# *ParseRegex User Manual*

## JavaScript code

***Version 1.0***

***By Henrik Vestermark (hve@hvks.com)***

# Contents

## Introduction to `parseRegex.js` Functions

The **`parseRegex.js`** module provides a robust framework for parsing, manipulating, and executing regular expressions through Abstract Syntax Trees (AST). Each function plays a vital role in the lifecycle of working with regular expressions in a structured, programmatic way:

1. **`parseRegexToAST`**: Transforms a regular expression string into an Abstract Syntax Tree (AST). This function is the foundation for regex analysis and manipulation, enabling detailed exploration of regex patterns.
2. **`ASTtoString`**: Converts an AST back into its equivalent string representation. This function ensures that modifications to the AST can be seamlessly converted into a usable regex format.
3. **`ASTtoRegex`**: Compiles an AST into a JavaScript `RegExp` object. This function allows regex patterns constructed or optimized in AST form to be directly used for matching and validation tasks.
4. **`executeAST`**: Executes a regular expression represented by an AST against a string input. This function adds flexibility by providing detailed matching results, including statistics and named group handling.
5. **`optimizeAST`**: Enhances the performance and simplicity of an AST by removing redundancies and applying structural improvements. It ensures that complex or verbose regex patterns are streamlined for optimal performance.
6. **`createRegexASTIterator`**: This function creates an iterator for traversing matches of a regex (in AST form) over a given input string. It is valuable for efficiently processing multiple matches with detailed stats.
7. **`createRegexIterator`**: Similar to `createRegexASTIterator`, this function works directly with regex patterns (not ASTs), allowing for straightforward iteration over matches in a string.

## 1. parseRegexToAST

- **Purpose**: Parses a regular expression string into its corresponding Abstract Syntax Tree (AST) representation, allowing structured analysis and manipulation of the regex pattern.
- **Parameters**:
  - `regex` (string): The regular expression pattern to be parsed.
- **Returns**:
  - `ASTNode`: The root node of the AST representing the parsed regular expression.
- **Usage Example**:

```
const regex = '[a-z]+\\d*';
const ast = parseRegexToAST(regex);
console.log(ast);
```

---

## 2. ASTtoString

- **Purpose**: Converts an Abstract Syntax Tree (AST) back into its string representation, preserving the original regular expression pattern.
- **Parameters**:
  - `ast` (ASTNode): The root node of the AST to be converted.
- **Returns**:
  - `string`: The string representation of the regular expression.
- **Usage Example**:

```
const regex = '[a-z]+\\d*';
const ast = parseRegexToAST(regex);
const regexString = ASTtoString(ast);
console.log(regexString); // Outputs: '[a-z]+\\d*'
```

---

## 3. ASTtoRegex

- **Purpose**: Converts an Abstract Syntax Tree (AST) into a RegExp object, enabling direct use of the regex for pattern matching operations.
- **Parameters**:
  - `ast` (ASTNode): The root node of the AST to be converted.
  - `flags` (string, optional): A string containing any combination of regular expression flags (e.g., 'g', 'i', 'm').
- **Returns**:
  - `RegExp`: The RegExp object constructed from the AST.
- **Usage Example**:

```
const regex = '[a-z]+\\d*';
const ast = parseRegexToAST(regex);
const regExpObject = ASTtoRegex(ast, 'gi');
console.log(regExpObject); // Outputs: /[a-z]+\d*/gi
```

---

## 4. executeAST

- **Purpose**: Executes a regular expression represented by an AST against a given input string, returning match results and statistics.
- **Parameters**:
  - `astNode` (ASTNode): The root node of the AST representing the regular expression.
  - `input` (string): The input string will be tested against the regular expression.
  - `index` (number): The starting position in the input string for the match attempt.
  - `namedGroups` (object, optional): An object to store named capturing groups.
- **Returns**:
  - `object`: An object containing the following properties:

