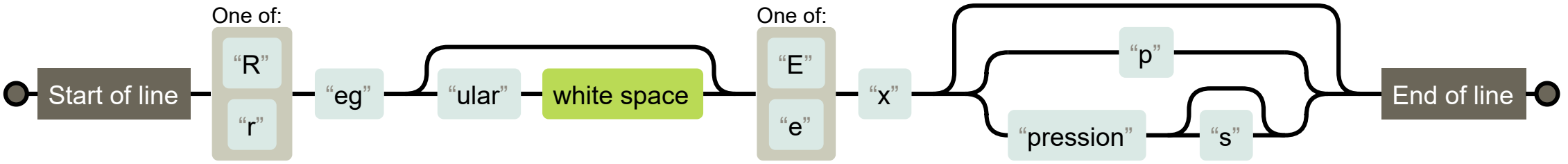
/^[Rr]eg(ular\s)?[Ee]x(p|pressions?)?$/



jordankasper.com/regex

Session Feedback:

Image credit: regexper.com

Jordan Kasper | @jakerella

WHAT ARE THEY?
STRING PATTERN MATCHERS

- Test a string against a pattern (true/false)
- Match specific data in a string
- Often used for data/input validation
- Can efficiently* find data in large text (e.g. log files)
- Sometimes used to find and replace specific text

WHAT LANGUAGE?

~~WHAT LANGUAGE?~~

WHAT ENGINE?

REGEX ENGINES

PCRE	Apache, Nginx, PHP, Erlang, Elixir
POSIX (BRE/ERE)	grep, sed, MySQL, Oracle, R*
Oniguruma	Ruby, Atom, Sublime, jq
Irregexp	Node.js, Chrome, Firefox
Standard libs	Java*, Perl, Python*, Rust, .NET

* Many languages have multiple engines available via libraries (e.g. Java's standard lib versus JRegex).

** Also, not all features are supported in a language, even if they are supported by the engine!

https://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines

WHAT DO THEY LOOK LIKE?

```
/the regex goes in here/
```

```
`the regex goes in here`
```

```
"the regex goes in here"
```

```
"/the regex goes in here/"
```

LANGUAGE-SPECIFIC

JavaScript

```
/foo/.test("foobar")
```

PHP

```
preg_match("/foo/", "foobar");
```

Ruby

```
/foo/ =~ "foobar"
```

Perl

```
"foobar" =~ /foo/
```

COMMON ELEMENTS

(Using JavaScript syntax)

SIMPLE CHARACTER TEST

```
/a/.test("JavaScript Rules")    // true
```

```
/z/.test("JavaScript Rules")    // false
```

TESTING VERSUS MATCHING

true/false vs. **matched groups**

```
/a/.test("JavaScript Rules")    // true
"JavaScript Rules".match(/a/)   // ["a"]
```

```
"JavaScript Rules".match(/z/)   // null
```

WHERE'S THE OTHER "a"?

```
"JavaScript Rules".match(/a/)    // ["a"]
```

We need to make it global!

```
"JavaScript Rules".match(/a/g)   // ["a", "a"]
```

The "g" here is called a "flag".

CASE SENSITIVITY

```
"JavaScript Rules".match(/r/)     // ["r"]

// make it global
"JavaScript Rules".match(/r/g)    // ["r"]

// make it case-insensitive AND global
"JavaScript Rules".match(/r/ig)   // ["r", "R"]
```

MULTIPLE CHARACTERS

```
/script/i.test("JavaScript Rules")    // true

"JavaScript Rules".match(/script/i)   // ["Script"]
```

MATCH ANY CHARACTER

```
/jordan/i.test("Jordan")    // true
```

```
/jord.n/i.test("Jordan")    // true
/jord.n/i.test("Jordon")    // true
/jord.n/i.test("Jordyn")    // true
```

The "." does NOT match newline (or carriage return)!

```
"foo\nbar".match(/./g)  // ["f", "o", "o", "b", "a", "r"]

"foo\tbar".match(/./g)  // ["f", "o", "o", "\t", "b", "a", "r"]
```

CLASSES, RANGES, AND REPETITION

`[]`, `+`, `{}`

```
"c52a93".match(/c52a93/)                  // ["c52a93"]
"b5d055".match(/c52a93/)                  // null
"c52a93".match(/[0123456789abcdef]/)      // ["c"]
"c52a93".match(/[0123456789abcdef]/g)     // ["c", "5", "2", "a", "9",
"c52a93".match(/[0123456789abcdef]+/)     // ["c52a93"]
"c52a93".match(/[0-9a-f]+/)               // ["c52a93"]
"c52a9300".match(/[0-9a-f]+/)             // ["c52a9300"]
"c52a9300".match(/[0-9a-f]{6}/)           // ["c52a93"]
"c52a9300".match(/[0-9a-f]{6,8}/)         // ["c52a9300"]
```

