified scenarios
                return { isCharClassOrPredefined, isLiteralGroups, isSingleLiteralGroups
};
            }

            // Detect alternation scenario
            const { isCharClassOrPredefined, isLiteralGroups, isSingleLiteralGroups } =
identifyAlternationScenario(node.children);

            // Optimization for char classes or predefined classes in alternation
            if (isCharClassOrPredefined) {
                const mergedClasses = node.children.map(child =>
child.value.replace(/^\[|\]$/g, '')).join('');
                const { normalized, isNegated } =
normalizeCharacterClass(mergedClasses);
                node.value = `[${isNegated ? '^' : ''}${normalized}]`;
                node.children = []; // Clear the children since we replaced them with
the merged class
                node.type = 'CHAR_CLASS';
                changes.add(`Merged character classes in alternation:
${mergedClasses}`);
                break;
            }

            // Optimization for multi-character LITERAL alternation with common
prefix/suffix
            if (isLiteralGroups && !isSingleLiteralGroups) {
                const values = node.children.map(literal => literal.value); // Direct
access to literals
                const hasEmptyAlternation = values[values.length - 1] === '';

                const prefix = values.reduce(commonPrefix);
                const suffix = values.reduce(commonSuffix, values[0]);
```

```
                // Handle common prefix but no suffix (e.g., abc|aef -> a(bc|ef))
                if (prefix.length > 0 && suffix.length === 0) {
                    const groupNode = new RegexNode('GROUP', '', node.parent);

                    // Add the common prefix as a LITERAL node
                    groupNode.addChild(new RegexNode('LITERAL', prefix, groupNode));

                    // Collect the remaining parts after the common prefix (e.g., bc,
ef)
                    const remaining = values.map(value =>
value.substring(prefix.length));

                    // Check if the remaining parts can be optimized into a character
class
                    if (remaining.every(part => part.length === 1)) {
                        // If all remaining parts are single characters, create a
character class
                        const charClass = `[${remaining.join('')}]`;
                        groupNode.addChild(new RegexNode('CHAR_CLASS', charClass,
groupNode));

                        changes.add(`Packed into character class after prefix:
${prefix}${charClass}`);
                    } else {
                        // Otherwise, create a new alternation node for the remaining
parts
                        const alternationNode = new RegexNode('ALTERNATION', '',
groupNode);
                        remaining.forEach(part => {
                            alternationNode.addChild(new RegexNode('LITERAL', part,
alternationNode));
                        });

                        groupNode.addChild(alternationNode);
                        changes.add(`Factored out common prefix with alternation:
${prefix}`);
                    }

                    // Replace the alternation node with the optimized node
                    node.type = groupNode.type;
                    node.value = groupNode.value;
                    node.children = groupNode.children;
                    node.children.forEach(child => child.parent = node);

                    break;
                }

                // Handle common suffix (e.g., abc|dbc -> [ad]bc)
                if (suffix.length > 1) {
                    const remainingPrefixes = values.map(value => value.substring(0,
value.length - suffix.length));

                    if (remainingPrefixes.every(prefix => prefix.length === 1)) {
                        const charClass = `[${remainingPrefixes.join('')}]`;
                        const groupNode = new RegexNode('GROUP', '', node.parent);
                        groupNode.addChild(new RegexNode('CHAR_CLASS', charClass,
groupNode));

                        for (let i = 0; i < suffix.length; i++) {
                            groupNode.addChild(new RegexNode('LITERAL', suffix[i],
groupNode));
```

```
                        }

                        node.type = groupNode.type;
                        node.value = groupNode.value;
                        node.children = groupNode.children;
                        node.children.forEach(child => child.parent = node);
                        changes.add(`Factored out common suffix in alternation:
${suffix}`);
                        break;
                    }
                }
            }