  - `matches` (array): An array of match results, each being an array where the first element is the full match, followed by any capturing group matches.
  - `namedGroups` (object): An object containing named capturing groups and their corresponding matched substrings.
  - `nextIndex` (number): The index in the input string immediately following the last matched substring.
  - `stats` (object): Various statistics about the matching process, such as match count, execution time, backtrack count, etc.
- **Usage Example**:

```
const regex = '(?<word>[a-z]+)(\\d*)';
const ast = parseRegexToAST(regex);
const input = 'hello123';
const result = executeAST(ast, input, 0);

console.log(result.matches); // Outputs: [['hello123', 'hello', '123']]
console.log(result.namedGroups); // Outputs: { word: 'hello' }
console.log(result.nextIndex); // Outputs: 8
console.log(result.stats); // Outputs: { matchCount: 1, executionTime:
..., backtrackCount: ..., ... }
```

---

## 5. optimizeAST

- **Purpose**: Optimizes an Abstract Syntax Tree (AST) by simplifying redundant nodes, combining patterns, and improving the performance of the regex while maintaining equivalent functionality.
- **Parameters**:
  - `node` (ASTNode): The root node of the AST to be optimized.
  - `changes` (Set, optional): Tracks a detailed list of modifications made during the optimization process.
- **Returns**:
  - `ASTNode`: The root node of the optimized AST.
- **Usage Example**:

```
const regex = '(?:a|b|c)+';
const ast = parseRegexToAST(regex);
const optimizedAST = optimizeAST(ast);
console.log(optimizedAST);
```

- **Optimization Techniques**:
  - **Merge Adjacent Character Classes**: Combines overlapping or adjacent character classes into a single optimized class.
    - Example: `[a-zA-Z0-9]|[0-9]` → `[a-zA-Z0-9]`
  - **Simplify Quantifiers**: Converts verbose quantifiers to their simpler equivalents.
    - Example: `{1,}` → `+`, `{0,1}` → `?`
  - **Remove Redundant Nodes**: Eliminates unnecessary non-capturing groups, empty groups, or redundant alternation nodes.

        - Example: `(?:a|b)` → `a|b`
    - **Factor Common Prefixes and Suffixes**: Consolidates shared patterns in alternations.
        - Example: `abc|adc` → `a(b|d)c`
    - **Handle Anchors and Assertions**: Removes redundant anchors (`^`, `$`) or ensures they are used optimally with lookaheads/lookbehinds.
- **Example Optimization Process**:

```
const regex = '(?:[a-z]|[0-9]){1,}';
const ast = parseRegexToAST(regex);
const optimizedAST = optimizeAST(ast);
console.log(ASTtoString(optimizedAST)); // Outputs: '[a-z0-9]+'
```

- **Key Notes**:
    - The function traverses the AST recursively, applying local and global optimization strategies.
    - It includes handling for alternations, quantifiers, character classes, and lookaheads/lookbehinds.
    - Provides detailed logs of all changes through the `changes` parameter.

---

## 6. createRegexASTIterator

**Purpose**: Creates an iterator that traverses matches of a regular expression represented by an Abstract Syntax Tree (AST) over a given input string. This iterator provides detailed statistics about the matching process.

**Parameters**:

- `astNode` (ASTNode): The root node of the AST representing the regular expression.
- `input` (string): The input string to be searched for matches.
- `modifiers` (string, optional): A string containing any combination of regular expression flags (e.g., 'g' for global, 'i' for case-insensitive).

**Returns**:

- `object`: An iterator object with the following methods:
    - `next()`: Returns the next match result as an object with properties `value` (the matched substring) and `done` (a boolean indicating if the iteration is complete).
    - `getStats()`: Returns an object containing statistics about the matching process, such as `matchCount`, `executionTime`, `backtrackCount`, etc.

- `[Symbol.iterator]`: Makes the object iterable, allowing it to be used in `for...of` loops.

