Complex Regular Expressions Are Just Paper Tigers

Recently, while researching HTML parsing, encountered a regular expression:

```
/([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?([\/, ]+)/is
```

It is used to match CSS selectors, such as `div > ul`.

In the past, I instinctively shied away from such complex expressions. Today, let’ s thoroughly understand it! Sometimes you have to push yourself to the limit!

Matching `div > ul`

I found a website, https://regex101.com/, that allows online matching and provides explanations.

Although the explanations on the right help clarify some aspects, the specifics of matching aren’ t entirely clear. Let’ s analyze a few examples.

The code that uses this regular expression is:

```
$matches = [];
preg_match_all($this->pattern, trim($selector).' ', $matches, PREG_SET_ORDER);
```

`preg_match_all` is used to get all the strings that match the pattern. For example:

```
preg_match_all("abc", "abcdabc", $matches)
```

The first parameter is the pattern, the second parameter is the string to be matched, and the third parameter is the result reference. After running, the `matches` array will contain two `abc`.

With this understanding, the `div > ul` in the example matches the first four characters `div >`. `regex101` does not support `preg_match_all` directly, but adding the `g` modifier will achieve this:

By adding `g`, it matches all occurrences instead of stopping at the first match.

With the `g` modifier, we matched `div > ul`:

On the right, it shows that the first match, `div`, is matched by the first group rule, and the space is matched by the seventh group rule.

Let’ s look at the explanation for the first group rule:

In this long expression, the first set of parentheses is the first group rule. This is a capturing group. Parentheses do not match themselves but are used for grouping. `[]` indicates a character set, and the rules inside define what characters can be matched. This character set includes:

- `\w` represents letters, digits, and underscores.

- `-:` directly includes these two characters in the set.
- `\*` requires escaping because `*` is a special character in regular expressions. `\*` indicates a literal `*`.
- `>` simply includes the `>` character.

`[\w-:\*>]*` with the trailing `*` means the preceding characters can appear zero or more times, matching as many as possible. It matches `div` because `\w` matches `d`, `i`, `v`. It stops at the space because the space is not in the set `[]`. Capturing groups mean this match will appear in the result array. There are also non-capturing groups, denoted by `(?:)`. For example, `([\w-:\*>]*)` can be written as `(?:[\w-:\*>]*)` if you don’ t need the match result.

Why use parentheses if not capturing? Parentheses are used for grouping, which is still meaningful. Refer to What is a non-capturing group? (?:) - StackOverflow.

After understanding how `div` matches the first group rule, let’ s see why the space matches the seventh group rule.

`[\/, ]` means it matches any of these four characters, and `+` means one or more occurrences, matching as many as possible. The space matches because it is included. The next character after `div` is `>`, so it stops matching the seventh group rule.

Now we understand how `div` is matched. Why don’ t groups 2 to 6 match the space, leaving it to group 7?

The second part explanation:

First, `(?:)` indicates a non-capturing group, and the trailing `?` means it can appear zero or one time. `(?:\#([\w-]+)|\.([\w-]+))?` can be present or absent. Removing the outermost `?:?`, we have `\#([\w-]+)|\.([\w-]+)`, and the `|` means “or,” matching either part. `\#([\w-]+)` matches `#` followed by other characters. `\.([\w-]+)` matches `.` followed by other characters.

Groups 2 to 6 are not matched because the space does not satisfy the initial characters of these groups. With the `?` modifier, they can be absent, so it moves to group 7.

Next, `div > ul` after the `>` works the same way:

The first group rule `([\w-:\*>]*)` matches `>`, and the seventh group rule `([\/, ]+)` matches the space. Then `ul` matches like `div`.

Matching `#answer-4185009 > table > tbody > td.answercell > div > pre`

Now let’ s look at a more complex selector `#answer-4185009 > table > tbody > td.answercell > div > pre` (you can paste it into https://regex101.com/ to test):

This was copied from Chrome:

The first match:

The first group rule `([\w-:\*>]*)` does not match `#`, so it matches zero times. The second group rule explanation:

The `|` means "or." The part before the `|` is `\#([\w-]+)`, where `\#` matches `#` and `[\w-]+` matches `answer-4185009`. The second part, `\.([\w-]+)`, would match if it were `.answer-4185009`.

Now for `td.answercell`:

The first group rule `([\w-:\*>]*)` matches `td`, and the second part `(?:\#([\w-]+)|\.([\w-]+))?` matches `.answercell` because of the `.`.

Matching `a[href="http://google.com/"]`

Now let' s match the selector `a[href="http://google.com/"]`:

The third part of the expression is:

The third part is `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`. The outermost `(?:)` indicates a non-capturing group, and the trailing `?` means it can appear zero or one time. Removing `(?:)?`, we have `\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\]`. `\[`, `[` matches `[`. `@?` means `@` is optional. The group `(!?[\w-:]+)` matches `href`. The next part `(?:([!*^$]?=)["']?(.*?)["']?)` is a non-capturing group, so removing `(?:)?`, we have `([!*^$]?=)["']?(.*?)["']?`. `([!*^$]?=)` means `[!*^$]?` matches zero or one of `!`, `*`, `^`, or `$`, followed by `=`. `["']?(.*?)["']?` matches `"http://google.com/"`, where `["']?` matches optional `"` or `'`. Removing `["']?`, we have `(.*?)`, which matches `http://google.com/`. The `*?` modifier means match as few as possible, ensuring that `"` or `'` is left for the next part. So the selector `a[href="http://google.com/"]` is fully matched.

Summary

Finally understood! The complex expression `([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]`
`]+)` consists of four main parts:

- `([\w-:\*>]*)`
- `(?:\#([\w-]+)|\.([\w-]+))?`
- `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`
- `([\/, ]+)`

The most complex third part has smaller components:

- `\[`
- `(!?[\w-:]+)`
- `(?:([!*^$]?=)["']?(.*?)["']?)?`
- `\]`

These smaller parts can be tackled individually. By analyzing multiple examples and using the explanations from https://regex101.com/, we can understand this seemingly complex regular expression. It turns out to be just a paper tiger!