Perl Compatible Regular Expressions in a Nutshell

Syntax

All features on this cheat sheet are available in Perl, however, the implementation in other environments may differ slightly. Also support for UTF-8 may vary, e.g. Ruby supports UTF-8 in regex only since Ruby 1.9.

Perl:
```
$string =~ m/pattern/modifier
$string =~ s/pattern/replace/flags
```
http://www.perl.com/doc/manual/html/pod/perlre.html

Ruby:
```
string.match(/pattern/flags)
string.sub(/pattern/flags, replace)
string.gsub(/pattern/flags, replace)
```
http://www.apidock.com/ruby/String

PHP:
```
preg_match('/pattern/flags', $string)
preg_replace('/pattern/flags', 'replace', $string)
```
http://www.php.net/manual/en/ref.pcre.php

JavaScript:
```
string.match(/pattern/flags)
string.replace(/pattern/flags, "replace")
```
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Objects:RegExp

Java:
```
string.matches("(?flags)pattern")
string.replaceFirst("(?flags)pattern", "replace")
string.replaceAl("(?flags)pattern", "replace")
```
http://java.sun.com/javase/6/docs/api/java/util/regex/package-summary.html

C: (requires PCRE library)
```
pcrecpp::RE("(?flags)pattern").PartialMatch(string)
pcrecpp::RE("(?flags)pattern").Replace("replace", &string)
```
http://www.pcre.org

Objective-C: (requires RegexKit framework)
```
[string isMatchedByRexec:@"(?flags)pattern"]
[string stringByMatching:@"(?flags)pattern" replace:1 withString:@"replace"]
[string stringByMatching:@"(?flags)pattern" replace:RKReplaceAll withString:@"replace"]
```
http://regexkit.sourceforge.net

Shell:
```
grep -P "(?flags)pattern" file.txt
```
http://www.gnu.org/software/grep/

Characters

These are the usual suspects well known from any C-ish language:

`a`	match the character a
`3`	match the number 3
`$a` or `#{a}`	match the contents of a variable `$a` or `a` respectively
`\n`	newline (NL, LF)
`\r`	return (CR)
`\f`	form feed (FF)
`\t`	tab (TAB)
`\x3C`	character with the hex code 3C
`\u561A`	character with the hex code 561A
`\e`	escape character (alias `\u001B`)
`\c`...	control character

Wildcards

Wildcards match if a character belongs to the designated class of characters:

`.`	match any character
`\`...	quote single metacharacter: `\.` matches a dot instead of any character and `\\` matches a single backslash
`\w`	alphanumeric + underscore (shortcut for `[0-9a-zA-Z_]`)
`\W`	any character not covered by `\w`
`\d`	numeric (shortcut for `[0-9]`)
`\D`	any character not covered by `\d`
`\s`	whitespace (shortcut for `[ \t\n\r\f]`)
`\S`	any character not covered by `\s`
`[…]`	any character listed: `[a5!d-g]` means `a`, `5`, `!` and `d`, `e`, `f`, `g`
`[^…]`	any character not listed: `[^a5!d-g]` means anything but `a`, `5`, `!` and `d`, `e`, `f`, `g`

Boundaries

Boundaries match the spots between characters and therefore have no width of their own (also called zero-width, � extensions):

`\b`	matches at a word boundary (spot between `\w` and `\W`)
`\B`	matches anything but a word boundary
`^`	matches at the beginning of a line (`m`) or entire string (`s`)
`\A`	matches at the beginning of the entire string

$	matches at the end of a line (**m**) or entire string (**s**)
\Z	matches at the end of the entire string ignoring a tailing **\n**
\z	matches at the end of the entire string
\G	matches where the previous regex call left off (➜ flag **g**)

Grouping

Any of the above constructs can be grouped in order to improve readability and/or create a reference that can be used in *pattern* or *replace* (➜ replacing):

(…)	the matching pattern is assigned to the reference **\1** and **$1** (outside of the regex context)
(…) (…) *etc*	first group is **\1** and **$1**, second group is **\1** and **$2** *etc*
(…|…|…)	matches if one of the group options matches and assignes it to **\1** and **$1**
\1, **\2** *etc*	include the contents of the corresponding group

Extensions

Less common functionality is covered by extensions using the **(?…)** syntax. Extensions do <u>not</u> create a reference like grouping does.

(?:…|…)	same as grouping, but no reference is created
(?=…)	zero-width positive lookahead assertion
(?!…)	zero-width negative lookahead assertion
(?<=…)	zero-width positive lookbehind assertion (no quantifiers allowed within)
(?<!…)	zero-width negative lookbehind assertion (no quantifiers allowed within)
(?>…)	zero-width independend subexpression
(?(…)…|…)	conditional expression
(?_flags_)	apply the flag(s) within the current group from this point forward (➜ flags)
(?_flags_:…)	apply the flag(s) for this pattern (no backreference created!)
(?#…)	zero-width comment (no round brackets allowed in comment text)

Quantifiers

Most of the above constructs may be quantified by adding one of the following symbols after them:

?	match 1 or 0 times
*	0 or more times
+	1 or more times
{_n_}	exactly *n* times
{_n_,}	at least *n* times
{_n_,_m_}	at least *n* but not more than *m* times, as often as possible

Greedyness

This is a very important feature, ignore it and you will be destined to produce clumsy and error prone regex! Quantifiers are greedy by default which means they match as often as possible. Limit their hunger by adding a **?** after them. Here's an example applied to the title of this section:

G.*e	matches to **Greedyne**
G.*?e	matches to **Gre**

Quoting

You want to ignore all of the above for a while? Here you go:

\…	quote single metacharacter: **\.** matches a dot instead of any character
\Q … **\E**	ignore all regex metacharacters in between

Replacing

The following symbols have special meanings in the *replace* part:

\1, **\2** *etc*	include the contents of the corresponding group (➜ grouping)
\{1}000	same as the above, use curly brackets if numbers follow the symbol
\l	lowercase the following character
\L … **\E**	lowercase all characters in between
\u	uppercase the following character
\U … **\E**	uppercase all characters in between

Flags

Optional flags determine the behaviour of the regex as a whole. May be used within the **(?_flags_)** construct (➜ extensions):

i	case-insensitive pattern matching
m	multiple lines: **.** does not match **\n** (Ruby uses this per default)
s	single line: **.** matches **\n** (Ruby uses **m** for this instead)
x	ignore whitespaces in pattern for better readability

The following <u>cannot</u> be used within the **(?_flags_)** construct:

g	apply the regex as many times as possible (i.e. for global replace)
e	evaluate the *replace* part as if it were source code !! DANGER !!
o	compile the pattern only once and therefore perform variable substitutions only once