MAKING SOMETHING OPTIONAL

?

```
"#c52a93".match(/[0-9a-f]{6,8}/)    // ["c52a93"]
"#c52a93".match(/#[0-9a-f]{6,8}/)   // ["#c52a93"]
"c52a93".match(/#[0-9a-f]{6,8}/)    // null
"c52a93".match(/#?[0-9a-f]{6,8}/)   // ["c52a93"]
```

ZERO OR MORE

*

```
"value='c52a93'".match(/value='[0-9a-f]+'/)   // ["value='c52a93'"]
"value=''".match(/value='[0-9a-f]+'/)         // null
"value=''".match(/value='[0-9a-f]*'/)         // ["value=''"]
```

NEGATION

[^]

```
"jordan kasper".match(/[a-z]+/g)      // [ "jordan", "kasper" ]
"jordan o'moran".match(/[a-z]+/g)     // [ "jordan", "o", "moran" ]
"jordan o'moran".match(/[^ ]+/g)      // [ "jordan", "o'moran" ]
```

GREEDINESS AND LAZINESS

RegEx likes to eat up characters, so be careful!

```
"<p class='foo' id='bar'>".match(/[a-z]+='.+'/g)
// What you wanted: [ "class='foo'", "id='bar'" ]
// What you got: [ "class='foo' id='bar'" ]
```

The `+` is greedy: keeps going until repeated token stops.

`.` is the token, which matches everything...

it matches the "f" in "foo", then "o" and "o"

then matches the single quote after "foo" and keeps going!

GREEDINESS AND LAZINESS

We can make the `+` lazy!

```
"<p class='foo' id='bar'>".match(/[a-z]+='.+?'/g)
// There it is: [ "class='foo'", "id='bar'" ]
```

However, less efficient due to backtracking required...
instead, we can use negation:

```
"<p class='foo' id='bar'>".match(/[a-z]+='[^']+'/g)
// Same result: [ "class='foo'", "id='bar'" ]
```

ESCAPING CHARACTERS

If you need to use the special characters as literals, you need to escape them with a backslash: `\`

```
"567.99".match(/[0-9]+.[0-9]{2}/)    // ["567.99"]
"567a99".match(/[0-9]+.[0-9]{2}/)    // ["567a99"]
"567.99".match(/[0-9]+\.[0-9]{2}/)   // ["567.99"]
"567a99".match(/[0-9]+\.[0-9]{2}/)   // null
```

ANCHORS

`^` and `$`

```
/[0-9]+\.[0-9]{2}/.test("576.99")       // true
/[0-9]+\.[0-9]{2}/.test("f00.00bar")    // true
/^[0-9]+\.[0-9]{2}$/.test("f00.00bar")  // false
/^[0-9]+\.[0-9]{2}$/.test("576.99")     // true
```

GROUPING, EFFICIENCY, AND SHORTHANDS

GROUPING AND ALTERNATION

`()` and `|`

```
"1234 Main St.".match(/[0-9]+ [a-z]+ [a-z\.]+/i)
// ["1234 Main St."]

"1234 Main St.".match(/([0-9]+) ([a-z]+) ([a-z\.]+)/i)
// ["1234", "Main", "St."]

"1234 Main St.".match(/([0-9]+) ([a-z]+) (st|rd|ave)\.?/i)
// ["1234", "Main", "St"]
```

Your language *may* include the entire matched string:

```
"1234 Main St.".match(/([0-9]+) ([a-z]+) (st|rd|ave)\.?/i)
// ["1234 Main St.", "1234", "Main", "St"]
```

GROUPING EFFICIENCY

Caution: matched groups add processing time & memory!

Use non-matching groups when possible: (?:)

```
"1234 Main St.".match(/([0-9]+) ([a-z]+) (st|rd|ave)\.?/i)
// [ "1234", "Main", "St" ]

"1234 Main St.".match(/[0-9]+ ([a-z]+) (st|rd|ave)\.?/i)
// [ "Main", "St" ]

"1234 Main St.".match(/[0-9]+ ([a-z]+ (st|rd|ave)\.?)/i)
// [ "Main St.", "St" ]

"1234 Main St.".match(/[0-9]+ ([a-z]+ (?:st|rd|ave)\.?)/i)
// [ "Main St." ]
```

MATCH REPLACEMENT

Caution! Varies widely across languages!

```
"I like {{fav}} the most".replace(/{{fav}}/, "dogs")

// "I like dogs the most"

const email = "mary@example.com"
email.replace(/([a-z]+)@([a-z.]+)/i, "https://$2/users/$1")

// "https://example.com/users/mary"
```

WHITESPACE & SHORTHANDS

Whitespace:

Space: " "

Tabs: `\t`

Newline: `\n`

Carriage return: `\r`

Shorthands:

Whitespace: `\s`

Digits: `\d` (same as `[0-9]`)

Word characters: `\w`
(generally: `[A-Za-z0-9_]`)

