Syntax of Regular Expressions

Regular Expressions are a widely-used method of specifying patterns of text to search for. Special **metacharacters** allow You to specify, for instance, that a particular string You are looking for occurs at the beginning or end of a line, or contains **n** recurrences of a certain character.

Simple matches

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.

You can cause characters that normally function as **metacharacters** or **escape sequences** to be interpreted literally by 'escaping' them by preceding them with a backslash "\", for instance: metacharacter "^" match beginning of string, but "\^" match character "^", "\\" match "\" and so on.

Examples:

`foobar`	*matchs string 'foobar'*
`\^FooBarPtr`	*matchs '^FooBarPtr'*

Escape sequences

Characters may be specified using a **escape sequences** syntax: "\n" matches a newline, "\t" a tab, etc. More generally, \xnn, where nn is a string of hexadecimal digits, matches the character whose ASCII value is nn. If You need wide (Unicode) character code, You can use '\x{nnnn}', where 'nnnn' - one or more hexadecimal digits.

`\xnn`	*char with hex code nn*
`\x{nnnn}`	*char with hex code nnnn (one byte for plain text and two bytes for Unicode)*
`\t`	*tab (HT/TAB), same as \x09*
`\n`	*newline (NL), same as \x0a*
`\r`	*car.return (CR), same as \x0d*
`\f`	*form feed (FF), same as \x0c*
`\a`	*alarm (bell) (BEL), same as \x07*
`\e`	*escape (ESC), same as \x1b*

Examples:

`foo\x20bar`	*matchs 'foo bar' (note space in the middle)*
`\tfoobar`	*matchs 'foobar' predefined by tab*

Character classes

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.

Examples:

`foob[aeiou]r`	*finds strings 'foobar', 'foober' etc. but not 'foobbr', 'foobcr' etc.*
`foob[^aeiou]r`	*find strings 'foobbr', 'foobcr' etc. but not 'foobar', 'foober' etc.*

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. If You want ']' you may place it at the start of list or escape it with a backslash.

Examples:

`[-az]`	*matchs 'a', 'z' and '-'*
`[az-]`	*matchs 'a', 'z' and '-'*
`[a\-z]`	*matchs 'a', 'z' and '-'*
`[a-z]`	*matchs all twenty six small characters from 'a' to 'z'*
`[\n-\x0D]`	*matchs any of #10,#11,#12,#13.*
`[\d-t]`	*matchs any digit, '-' or 't'.*
`[]-a]`	*matchs any char from ']'..'a'.*

Metacharacters

Metacharacters are special characters which are the essence of Regular Expressions. There are different types of metacharacters, described below.

Metacharacters - line separators

`^`	*start of line*
`$`	*end of line*
`\A`	*start of text*
`\Z`	*end of text*
`.`	*any character in line*

Examples:

`^foobar`	*matchs string 'foobar' only if it's at the beginning of line*
`foobar$`	*matchs string 'foobar' only if it's at the end of line*
`^foobar$`	*matchs string 'foobar' only if it's the only string in line*
`foob.r`	*matchs strings like 'foobar', 'foobbr', 'foob1r' and so on*

The "^" metacharacter by default is only guaranteed to match at the beginning of the input string/text, the "$" metacharacter only at the end. Embedded line separators will not be matched by "^" or "$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any line separator within the string, and "$" will match before any line separator. You can do this by switching On the modifier /m.

The \A and \Z are just like "^" and "$", except that they won't match multiple times when the modifier /m is used, while "^" and "$" will match at every internal line separator.

The "." metacharacter by default matches any character, but if You switch Off the modifier /s, then '.' won't match embedded line separators.

"^" is at the beginning of a input string, and, if modifier /m is On, also immediately following any occurrence of \x0D\x0A or \x0A or \x0D. Note that there is no empty line within the sequence \x0D\x0A.

"$" is at the end of a input string, and, if modifier /m is On, also immediately preceding any occurrence of \x0D\x0A or \x0A or \x0D. Note that there is no empty line within the sequence \x0D\x0A.

"." matchs any character, but if You switch Off modifier /s then "." doesn't match \x0D\x0A and \x0A and \x0D.

Note that "^.*$" (an empty line pattern) does not match the empty string within the sequence \x0D\x0A, but matchs the empty string within the sequence \x0A\x0D.

Metacharacters - predefined classes

`\w`	*an alphanumeric character (including "_")*
`\W`	*a nonalphanumeric*
`\d`	*a numeric character*
`\D`	*a non-numeric*
`\s`	*any space (same as [\t\n\r\f])*
`\S`	*a non space*

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

Examples:

`foob\dr`	*matchs strings like 'foob1r', "foob6r' and so on but not 'foobar', 'foobbr' and so on*
`foob[\w\s]r`	*matchs strings like 'foobar', 'foob r', 'foobbr' and so on but not 'foob1r', 'foob=r' and so on*

Metazeichen – wordlimit

`\b`	*finds a wordlimit*
`\B`	*finds everything but a wordlimit*

A wordlimit (\b) ist a place between two characters, which has a \w on one side and a \W on the other side or vice versa. \b denotes all characters oft the \w till the front of the first character of \W vice versa.

