



Articles / General Programming / Regular Expressions

★ Javascript ★ regular-expression ★ patterns ★ group

A tool for decomposing regular expressions



Henrik Vestermark

10 Oct 2024 CPOL 11 min read 👁 6.9K ↓ 13 📌 9

A JavaScript function for breaking down a regular expression into its base components.

Regular expressions (regex) are a fundamental tool in software development, helping in pattern matching and data manipulation across various programming languages. Despite their utility, regex syntax can be complex and intimidating, especially when dealing with complex patterns. This paper introduces the `decomposingRegex()` function, a robust tool designed to simplify understanding regular expressions by decomposing them into their fundamental components. Through detailed breakdowns of regex structures such as quantifiers, groups, and character classes, `parseRegex()` enhances clarity and helps a deeper understanding, helping debug, learn, and teach regular expression patterns. This paper covers practical applications, provides a guide to regex components, and reviews several online regex tools that further support regex testing and debugging. This paper helps developers to write more efficient and effective code.

Introduction

For those proficient with the basics of regular expressions, advancing your skills can vastly improve your ability to analyze and manipulate strings precisely and efficiently. Regular expressions, while powerful, are often underutilized due to their perceived complexity. However, mastering advanced regex techniques can open up many possibilities for solving complex programming challenges. Expanding your repertoire with advanced quantifiers, lookarounds, and atomic groups can refine your pattern matching. For example, using lazy quantifiers (`*? +?`) allows for minimal matching, which is crucial in scraping tasks or working with large datasets to prevent excessive backtracking. Lookahead and look-behind assertions (`?=...`), (`?<=...`) provide conditional matching capabilities without

consuming characters, enabling you to construct regexes that are both powerful and efficient. Advanced grouping techniques, including non-capturing groups (?...) and named capture groups (?<name>...), enhance the organization and readability of your regular expression patterns. Named capture groups, in particular, facilitate easier maintenance of complex patterns and improve code clarity, mainly when regexes are used in large-scale search-and-replace operations or as part of larger programming constructs. By mastering these techniques, you can feel more organized and efficient in your coding tasks.

Understanding the performance implications of various regex constructs can lead to more optimized and faster executing patterns. Techniques such as character class intersection and subtraction and the use of anchors and word boundaries ensure that your regular expressions run efficiently. Furthermore, exploring engine-specific optimizations and features can provide significant performance benefits. Regular expressions are implemented slightly differently across programming environments, which can affect their behavior and efficiency. Staying informed about the differences between regex implementations in languages like Python, JavaScript, and PHP, as well as in tools like grep and sed, can help you avoid common pitfalls and harness the full power of regex in various contexts. Regular expressions are powerful tools for pattern matching and text manipulation, widely used across different programming environments. However, understanding complex regular expressions can take time and effort. The `decomposingRegex()` function aims to demystify this by breaking down any regular expression into its constituent parts, making it easier to understand, debug, and optimize.

Change log

9 Oct 2024. Change the name of the JavaScript function `parseRegex()` to `decomposingRegex()`. Added a download button.

16-Sep 2024 Minor inaccuracy corrected and some parts restructured.

Contents

- Abstract
- Introduction
- Core Concept of Regex Decomposition
- Deep Dive into `decomposingRegex()`
- Code Explanation
- The `decomposingRegex()` Function: Detailed Explanation
- Summary
- Practical Examples
- Conclusion
- Useful online tools.
 - 1. Regex101
 - 2. RegExr
 - 3. RegexBuddy
 - 4. Regex Pal

- 5. Regex Tester
- 6. Regular expression tester
- Reference

Download decomposing source code

Core Concept of Regex Decomposition

Regex decomposition 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 was `decomposingRegex()`.

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 without 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 noncapturing 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 noncapturing 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.

Deep Dive into decomposingRegex()

The decomposingRegex() function automates breaking down complex regular expressions. It identifies and recursively processes nested groups and other components of the regex. This is particularly useful in educational contexts or debugging sessions where understanding the exact behavior of a regex is necessary.

Code Explanation

At the heart of decomposingRegex() is a pattern-matching logic that recursively identifies groups and handles them.

The decomposingRegex() Function: Detailed Explanation

Purpose: The function aims to parse and decompose a given regular expression into its constituent components, such as capturing and non-capturing groups, and provide a detailed breakdown of simpler regex patterns like quantifiers and character classes. This helps understand and debug complex regex patterns.

Parameters:

- regex: The regular expression as a string to be parsed.
- depth: Tracks the recursion depth, initially set to 0.
- groupCounter: An object that keeps count of the capturing groups to assign each group a unique identifier.

The subfunctions findClosingBrackets(), which find the start and end of capturing and non-capturing groups to ensure that nested groups are handled correctly. Otherwise, the regular expression parses one character at a time and labels the quantities +, * and {}, the character class [], the escaped characters initiated by the \, and the alternation. It further can distinguish between captured groups (), named groups (?<namegroup>...) and different non-capturing groups (?:, (?=, (?!, (?<=, (?<!

