Js-RegExp-Cheat-Sheet 1.0.0

Basic definitions

- String `s` matches the regex pattern `/p/` whenever `s` contains the pattern 'p'.
- Example: `abc` matches `/a/`, `/b/`, `/c/`
- For simplicity, we will use the matches verb loosely in a sense that
- a string can match a regex (e.g. 'a' matches /a/)
- a regex can match a string (e.g. /a/ matches 'a')
- A regex pattern consists of literal characters that match itself, and metasyntax characters - Literal characters can be concatenated in a regular expression. String `s` matches `/ab/` if there is an `a` character directly followed by a `b` character.
- Example: *`abc`* matches *`/ab/`*, *`/bc/`*, *`/abc/`*
- Example: *`abc`* does not match *`/ac/`*, *`/cd/`*, *`/abcd/`*
- *Alternative execution can be achieved with the metasyntax character `|`
- `/a|b/` means: match either `a` or `b`
- Example: 'ac', 'b', 'ab' match `/a|b/`
- Example: 'c' does not match `/a|b/`
- Iteration is achieved using repeat modifiers. One repeat modifier is the `*` (asterisk) metasyntax character.
- Example: `/a*/` matches any string containing any number of `a` characters
- Example: `/a*/` matches any string, including `''`, because they all contain at least zero `a` characters - Matching is greedy. A greedy match attempts to stay in iterations as long as possible.
- Example: `s = 'baaa'` matches `/a*a/` in the following way:
- `s[0]`: `'b'` is discarded
- `s[1]`: `'a'` matches the pattern `a*`
- `s[1] - s[2]`: `'aa'` matches the pattern `a*`
- `s[1] - s[3]`: `'aaa'` matches the pattern `a*`
- as there are no more characters in `s` and there is a character yet to be matched in the regex, we backtrack one character
- `s[1] - s[2]`: `'aa'` matches the pattern `a*`, and we end investigating the `a*` pattern - `s[3]`: `'a'` matches the `a` pattern
- there is a complete match, `s[1] - s[2]` match the `a*` pattern, and `s[3]` matches the `a` pattern. The returned match is `aaa` starting at index `1` of string `s`
- Backtracking is minimal. We attempt to backtrack one character at a time in the string, and attempt to interpret the rest of the regex pattern on the remainder of the string.

Constructing a regex

- literal form: `/regex/`
- constructor: `new RegExp( 'regex' );`
  - escaping: `/\d/` becomes `new RegExp( '\\d' )`
  - argument list: `new RegExp( pattern, modifiers );`

- `RegExp` constructor also accepts a regular expression:

```
new RegExp( /regex/, 'ig' ); // equivalent to:
/regex/ig
```

List of JavaScript regex modifiers

Modifier	Description
`i`	non-case sensitive matching. Upper and lower cases don’t matter.
`g`	global match. We attempt to find all matches instead of just returning the first match. The internal state of the regular expression stores where the last match was located, and matching is resumed where it was left in the previous match.
`m`	multiline match. It treats the `^` and `$` characters to match the beginning and the end of each line of the tested string. A newline character is determined by `\n` or `\r`.
`u`	unicode search. The regex pattern is treated as a unicode sequence
`y`	Sticky search

Example:

```
> const str = 'Regular Expressions';

> /re/.test( str ); false
> /re/i.test( str ); // matches: 're', 'Re', 'rE', 'RE' true
```

Regex API

- `regex.exec( str )`: returns information on the first match. Exec allows iteration on the regex for all matches if the `g` modifier is set for the regex - `regex.test( str )`: true iff regex matches a string

```
> const regex = /ab/;
> const str = 'bbababb';
> const noMatchStr = 'c';

> regex.exec( str ); // first match is at index 2
[ 0: "ab", index: 2, input: "bbababb" ]
> regex.exec( noMatchStr ); null

> regex.test( str ); true
> regex.test( noMatchStr ); false

> regex.exec( noMatchStr );

> const globalRegex = /ab/g;
> globalRegex.exec( str );
> globalRegex.exec( str );
[ 0: "ab", index: 2, input: "bbababb" ] >
globalRegex.exec( str );
```

```
[ 0: "ab", index: 4, input: "bbababb" ]
> globalRegex.exec( str ); null

> let result;
> while ((result = globalRegex.exec(str)) !== null) { console.log(result); }
[ 0: "ab", index: 2, input: "bbababb" ]
[ 0: "ab", index: 4, input: "bbababb" ]
```

