Regular expression

In computing, a regular expression (abbreviated as *regexp* or *regex*, with plural forms *regexps*, *regexes*, or *regexen*) is a string that describes or matches a set of strings, according to certain syntax rules. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns. Many programming languages support regular expressions for string manipulation. For example, Perl and Tcl have a powerful regular expression engine built directly into their syntax. The set of utilities (including the editor ed and the filter grep) provided by Unix distributions were the first to popularize the concept of regular expressions.

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is known as globbing. Wildcards differ from regular expressions in that they can only express very restrictive forms of alternation.

Source: https://en.wikipedia.org/

Regular expressions explained

Regular expressions look ugly for novices, but really it's a very simple (well, usually simple!), easy to handle and a powerful tool.

Some examples

Real number (e.g.'13.88e-4', '-7E2'):

```
([+\-]?\d+(\.\d+)?([eE][+\-]?\d+)?)
```

Phone number (e.g. '+7(812) 555-5555', '(20)555-55-55', '555-5555'):

```
((\+\d *)?(\(\d{2,4}\) *)?\d{3}(-\d*)*)
```

Email address (e.g. 'anso@mail.ru', 'anso@mailbox.alkor.ru'):

```
([_a-zA-Z\d\-\.]+@[_a-zA-Z\d\-]+(\.[_a-zA-Z\d\-]+)+)
```

Internet URL (e.g. 'https://www.paycash.ru', 'ftp://195.5.138.172/default.html'):

```
([Ff][Tt][Pp]|[Hh][Tt][Tt][Pp]://([_a-zA-Z\d\-]+(\.[_a-zA-Z\d\-]+))((/[ _a-zA-Z\d\-\\\.]+)+)*
```

Detailed explanation

Any single character matches itself, unless it is a metacharacter with a special meaning described below.

A series of characters matches that series of characters in the target string, so the pattern bluh would match bluh in the target string. Quite simple eh ?

You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a \. For example, ^ match beginning of string, but \^ match character ^, \\ match \ and so on.

You can specify a character class, by enclosing a list of characters in [], which will match any one character from the list. If the first character after the [is ^, the class matches any character not in the list.

Within a list, the - character is used to specify a range, so that a-z represents all characters between a and z, inclusive. If you want - itself to be a member of a class, put it at the start or end of the list, or escape it with a backslash.

The following all specify the same class of three characters: az], [az, and [a\-z]. All are different from [a-z], which specifies a class containing twenty-six characters. If you want ']' you may place it at the start of list or escape it with a backslash.

Examples of queer ranges: [\n-\x0D] match any of #10,#11,#12,#13.

[\d-t] match any digit, '-' or 't'. []-a] match any char from ']'..'a'.

Characters may be specified using a metacharacter syntax much like that used in C: \n matches a newline, \t a tab, \r a carriage return, “\f” a form feed, etc. More generally, \xnn, where nn is a string of hexadecimal digits, matches the character whose ASCII value is nn.

Finally, the . metacharacter matches any character except \n (unless you use the /s modifier - see below). You can specify a series of alternatives for a pattern using | to separate them, so that fee|fie|foe will match any of fee, fie, or foe in the target string (as would f(e|i|o)e). The first alternative includes everything from the last pattern delimiter ((, [, or the beginning of the pattern) up to the first |, and the last alternative contains everything from the last | to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching foo|foot against barefoot, only the foo part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)

Also remember that | is interpreted as a literal within square brackets, so if you write [fee|fie|foe] you're really only matching [feio|].

The bracketing construct (...) may also be used for define r.e. subexpressions (after parsing you may find subexpression positions, lengths and actual values in MatchPos, MatchLen and Match properties of TRegExpr, and substitute it in template strings by TRegExpr.Substitute).

Subexpressions are numbered based on the left to right order of their opening parenthesis.

The first subexpression has the number '1' (whole r.e. match has number '0' - you may substitute it in RegExpr.Substitute as '$0' or '$&').

Any item of a regular expression may be followed with digits in curly brackets.

A short list of metacharacters

^	Start of line
$	End of line
.	Any character
\	Quote next character
*	Match zero or more
+	Match one or more
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match at least n but not more than m times
[aeiou0-9]	Match a, e, i, o, u, and 0 thru 9;
[^aeiou0-9]	Match anything but a, e, i, o, u, and 0 thru 9
\w	Matches an alphanumeric character (including _)
\W	A non alphanumeric
\d	Matches a numeric character
\D	A non-numeric
\s	Matches any space (same as [\t\n\r\f])
\S	A non space

You may use \w, \d and \s within character classes.

By default, the ^ character is only guaranteed to match at the beginning of the string, the $ character only at the end (or before the new line at the end) and perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by ^ or $.

You may, however, wish to treat a string as a multi-line buffer, such that the ^ will match after any newline within the string, and $ will match before any newline. At the cost of a little more overhead, you can do this by using the m modifier on the pattern match operator.

To facilitate multi-line substitutions, the . character never matches a new line unless you use the s modifier, which in effect tells TRegExpr to pretend the string is a single line - even if it isn't.

List of modifiers (Note: only “i”, “s” and “r” implemented)

- i Do case-insensitive pattern matching (using installed in your system local settings).
- s Treat string as single line. That is, change . to match any character whatsoever, even a new line, which it normally would not match. The s modifier without m will force ^ to match only at the beginning of the string and $ to match only at the end (or just before a new line at the end) of the string. Together, as ms, they let the . match any character whatsoever, while yet allowing ^ and $ to match, respectively, just after and just before new lines within the string.
- r Non-standard modifier.

Perl extensions

(?imsxr-imsxr) You may use it into r.e. for modifying modifiers by the fly, for example, (?i)Saint-Petersburg - will match string 'Saint-petersburg' and 'Saint-Petersburg', but (?i)Saint-(?-i)Petersburg - will match only 'Saint-Petersburg'.

If this construction is inlined into a subexpression, then it effects only into this subexpression

(?i)(Saint-)?Petersburg - will match 'Saint-petersburg' and 'saint-petersburg' , but (?i)Saint-)?Petersburg - will match 'saint-Petersburg', but not 'saint-petersburg'. (?#text) - A comment. The text is ignored.

Source: © 1999 Andrey V. Sorokin, anso@mail.ru