JavaScript Function Code:

JavaScript

```
function decomposingRegex(regex, depth = 0, groupCounter = { count: 0 }) {
  let str = ' '.repeat(depth * 2) + `Exploring level ${depth}: ${regex}\n`;
  let i = 0;
  let len = regex.length;
  let end=0;
  let description="";
  let quant='';

  function findClosingBracket(index, open, close) {
    let count = 1;
    while (index < len && count > 0) {
      const char = regex[index];
      if (char === open)
        count++;
      else if (char === close)
        count--;
      index++;
    }
    return index - 1;
  }

  function appendQuantifierInfo(regex, i, char, depth) {
    const isLazy = regex[i + 1] === '?';
    const quant = char + (isLazy ? '?' : '');
    const descriptions = {
      '+': 'one or more occurrences',
      '*': 'zero or more occurrences',
      '?': 'zero or one occurrence'
    };
    const description = (isLazy ? 'lazy ' : '') + descriptions[char];
    str += ' '.repeat((depth + 1) * 2) + `Found Quantifier: ${quant}\t#
    ${description}\n`;
    if (isLazy)
      ++i;
    return i;
  }

  function handleEscapeCharacter(regex, i, depth) {
    const descriptions = {
      'd': "matches any decimal digit",
      'D': "matches any non-digit character",
      'w': "matches any word character (alphanumeric & underscore)",
      'W': "matches any non-word character",
      's': "matches any whitespace character",
      'S': "matches any non-whitespace character",
    };
  }
}
```

```

    'b': "matches a word boundary",
    'B': "matches a non-word boundary",
    'n': "matches a newline character",
    'r': "matches a carriage return character",
    't': "matches a tab character",
    'f': "matches a form feed character",
    'v': "matches a vertical tab character"
  };
  const char = regex[i + 1];
  const description = descriptions[char] || "special character";
  str += ' '.repeat((depth + 1) * 2) + `Found Escaped character: \\${char}\t#
  ${description}\n`;
  return i + 1; // Move past the escaped character
}

while (i < len) {
  const char = regex[i];

  switch (char) {
    case '\\':
      i = handleEscapeCharacter(regex, i, depth);
      break;
    case '[':
      end = findClosingBracket(i + 1, '[', ']');
      str += ' '.repeat((depth + 1) * 2) + `Found Character class:
  ${regex.substring(i, end + 1)}\n`;
      i = end;
      break;
    case '(':
      let next3=regex.substring(i, i + 3);
      let next4=regex.substring(i,i+4);
      if (next3 === '(?:' || next3 === '(?=' || next3==='(?!')
        { // Non-capturing group
          let end = findClosingBracket(i + 1, '(', ')');
          let content = regex.substring(i + 3, end);
          description="";
          switch (next3) {
            case '(?:':
              description='Non capturing group'; break;
            case '(?=':
              description='Non capturing group for positive lookaheads ';
              break;
            case '(?!':
              description='Non capturing group for negative lookaheads';
              break;
          }
          str += ' '.repeat((depth + 1) * 2) + `Found Non-capturing group:
  ${next3}${content})\t#${description}\n`;
          // skip to content
          i += 2;}
      else if(next4==='(?<=' || next4==='(?<!'')
        { // Non-capturing group
          let end = findClosingBracket(i + 1, '(', ')');
          let content = regex.substring(i + 4, end);
          description="";
          switch (next4) {
            case '(?<=':

```

```

        description='Non capturing group for positive lookbehind';
break;
        case '(?<!':
            description='Non capturing group for negative lookbehind ';
break;
    }
    str += ' '.repeat((depth + 1) * 2) + `Found Non-capturing group:
${next4}${content})\t#${description}\n`;
    // skip to content
    i += 3;}
else {
    // Check for name capturing group and bypass it
    let name="";
    if (next3 === '(?<'){
        const closeTagIndex = regex.indexOf('>', i);
        if (closeTagIndex !== -1)
            {
                name=regex.substring(i+3,closeTagIndex);
                i = closeTagIndex; // Move the index to the position right after
'>'
            }
    }
    groupCounter.count++;
    let end = findClosingBracket(i + 1, '(', ')');
    let content = regex.substring(i + 1, end);
    str += ' '.repeat((depth + 1) * 2) + `Found Capturing group
${groupCounter.count}: (${content})\n`;
    if(name!="")
        str += ' '.repeat((depth + 1) * 2) + `Is also a named Capturing
group: ${name}\n`;
    str += decomposingRegex(content, depth + 1, groupCounter);
    i = end;
    }
break;
case '{':
    end = findClosingBracket(i + 1, '{', '}');
    quant = regex.substring(i, end + 1);
    description = "";
    if(regex[end+1]==='?')
        { description="lazy "; ++end; }
    if (quant.match(/^{\d+\}$/) ) {
        description += "matches exactly " + quant.match(/\d+\/)[0] + " times";
    } else if (quant.match(/^{\d+,\}$/) ) {
        description += "matches at least " + quant.match(/\d+\/)[0] + " times";
    } else if (quant.match(/^{\d+,\d+\}$/) ) {
        let numbers = quant.match(/\d+\/g);
        description += "matches between " + numbers[0] + " and " + numbers[1] +
" times";
    }
    str += ' '.repeat((depth + 1) * 2) + `Found Quantifier: ${quant} \t#
${description}\n`;
    i = end;
    break;
case '+':
case '*':
case '?':
    i = appendQuantifierInfo(regex, i, char, depth);