            // Optimization for single-character alternation into CHAR_CLASS
            if (isSingleLiteralGroups) {
                const literalValues = node.children.map(child => child.value); // Direct
access to literals
                const hasEmptyAlternation = literalValues[literalValues.length - 1] ===
'';

                if (literalValues.length > 0) {
                    let charClass = `[${literalValues.join('')}]`;
                    const { normalized, isNegated } =
normalizeCharacterClass(charClass);
                    charClass = `[${isNegated ? '^' : ''}${normalized}]`;
                    const charClassNode = new RegexNode('CHAR_CLASS', charClass,
node.parent);

                    if (hasEmptyAlternation) {
                        const quantifierNode = new RegexNode('QUANTIFIER', '?',
node.parent);
                        quantifierNode.addChild(charClassNode);
                        node.type = quantifierNode.type;
                        node.value = quantifierNode.value;
                        node.children = quantifierNode.children;
                    } else {
                        node.type = 'CHAR_CLASS';
                        node.value = charClass;
                        node.children = [];
                    }

                    changes.add(`Optimized alternation of single literals into character
class: ${charClass}`);
                    break;
                }
            }
            break;
        }
        case 'LOOKAHEAD':
        case 'NEGATIVE_LOOKAHEAD':
            // Handle both positive and negative lookaheads
            // Check if the lookahead or negative lookahead is empty
            if (node.children.length === 0) { // Empty lookahead
                // Now, remove the empty lookahead node from the parent's children
                node.parent.children = node.parent.children.filter(child => child !==
node);
                // Log the change for debugging or tracking purposes
                changes.add(`Removed empty lookahead`);
                break;
            }
```

```
            // Get the lookahead content directly from node children
            let lookaheadContent = node.children.map(child => child.value ||
'').join('');
            let parent = node.parent;

            // 1. Redundant Lookahead Removal: Check if the lookahead is immediately
followed by the same content
            if (parent && parent.children && parent.children.length > 0) {
                // Get the literals following the lookahead
                let nextLiterals = parent.children.slice(parent.children.indexOf(node) +
1)
                    .filter(child => child.type === 'LITERAL')
                    .map(child => child.value)
                    .join('');

                // If the lookahead content matches the following literals
                if (lookaheadContent === nextLiterals) {
                    // Remove the lookahead node
                    parent.children = parent.children.filter(child => child !== node);
                    changes.add(`Removed redundant lookahead: (?=${lookaheadContent})`);
                }
            }

            // 2. Combine Multiple Lookaheads: Combine consecutive lookaheads (both
positive and negative)
            if (parent && parent.children.filter(c =>
c.type.endsWith('LOOKAHEAD')).length > 1) {
                let lookaheadSiblings = parent.children.filter(c =>
c.type.endsWith('LOOKAHEAD'));
                let combinedContent = lookaheadSiblings.map(la => la.children.map(c =>
c.value).join('')).join('|');

                // Create a new lookahead node with combined content
                let combinedLookahead = new RegexNode(node.type, null, parent);
                let combinedAlternationNode = new RegexNode('ALTERNATION',
combinedContent, combinedLookahead);
                combinedLookahead.addChild(combinedAlternationNode);

                // Remove old lookaheads and replace them with the combined one
                parent.children = parent.children.filter(c =>
!lookaheadSiblings.includes(c));
                parent.addChild(combinedLookahead);

                changes.add(`Combined multiple lookaheads into:
(?=${combinedContent})`);
            }

            // 3. Handle Quantifiers within Lookaheads: Ensure quantifiers are properly
accounted for
            if (node.children.some(child => child.type === 'QUANTIFIER')) {
                node.children.forEach(child => {
                    if (child.type === 'QUANTIFIER') {
                        let quantifierValue = child.value;
                        let quantifiedNode = child.children[0];
                        changes.add(`Detected quantifier (${quantifierValue}) in
lookahead for: ${quantifiedNode.value}`);
                    }
                });
            }
            break;
        case 'LOOKBEHIND':
```

```
        case 'NEGATIVE_LOOKBEHIND': {
            // Handle both positive and negative lookbehinds
            let parent = node.parent;

            // 1. Empty Lookbehind Removal
            if (node.children.length === 0) {
                // Remove the empty lookbehind node
                node.parent.children = node.parent.children.filter(child => child !==
node);
                changes.add(`Removed empty lookbehind`);
                break;
            }

         // 2. Combine Multiple Lookbehinds: Combine consecutive lookbehinds (both
positive and negative)
        if (parent && parent.children.filter(c => c.type.endsWith('LOOKBEHIND')).length
> 1) {
            let lookbehinds = parent.children;
            let consecutiveLookbehinds = [];
            let previousIsLookbehind = false;