**Usage Example**:

```
const regex = '(\\d+)';
const ast = parseRegexToAST(regex);
const input = '123 abc 456';
const iterator = createRegexASTIterator(ast, input, 'g');

for (const match of iterator) {
    console.log(match); // Outputs: ['123'] and then ['456']
}

const stats = iterator.getStats();
console.log(stats); // Outputs: { matchCount: 2, executionTime: ...,
backtrackCount: ..., ... }
```

---

## 7. createRegexIterator

**Purpose**: Creates an iterator that traverses matches of a regular expression pattern over a given input string. This iterator provides detailed statistics about the matching process.

**Parameters**:

- `pattern` (string): The regular expression pattern to be used for matching.
- `input` (string): The input string to be searched for matches.
- `modifiers` (string, optional): A string containing any combination of regular expression flags (e.g., 'g' for global, 'i' for case-insensitive).

**Returns**:

- `object`: An iterator object with the following methods:
  - `next()`: Returns the next match result as an object with properties `value` (the matched substring) and `done` (a boolean indicating if the iteration is complete).
  - `getStats()`: Returns an object containing statistics about the matching process, such as `matchCount`, `executionTime`, `backtrackCount`, etc.
  - `[Symbol.iterator]`: Makes the object iterable, allowing it to be used in `for...of` loops.

**Usage Example**:

```
const pattern = '\\d+';
const input = '123 abc 456';
const iterator = createRegexIterator(pattern, input, 'g');

for (const match of iterator) {
    console.log(match); // Outputs: ['123'] and then ['456']
```

```
}

const stats = iterator.getStats();
console.log(stats); // Outputs: { matchCount: 2, executionTime: ...,
backtrackCount: ..., ... }
```

---

These functions facilitate the creation of iterators for traversing matches of regular expressions, providing both the matched substrings and detailed statistics about the matching process.

## Example: Encapsulating Operations with RegexASTHandler

The RegexASTHandler at the start of the parseRegex.js file serves as a convenient abstraction to encapsulate and manage the operations provided by the module. Here's an example description that showcases its usage and purpose:

The RegexASTHandler class provides a higher-level abstraction for working with regular expressions through their AST representations. It encapsulates common operations such as parsing, optimizing, and executing regex patterns, making it easier to manage complex regex workflows.

Features of RegexASTHandler

1. Automatically parses a regex string into its AST representation upon initialization.
2. Provides methods to:
   - Convert the AST back into a regex string or a RegExp object.
   - Optimize the AST for performance and simplicity.
   - Create an iterator for traversing matches in an input string.

Class Definition

The RegexASTHandler class encapsulates the following methods:

- toString(): Converts the stored AST to its string representation.
- toRegex(): Returns a RegExp object derived from the AST.
- optimize(): Optimizes the stored AST for better performance.
- createIterator(input, modifiers): Creates an iterator for matching the regex against an input string.

Usage Example

Here's how the RegexASTHandler class can be used:

```
// Import the module (if applicable)
// const { RegexASTHandler } = require('./parseRegex.js');

// Step 1: Initialize the handler with a regex pattern
const handler = new RegexASTHandler("[0-9]+");

// Step 2: Convert the AST back into a string
console.log("Original Regex:", handler.toString()); // Outputs: "[0-9]+"

// Step 3: Optimize the AST
handler.optimize();
console.log("Optimized Regex:", handler.toString()); // Outputs: "\\d+"

// Step 4: Create an iterator to find matches in a string
```

```
const input = "123 456 789";
const iterator = handler.createIterator(input, "g");

// Step 5: Iterate over matches
for (const match of iterator) {
    console.log("Match:", match); // Outputs: ["123"], ["456"], ["789"]
}

// Additional Step: Access the optimized regex as a RegExp object
const regexObject = handler.toRegex();
console.log(regexObject.test("456")); // Outputs: true
```

Explanation

- Initialization: The regex pattern is automatically parsed into an AST when the RegexASTHandler is instantiated.
- Optimization: The optimize method simplifies and enhances the regex for better performance.
- Iteration: The createIterator method provides a convenient way to traverse matches, along with detailed matching statistics if needed.