Making a simple RegEx engine: Part 1: Introduction to RegEx

Git revision #fee2a364[1]
Modified at 26. July 2025 14:20

Written by alex_s168[2]

Introduction

If you are any kind of programmer, you've probably heard of RegEx[3]

RegEx (Regular expression) is kind of like a small programming language used to define string search and replace patterns.

RegEx might seem overwhelming at first, but you can learn the most important features of RegEx very quickly.

It is important to mention that there is not a single standard for RegEx syntax, but instead each "implementation" has it's own quirks, and additional features. Most common features however behave identically on most RegEx "engines"/implementations.

Syntax

The behavior of RegEx expressions / patterns depends on the match options passed to the RegEx engine.

Common match options:
- Anchored at start and end of line
- Case insensitive
- multi-line or instead whole string

"Atoms"

In this article, we will refer to single expression parts as "atoms".

Characters

Just use the character that you want to match. For example `a` to match an `a`. This however does not work for all characters, because many are part of special RegEx syntax.

[1] https://github.com/alex-s168/website/tree/fee2a36453b624f2e6ede5186b5f3e59aa3e6cc7
[2] https://alex.vxcc.dev
[3] https://en.wikipedia.org/wiki/Regular_expression

Escaped Characters

Thee previously mentioned special characters like `[` can be matched by putting a backslash in front of them: `\[`

Pattern	Description
`\\`	match a literal backslash
`\n`	match a new-line

Character Groups

RegEx engines already define some groups of characters that can make writing RegEx expressions quicker.

Pattern	Description
`.`	any character except for line breaks
`\s`	any whitespace or line break
`\S`	any character except whitespaces or line breaks
`\d`	any digit from 0 to 9
`\D`	any character except digits from 0 to 9
`\w`	a letter, digit, or underscore
`\W`	any character except for letters, digits, and underscores

Anchors

`^` is used to assert the beginning of a line in multi-line mode, or the beginning of the string in whole-string mode.

`$` is used to assert the end of a line in multi-line mode, or the end of the string in whole-string mode.

The behaviours of these depend on the match options

Greedy VS Lazy

Some combinators will either match “lazy”, or “greedy”.

Lazy is when the engine only matches as many characters required to get to the next step. This should almost always be used.

Greedy matching is when the engine tries to match as many characters as possible. The problem with this is that it might cause “backtracking”, which happens when the engine goes back in the pattern multiple times to ensure that as many characters as possible where matched. This can cause big performance issues.

Combinators

Multiple atoms can be combined together to form more complex patterns.

Chain

When two expressions are next to each other, they will be chained together, which means that both will be evaluated in-order.

Example: `x\d` matches a `x` and then a digit, like for example `x9`

Or

Two expressions separated by a `|` cause the RegEx engine to first try to match the left side, and only if it fails, it tries the right side instead.

Note that “or” has a long left and right scope, which means that `ab|cd` will match either `ab` or `cd`

Or-Not

Tries to match the expression on the left to it, but won’t error if it doesn’t succeed.

Note that “or-not” has a short left scope, which means that `ab?` will always match `a`, and then try to match `b`

Repeated

A expression followed by either a `*` for greedy repeat, or a `*?` for lazy repeat.

This matches as many times as possible, but can also match the pattern zero times.

Note that this has a short left scope.

Repeated At Least Once

A expression followed by either a `+` for greedy repeat, or a `+?` for lazy repeat.

This matches as many times as possible, and at least one time.

Note that this has a short left scope.

(Non-Capture) Group

Groups multiple expressions together for scoping.

Example: `(?:abc)` will just match `abc`

Capture Group

Similar to Non-Capture Groups except that they capture the matched text. This allows the matched text of the inner expression to be extracted later.

Capture group IDs are enumerated from left to right, starting with 1.

Example: `(abc)de` will match `abcde`, and store `abc` in group 1.

Character Set

By surrounding multiple characters in square brackets, the engine will match any of them. Special characters or expressions won't be parsed inside them, which means that this can also be used to escape characters.

For example: `[abc]` will match either `a`, `b` or `c`.

and `[ab(?:c)]` will match either `a`, `b`, `(`, `?`, `:`, `c`, or `)`.

Character groups and escaped characters still work inside character sets.

Character sets can also contain ranges. For example: `[0-9a-z]` will match either any digit, or any lowercase letter.

Conclusion

RegEx is perfect for when you just want to match some patterns, but the syntax can make patterns very hard to read or modify.

In the next article, we will start to dive into implementing RegEx.

Stay tuned!