Metacharacters - iterators

Any item of a regular expression may be followed by another type of metacharacters - **iterators**. Using this metacharacters You can specify number of occurences of previous character, **metacharacter** or **subexpression**.

`*`	*zero or more ("greedy"), similar to {0,}*
`+`	*one or more ("greedy"), similar to {1,}*
`?`	*zero or one ("greedy"), similar to {0,1}*
`{n}`	*exactly n times ("greedy")*
`{n,}`	*at least n times ("greedy")*
`{n,m}`	*at least n but not more than m times ("greedy")*
`*?`	*zero or more ("non-greedy"), similar to {0,}?*
`+?`	*one or more ("non-greedy"), similar to {1,}?*
`??`	*zero or one ("non-greedy"), similar to {0,1}?*
`{n}?`	*exactly n times ("non-greedy")*
`{n,}?`	*at least n times ("non-greedy")*
`{n,m}?`	*at least n but not more than m times ("non-greedy")*

So, digits in curly brackets of the form {n,m}, specify the minimum number of times to match the item n and the maximum m. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. There is no limit to the size of n or m, but large numbers will chew up more memory and slow down r.e. execution.

If a curly bracket occurs in any other context, it is treated as a regular character.

Examples:

`foob.*r`	*matchs strings like 'foobar', 'foobalkjdflkj9r' and 'foobr'*
`foob.+r`	*matchs strings like 'foobar', 'foobalkjdflkj9r' but not 'foobr'*
`foob.?r`	*matchs strings like 'foobar', 'foobbr' and 'foobr' but not 'foobalkj9r'*
`fooba{2}r`	*matchs the string 'foobaar'*
`fooba{2,}r`	*matchs strings like 'foobaar', 'foobaaar', 'foobaaaar' etc.*
`fooba{2,3}r`	*matchs strings like 'foobaar', or 'foobaaar' but not 'foobaaaar'*

A little explanation about "greediness". "Greedy" takes as many as possible, "non-greedy" takes as few as possible. For example, 'b+' and 'b*' applied to string 'abbbbc' return 'bbbb', 'b+?' returns 'b', 'b*?'

returns empty string, 'b{2,3}?' returns 'bb', 'b{2,3}' returns 'bbb'.

You can switch all iterators into "non-greedy" mode (see the modifier /g).

Metacharacters - alternatives

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|].

Examples:

`foo(bar|foo)`	*matchs strings 'foobar' or 'foofoo'.*

Metacharacters - subexpressions

The bracketing construct (...) may also be used for define regular subexpressions.

Subexpressions are numbered based on the left to right order of their opening parenthesis.
First subexpression has number '1', whole regular expression match has number '0'

Examples:

`(foobar){8,10}`	*matchs strings which contain 8, 9 or 10 instances of the 'foobar'*
`foob([0-9]|a+)r`	*matchs 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar' etc.*

Metacharacters - backreferences

Metacharacters \1 through \9 are interpreted as backreferences. \<n> matches previously matched **subexpression** #<n>.

Examples:

`(.)\1+`	*matchs 'aaaa' and 'cc'.*
`(.+)\1+`	*also match 'abab' and '123123'*
`(['"]?)(\d+)\1`	*matchs '"13" (in double quotes), or '4' (in single quotes) or 77 (without quotes)*

Modifiers

Modifiers are for changing behaviour of regular search. Any of these modifiers may be embedded within the regular expression itself using the (?...) construct.

i
: Do case-insensitive pattern matching (using installed in you system locale settings), see also InvertCase.

m
: Treat string as multiple lines. That is, change "^" and "$" from matching at only the very start or end of the string to the start or end of any line anywhere within the string, see also Line separators.

s
: Treat string as single line. That is, change "." to match any character whatsoever, even a line

separators (see also Line separators), which it normally would not match.

g

Non standard modifier. Switching it Off You'll switch all following operators into non-greedy mode (by default this modifier is On). So, if modifier /g is Off then '+' works as '+?', '*' as '*?' and so on

x

Extend your pattern's legibility by permitting whitespace and comments (see explanation below).

The modifier /x itself needs a little more explanation. It tells to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The # character is also treated as a metacharacter introducing a comment, for example:

```
(
(abc) # comment 1
  |   # You can use spaces to format regular expression
(efg) # comment 2
)
```

This also means that if you want real whitespace or # characters in the pattern (outside a character class, where they are unaffected by /x), that you'll either have to escape them or encode them using octal or hex escapes. Taken together, these features go a long way towards making regular expressions text more readable.

Source

The regular expressions support has been realized with the help of the class TRegExpr from the programmer Andrey Sorokin.