You can *negate* these shorthands using a capital letter:
`\S`, `\D`, `\W`
or by using a negation character class: `[^\s]`

SHORTHAND EXAMPLE

```
const users = [
    "jakerella      Jordan 42",
    "ro_ro      Alex    999",
    "Zee3ii Nick 13"
]
```

```
users.forEach((user) => {
    console.log(  user.match(/(\w+)\s+(\w+)\s+(\d+)/)   )
})
// ["jakerella", "Jordan", "42"]
// ["ro_ro", "Alex", "999"]
// ["Zee3ii", "Nick", "13"]
```

ANCHORS & MULTILINE

`^` and `$`

```
jakerella       Jordan 42
ro_ro   Alex     999
Zee3ii Nick 13
```

What if we want all of the usernames?

```
file.match(/\w+/)    // [ "jakerella" ]

file.match(/\w+/g)   // [ "jakerella", "Jordan", "42", "ro_ro", ... ]

file.match(/^\w+/g) // [ "jakerella" ]
```

THE MULTILINE FLAG

Use the global **and** multiline flags: `g` & `m`

```
jakerella      Jordan 42
ro_ro   Alex    999
Zee3ii Nick 13
```

```
file.match(/^\w+/gm)  // [ "jakerella", "ro_ro", "Zee3ii" ]
```

EFFICIENCY WARNINGS

Watch out for:

- Too many groups (use non-matching)
- Large source texts (use anchors)
- Unstructured data (like HTML)

HTML WARNING

You can't parse [X]HTML with regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions so many times before, the use of regex will not allow you to consume HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. It is too late it is too late we cannot be saved the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) dear lord help us how can anyone survive this scourge using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes using regex as a tool to process HTML establishes a breach between this world and the dread realm of corrupt entities (like SGML entities, but more corrupt) a mere glimpse of the world of regex parsers for HTML will instantly transport a programmer's consciousness into a world of ceaseless screaming, he comes, the pestilent slithy regex-infection will devour your HTML parser, application and existence for all time like Visual Basic only worse he comes he comes do not fight he comes, his unholy radiance destroying all enlightenment, HTML tags leaking from your eyes like liquid pain, the song of regular expression parsing will extinguish the voices of mortal man from the sphere I can see it can you see it it is beautiful the final snuffing of the lies of Man ALL IS LOST ALL IS LOST the pony he comes he comes he comes the ichor permeates all MY FACE MY FACE oh god no NO NOOO O NO stop the angles are not real ZALGO IS TONY THE PONY HE COMES

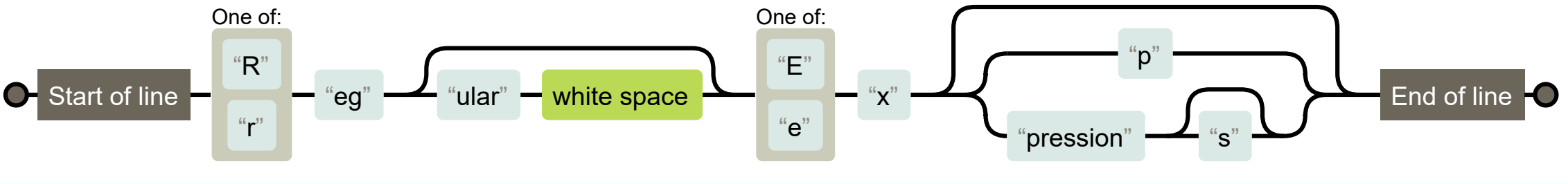
https://stackoverflow.com/a/1732454/1985406

RESOURCES

- Playground: regex101.com
- Various Engine Info: Wikipedia
- Reference Docs: regular-expressions.info
- Diagrams: regexper.com
- Dead Tree Version: "Mastering Regular Expressions"
- Fun: regexcrossword.com

THANK YOU!

/^[Rr]eg(ular\s)?[Ee]x(p|pressions?)?$/



Jordan Kasper | GH | Bsky | @jakerella

jordankasper.com/regex

Session Feedback:

BONUS CONTENT!

LOOK AROUNDS

They are **not** matches, but **assertions**.

Not supported in all engines / languages!

LOOK AHEADS

Assert that the group exists ahead of the match: (?=)

```
"The fat cat sat on the mat".match(/(the)(?=\sfat)/ig)
                                // ["The"]
```

Assert that the group doesn't exist ahead of the match: (?!)

```
"The fat cat sat on the mat".match(/(the)(?!\sfat)/ig)
                                // ["the"]
```

LOOK BEHINDS

Assert that the group exists behind the match: (?<=)

```
"The fat cat sat on the mat".match(/(?<=the\s)([a-z]at)/ig)
                                   // ["fat", "mat"]
```

Assert that the group doesn't exist behind the match: (?<!)

```
"The fat cat sat on the mat".match(/(?<!the\s)([a-z]at)/ig)
                                   // ["cat", "sat"]
```

© Jordan Kasper