            // Traverse through all the parent children to find consecutive lookbehinds
            lookbehinds.forEach((child, index) => {
                if (child.type.endsWith('LOOKBEHIND')) {
                    if (previousIsLookbehind) {
                        consecutiveLookbehinds.push(child); // Continue collecting
consecutive lookbehinds
                    } else {
                        consecutiveLookbehinds = [child]; // Start a new group of
consecutive lookbehinds
                    }
                    previousIsLookbehind = true;
                } else {
                    if (consecutiveLookbehinds.length > 1) {
                        // We found a group of consecutive lookbehinds, so combine them
                        combineLookbehinds(consecutiveLookbehinds, parent);
                        changes.add(`Combined consecutive lookbehinds into:
(?<=${getCombinedContent(consecutiveLookbehinds)})`);
                    }
                    previousIsLookbehind = false;
                    consecutiveLookbehinds = []; // Reset if we encounter a non-
lookbehind node
                }
            });

            // Check at the end of the traversal in case there are consecutive
lookbehinds at the end
            if (consecutiveLookbehinds.length > 1) {
                combineLookbehinds(consecutiveLookbehinds, parent);
                changes.add(`Combined consecutive lookbehinds into:
(?<=${getCombinedContent(consecutiveLookbehinds)})`);
            }
        }

        // Helper function to combine consecutive lookbehinds
        function combineLookbehinds(lookbehinds, parent) {
            // Create a new lookbehind node with ALTERNATION
            let combinedLookbehind = new RegexNode(lookbehinds[0].type, null, parent);
            let alternationNode = new RegexNode('ALTERNATION', '', combinedLookbehind);
```

```
            // For each consecutive lookbehind, add a LITERAL node with the content to
the ALTERNATION node
            lookbehinds.forEach(lb => {
                lb.children.forEach(child => {
                    let literalNode = new RegexNode('LITERAL', child.value,
alternationNode);
                    alternationNode.addChild(literalNode);
                });
            });

            // Attach the alternation node to the combined lookbehind
            combinedLookbehind.addChild(alternationNode);

            // Find the position of the first lookbehind in the consecutive group
            let indexOfFirstLookbehind = parent.children.indexOf(lookbehinds[0]);

            // Remove old lookbehinds from parent and insert the combined lookbehind at
the correct position
            parent.children = parent.children.filter(c => !lookbehinds.includes(c));
            parent.children.splice(indexOfFirstLookbehind, 0, combinedLookbehind);  //
Insert at the original position
        }

        // Helper function to get combined content for logging purposes
        function getCombinedContent(lookbehinds) {
            return lookbehinds.map(lb => lb.children.map(c =>
c.value).join('|')).join('|');
        }
            // 3. Redundant Lookbehind Removal
            let lookbehindContent = node.children.map(child => child.value ||
'').join('');


            // Ensure we only check literals before the current node
            if (parent && parent.children.length > 0) {
                let lookbehindIndex = parent.children.indexOf(node);  // Get the
position of the current lookbehind
                let prevLiterals = parent.children.slice(lookbehindIndex-1,
lookbehindIndex)  // Only consider nodes before the lookbehind
                    .filter(child => child.type === 'LITERAL')  // Only consider literal
nodes
                    .map(child => child.value)
                    .reverse()  // Reverse since we're looking backward
                    .join('');

                // If the lookbehind content matches the preceding literals
                if (lookbehindContent === prevLiterals) {
                    // Remove the redundant lookbehind node
                    parent.children = parent.children.filter(child => child !== node);
                    changes.add(`Removed redundant lookbehind:
(?<=${lookbehindContent})`);
                }
            }

            // 4. Handle Quantifiers within Lookbehinds: Ensure quantifiers are properly
accounted for
            if (node.children.some(child => child.type === 'QUANTIFIER')) {
                node.children.forEach(child => {
                    if (child.type === 'QUANTIFIER') {
                        let quantifierValue = child.value;
                        let quantifiedNode = child.children[0];
```

```
                        changes.add(`Detected quantifier (${quantifierValue}) in
lookbehind for: ${quantifiedNode.value}`);
                    }
                });
            }

            break;
        }
        }


    return Array.from(changes).join('\n');
}
```