```



```

        break;
    case '|':
        str += ' '.repeat((depth + 1) * 2) + `Found Alternation: |\n`;
        break;
    case '.':
        str += ' '.repeat((depth + 1) * 2) + `Found Any character: .\n`;
        break;
    }
    i++;
}

return str;
}

```

Summary

The `decomposingRegex()` function offers a comprehensive approach to dissecting and understanding regular expressions. Recursively parsing groups and detailing simpler parts provides valuable insights into the structure and functionality of complex regex patterns, making it an essential tool for developers working with regular expressions in their projects.

Practical Examples

For instance, consider the regex:

```

/([+-]?(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?/g

```

Which is a regular expression that matches a signed floating-point number. Using `decomposingRegex()`, this can be decomposed to reveal its structure: an optional sign, a number with an optional decimal part, followed by an optional exponent. Decomposition clarifies each component's role and syntax.

The decomposition result is shown below:

```

Exploring level 0: /([+-]?(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?/g
Found Capturing group 1: ([+-]?(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?
Exploring level 1: [+-]?(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?
Found Character class: [+-]
Found Quantifier: ? # zero or one occurrence
Found Capturing group 2: (\d+(\.\d*)?)|\.\d+
Exploring level 2: \d+(\.\d*)?)|\.\d+
Found Escaped character: \d # matches any decimal digit
Found Quantifier: + # one or more occurrences

```

```

Found Capturing group 3: (\\.\\d*)
Exploring level 3: \\.\\d*
  Found Escaped character: \.    # special character
  Found Escaped character: \\d    # matches any decimal digit
  Found Quantifier: *    # zero or more occurrences
Found Quantifier: ?    # zero or one occurrence
Found Alternation: |
Found Escaped character: \.    # special character
Found Escaped character: \\d    # matches any decimal digit
Found Quantifier: +    # one or more occurrences
Found Capturing group 4: ([eE][+-]?\\d+)
Exploring level 2: [eE][+-]?\\d+
  Found Character class: [eE]
  Found Character class: [+-]
  Found Quantifier: ?    # zero or one occurrence
  Found Escaped character: \\d    # matches any decimal digit
  Found Quantifier: +    # one or more occurrences
Found Quantifier: ?    # zero or one occurrence

```

Notice that group 1, group 2, etc., matched the \$1, and \$2 parameters, etc.

```

This is displayed in the result window in the web app.
RegExp=/[+-]?(\\d+(\\.\\d*)?|\\.\\d+)([eE][+-]?\\d+)?/g
Input: -12.34e+56 => Matched: -12.34e+56
      $1=12.34 $2=.34 $3=e+56

```

Conclusion

Understanding regular expressions through decomposition not only aids in debugging and development but also enhances one's ability to write more efficient and error-free regexes. Tools like `decomposingRegex()` play an important role in this educational process. The Author's regular expression tool is the sixth tool mentioned. It has a unique feature allows you to test two regular expressions simultaneously. 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.

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](https://regex101.com/)
- **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 to learn, build, and test 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.
 - 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 testing JavaScript regular expressions quickly. 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 \(hvks.com\)](https://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 expression for easier debugging and understanding.
 - 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 \(hvks.com\)](https://hvks.com)

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOl\)](https://www.apache.org/licenses/LICENSE-2.0)

Written By


Henrik Vestermark

 Denmark

With a Master's degree in Electronic Engineering, I've further specialized in the intersection of Computer Science, Numerical Analysis, and Applied Mathematics. In my earlier career, I gained extensive experience in developing compilers, linkers, and interpreters. This foundational work has

equipped me to later focus on designing high-performance algorithms for arbitrary precision arithmetic. My expertise lies in leveraging optimization techniques to solve complex problems within these fields.

Comments and Discussions

 **2 messages** have been posted for this article Visit

<https://www.codeproject.com/Articles/5387559/A-tool-for-decomposing-regular-expressions> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Posted 2 Sep 2024

Article Copyright 2024 by Henrik
Vestermark
Everything else Copyright ©
[CodeProject](#), 1999-2024

Web01 2.8:2024-07-22:1