String API

- `str.match( regex )`: for non-global regular expression arguments, it returns the same return value as `regex.exec( str )`. For global regular expressions, the return value is an array containing all the matches. Returns `null` if no match has been found.
- `str.replace( regex, newString )`: replaces the first full match with `newString`. If `regex` has a global modifier, `str.replace( regex, newString )` replaces all matches with `newString`. Does not mutate the original string `str`.
- `str.search( regex )`: returns the index of the first match. Returns `-1` when no match is found. Does not consider the global modifier.
- `str.split( regex )`: does not consider the global modifier. Returns an array of strings containing strings in-between matches.

```
> const regex = /ab/;
> const str = 'bbababb';
> const noMatchStr = 'c';

> str.match( regex );
["ab", index: 2, input: "bbababb"]

> str.match( globalRegex );
["ab", "ab"]

> noMatchStr.match( globalRegex ); null

> str.replace( regex, 'X' ); "bbXabb"

> str.replace( globalRegex, 'X' ) "bbXXb"

> str.search( regex );
2
> noMatchStr.search( regex ); -1

> str.split( regex );
["bb", "", "b"]

> noMatchStr.split( regex ); ["c"]
```

Literal characters

A regex literal character matches itself. The expression `/a/` matches any string that contains the `a` character. Example:

```
/a/.test( 'Andrea' )// true, because the last character is 'a'
```

```
/a/.test( 'André' )      // false, because there is no 'a' in the string
/a/i.test( 'André')      // true: 'A' matches /a/i
```

Literal characters are: all characters except metasyntax characters such as: ., *, ^, $, [,], (,), {, }, |, ?, +, \

When you need a metasyntax character, place a backslash in front of it. Examples: \., \\, \[.

Whitespaces:

- behave as literal characters, exact match is required - use character classes for more flexibility, such as:
- `\n` for a newline
- `\t` for a tab
- `\b` for word boundaries

Metasyntax characters

Metasyntax character	Semantics
.	arbitrary character class
[]	character sets, [012] means 0, or 1, or 2
^	(1) negation, e.g. in a character set [^890] means not 8, not 9, and not 10, (2) anchor matching the start of the string or line
$	anchor matching the end of the string or line
|	alternative execution (or)
*	iteration: match any number of times
?	optional parts in the expression
+	match at least once
{} and {,}	specify a range for the number of times an expression has to match the string. Forms: {3} exactly 3 times, {3,} at least 3 times, {3,5} between 3 and 5 times.
()	(1) overriding precedence through grouping, (2) extracting substrings
\	(1) before a metasyntax character: the next character becomes a literal character (e.g. \\). (2) before a special character: the sequence is interpreted as a special character sequence (e.g. \d as digit).
(?:,)	non-capturing parentheses. Anything in-between (?: and)is matched, but not captured. Should be used to achieve only functionality (1) of () parentheses.
(?=,)	lookahead. E.g. .(?=end) matches an arbitrary character if it is followed by the characters end. Only the character is returned in the match, end is excluded.
(?!,)	negative lookahead. E.g. .(?!end) matches an arbitrary character if it is not followed by the characters end. Only the character is returned in the match, end is excluded.
\b	word boundary. Zero length assertion. Matches the start or end position of a word. E.g. \bworld\b matches the string 'the world'
[\b]	matches a backspace character. This is not a character set including a word boundary.
\B	not a word boundary.
\c	\c is followed by character x. \cx matches CTRL + x.
\d	digit. [0-9]
\D	non-digit. [^0-9]
\f	form feed character
\n	newline character (line feed)
\r	carriage return character
\s	one arbitrary white space character
\S	one non-whitespace character

\t	tab character
\u	\u followed by four hexadecimal digits matches a unicode character described by those four digits when the u flag is not set. When the u flag is set, use the format \u{0abc}.
\v	vertical tab character
\w	alphanumeric character. [A-Za-z0-9_]
\W	non-alphanumeric character
\x	\x followed by two hexadecimal digits matches the character described by those two digits.
\0	NULL character. Equivalent to \x00 and \u0000. When the u flag is set for the regex, it is equivalent to \u{0000}.
\1, \2, …	backreference. \i is a reference to the matched contents of the ith capture group.

Examples:

```
/.../.test( 'ab' )      // false, we need at least three arbitrary characters
/.a*/.test( 'a' )       // true, . matches 'a', and a* matches ''
/^a/.test( 'ba' )       // false, 'ba' does not start with a
/^a/.test( 'ab' )       // true, 'ab' starts with a
/a$/.test( 'ba' )       // true, 'ba' ends with a
/a$/.test( 'ab' )       // false, 'ab' does not end with a
/^a$/.test( 'ab' )      // false, 'ab' does not fully match a
/^a*$/.test( 'aa' )     // true, 'aa' fully matches a pattern consisting of
                           // a characters only
/[ab]/.test( 'b' )         // true, 'b' contains a character that is a
                           // member of the character class `[ab]`
/a|b/.test( 'b' )          // true, 'b' contains a character that is
                           // either `a` or `b`
/ba?b/.test( 'bb' )        // true, the optional a is not included
/ba?b/.test( 'bab')        // true, the optional a is included
/ba?b/.test( 'bcb')        // false, only matches 'bb' or 'bab'
/a+/.test( '' )            // false, at least one `a` character is needed
/a+/.test( 'ba' )          // true, the `a` character was found
/a{3}/.test('baaab')       // true, three consecutive 'a' characters were found
/(a|b)c/.test('abc')       // true, a `b` character is followed by `c`
/a(?=b)/.test('ab')        // true. It matches 'a', because 'a' is followed by 'b'
/a(?!b)/.test('ab')        // false, because 'a' is not followed by 'b'
/\ba/.test('bab')          // false, because 'a' is not the first character of a word
/\Ba/.test('bab')          // true, because 'a' is not the first character of a word /(\d)\1/.test('55')
// true. It matches two consecutive digits with the same value
```

In the last example, notice the parentheses. As the `|` operator has the lowest precedence out of all operators, parentheses made it possible to increase the prcedence of `a|b` in `(a|b)c`.

Character sets, character classes

- `[abc]` is `a|b|c` - `[a-c]` is `[abc]`
- `[0-9a-fA-F]` is a case-insensitive hexadecimal digit
- `[^abc]` is an arbitrary character that is not `a`, not `b`, and not `c`
- `.`: arbitrary character class
- Example: `/..e/`: three character sequence ending with `e`
- other character classes such as digit (`\d`), not a digit (`\D`), word (`\w`), not a word (`\W`), whitespace character (`\s`): check out the section on metasyntax characters

Greedy and Lazy Repeat modifiers

Matching is maximal. Backtracking is minimal, goes character by character.

Greedy Repeat Modifier	Lazy Repeat Modifier	Description
`+`	`+?`	Match at least once
`?`	`??`	Match at most once
`*`	`*?`	Match any number of times
`{min,max}`	`{min,max}?`	Match at least min, and at most max times
`{n}`	`{n}?`	Match exactly n times

Examples:

- `/^a+$/` matches any string consisting of one or more `'a'` characters and nothing else
- `/^a?$/` matches `''` or `'a'`. The string may contain at most one `'a'` character
- `/^a*$/` matches the empty string and everything matched by `/^a+$/`
- `/^a{3,5}$/` matches `'aaa'`, `'aaaa'`, and `'aaaaa'`
- `/(ab){3}/` matches any string containing the substring `'ababab'`
- `/^a+?$/` lazily matches any string consisting of one or more `'a'` characters and nothing else
- `/^a??$/` lazily matches `''` or `'a'`. The string may contain at most one `'a'` character - `/^a*?$/` lazily matches the empty string and everything matched by `/^a+$/`
- `/^a{3,5}?$/` lazily matches `'aaa'`, `'aaaa'`, and `'aaaaa'`
- `/(ab){3}?/` lazily matches any string containing the substring `'ababab'`

Capture groups

- `(` and `)` captures a substring inside a regex
- Capture groups have a reference number equal to the order of the starting parentheses of the open parentheses of the capture group starting with `1`
- `(?:` and `)` act as non-capturing parentheses, they are not included in the capture group numbering Examples:

```
 /a(b|c(d|(e))(f))$/
   ^   ^  ^   ^
   |   |  |   |
   1   2  3   4
```

```
> console.table( /^a(b|c(d|(e))(f+))$/.exec( 'ab' ) )
(index) Value
0       "ab"
1       "b"
2       undefined
3       undefined 4 undefined index 0 input "ab"

> console.table( /^a(b|c(d|(e))(f+))$/.exec( 'aceff' ) )
(index) Value
```

```
0       "aceff"
1       "ceff"
2       "e"
3       "e" 4       "ff"
index 0 input "aceff"

> console.table( /^a(b|c(?:d|(e))(f+))$/.exec( 'aceff' ) )
(index) Value
0       "aceff"
1       "ceff"
2       "e" 3       "ff"
index 0 input "aceff"
```

Lookahead and Lookbehind

Lookahead type	JavaScript syntax	Remark
positive lookahead	`(?=pattern)`	
negative lookahead	`(?!pattern)`	
positive lookbehind	`(?<=pattern)`	in ES2018
negative lookbehind	`(?<!pattern)`	in ES2018
word boundary	`\b`	both lookahead & lookbehind
start of string/line	`^`	used as a lookbehind
end of string/line	`$`	used as a lookbehind

- Lookaheads and lookbehinds are non-capturing
- Opposed to Perl 5, lookbehinds can be of variable length, because matching is implemented backwards. This means no restrictions in lookbehind length
- as a consequence of backwards matching, using capture groups inside lookbehinds are evaluated according to the rules of backwards matching Examples:

```
> /a(?=b)/.exec( 'ab' )
["a", index: 0, input: "ab"]

> /a(?!\d)/.exec( 'ab' )
["a", index: 0, input: "ab"] >
/a(?!\d)/.exec( 'a0' ) null

> /(?<=a)b/.exec( 'ab' )
["b", index: 1, input: "ab"] // executed in latest Google Chrome

/(?<!a)b/.exec( 'Ab' )
["b", index: 1, input: "Ab"] // executed in latest Google Chrome
```

```
/\bregex\b/.exec( 'This regex expression tests word boundaries.' )
["regex", index: 5, input: "This regex expression tests word boundaries."] /^regex$/.exec(
'This\nregex\nexpression\ntests\nanchors.' ) null

/^regex$/m.exec( 'This\nregex\nexpression\ntests\nanchors.' )
["regex", index: 5, input: "Thisregexexpressiontestsanchors."]
```

Possessive Repeat Modifiers

Attempts a maximal (greedy) match first. The loop does not backtrack though. It either accepts the maximal match, or fails.

Possessive repeat modifiers don't exist in JavaScript. However, there are workarounds. Assuming we don't have any other capture groups in front of the expression, use

- `(?=(a+))\1` instead of the generic PCRE pattern `a++`
- `(?=(a*))\1` instead of the generic PCRE pattern `a*+` - etc.

Possessive Alternation

Attempts to match each branch of the alternation until the first match is found. After matching a branch in the alternation, we are not allowed to backtrack and try out any other branches in the alternation.

Possessive alternation does not exist in JavaScript. However, there are workarounds.

Assuming we don't have any other capture groups in front of the expression, use `(?=(a|b))\1` instead of the generic PCRE pattern `(?>a|b)`

Recommended RegExp library

XRegExp[1]

- Extended formatting
- Named captures
- Possessive loops
- etc.

Articles

Check out my articles on regular expressions on zsoltnagy.eu/category/regular-expressions[2]

[1] https://github.com/slevithan/xregexp
[2] http://zsoltnagy.eu/category/regular-expressions