Regular Expressions Basics

re = *regular expression*.
greedy = always favour matching
lazy = always favour not matching

Any character

`.` ..... (dot) Any character, except new line
`\.` ..... Literal dot

Character classes (custom)

`[...]` ..... Any character listed
`[^...]` ..... Any character *not* listed
`[a-z]` ..... Range, any character from a to z
`[-...]` ..... To include dash (-), add it first

Boundaries

`^, $` ..... [Start, End] of line
`\A, \Z, \z` ..... [Start, End, End] of string
The difference between these occurs in multi-line mode.

`\b` ..... Word boundary (Perl, Java, .Net)
`\B` ..... Not word boundary
`\<,\>` ..... [Start, End] of word† (Vim)
`\G` ... Match starts where the previous match ended
`\Q`*text*`\E` ..... Literal *text* inside regex

† Perl/Java/.Net:
`\<` → `(?<!\w)(?=\w)`
`\>` → `(?<=\w)(?!\w)`
(see *Lookaround* in next page).

Grouping and Backreferences

`(`*re*`)` ..... Group *re* as a unit (Perl, Java, .Net)
`\(`*re*`\)` ..... Vim without *very magic*
`\1,...,\9` ..... $[1..9]$-th matched `()`

Substitutions

`$1,...,$9` ..... $[1..9]$-th matched `()`
`$+` ..... Last matched `()`
`$&, &` ..... Entire matched string (Perl, Vim)
`` $`, $' `` ..... Text [before, after] match
`$_` ..... Entire input text

Quantifiers (greedy)

(Perl/Java/.Net, Vim without *very magic*):
`re?, re\?` ..... *re* optional (0 or 1)
`re*` ..... Any quantity of *re* (0 or more)
`re+, re\+` ..... At least one of *re* (1 or more)
`re{n}, re\{n}` ..... *re* exactly n times
`re{n,}, re\{n,}` ..... *re* at least n times
`re{n,m}, re\{n,m}` .... *re* at least n and at most m times

Quantifiers (lazy)

(Perl/Java/.Net, Vim without *very magic*):
`re??, re\{-,1}` ..... *re* optional (0 or 1)
`re*?, re\{-}` ..... Any quantity of *re* (0 or more)
`re+?, re\{-1,}` ..... At least one of *re* (1 or more)
`re{n}?, re\{-n}` ..... *re* exactly n times
`re{n,}?, re\{-n,}` ..... *re* at least n times
`re{n,m}?, re\{-n,m}` .. *re* between n and m times

Alternation (branching)

`re|re` ..... *re* or *re* (Perl/Java/.Net)
`re\|re` ..... *re* or *re* (Vim without *very magic*)
Note: In NFA engines alternatives are tried in order of appearance. So placing them in probability order prevents backtracking, improving efficiency.

Shorthands

`\s` ..... White space (space, tab, etc.)†
`\S` ..... Not `\s`
`\d` ..... Digit (`[0-9]`)†
`\D` ..... Not `\d`
`\w` ..... Word char (`[a-zA-Z0-9_]`)†
`\W` ..... Not `\w`
`\t, \n` ..... Tab, New line
† Some flavours also recognise special Unicode characters in the same character group.
In Vim `\_s` also includes new line.

Modes and Flags

`i` ..... Ignore case† (Perl/Vim)
`m` ..... Multi-line mode (Perl)
`s` ..... Dot matches new line‡ (Perl)
`x` ..... Ignore white space§ (Perl)
`g` .... Apply substitution in all occurrences (Perl/Vim)

† Java: `Pattern.CASE_INSENSITIVE`,
.Net: `RegexOptions.IgnoreCase`.
‡ In Vim, use `\_.` to make dot match new lines.
§ Java: `Pattern.COMMENTS`,
.Net: `RegexOptions.IgnorePatternWhiteSpace`.

Regular Expressions Advanced

Lookaround

Lookahead = Check if *re* matches at current position, but do not consume.
Lookbehind = Check if *re* matches just before what follows, but do not consume.
Negative = Check if *re* does *not* match.

(Perl/Java/.Net, Vim):
`(?=re)`, `re\@=` ..... Lookahead
`(?!re)`, `re\@!` ..... Negative lookahead
`(?<=re)`, `re\@<=` ..... Lookbehind
`(?<!re)`, `re\@<!` ..... Negative lookbehind

Non-capturing parentheses and comments

Group as a unit, but do not capture for backreferencing.
`(?:re)` ..... Perl, Java, .Net
`\%(re\)` ..... Vim

`(?#free text)` ..... Comment inside regex

Branch reset

`(?|re)` . Restart group numbering for each branch in *re*

Not available in Vim.

Named captures

`(?<name>re)` ..... Capture *re* under *name*
`\k<name>` ..... Named backreference
`$+{name}` ..... Named substitution (Perl)
`${name}` ..... Named substitution (.Net)

Not available in Vim.

Atomic grouping

Match *re* without retry. In a NFA engine, discard all the possible backtracking states for the enclosed *re*.

`(?>re)` ..... Perl/Java/.Net
`re\@>` ..... Vim

Possessive quantifiers

`re?+`	`(?>re?)`
`re*+`	`(?>re*)`
`re++`	`(?>re+)`
`re{n,m}+`	`(?>re{n,m})`

Not available in Vim (but can be done with atomic grouping).

End of previous match

`\G` ..... Matches where the previous match ended
Useful when using the `g` mode.
In Perl, use `pos($str)` to know the current location where `\G` would match in `$str`.
Not available in Vim.

Conditional expressions

If *cond* then *re*1, else *re*2. *re*2 is optional.

Perl/Java/.Net:
`(?(cond)re)` ..... Without *else re*
`(?(cond)re1|re2)` ..... With *else re*
`(?(n)…)` ..... *cond* = n-th `()` matched

cond can be a *lookaround* expression (see the examples).
Not available in Vim.

Dynamic regex and embedded code

`(??{code})` ..... Match *re* built by *code* here
`(?{code})` ..... *code* that does anything
Useful for debugging regex by `print`ing.

Not available in Vim.

Recursive expressions

`(?R)` ..... Repeat entire *re* here
`(?Rn)` ..... Repeat *re* captured under group n here

Not available in Vim.

Mode modifiers

(Perl/Java/.Net, Vim):
`(?i)`, `\c` ..... Turn case-insensitive on
`(?-i)`, `\C` ..... Turn case-insensitive off
`(?i:re)` ..... Be case insensitive for *re*

Regular Expressions and Characters

Unicode properties

Perl/Java/.Net:

`\p{L}` ..... Letters
`\p{M}` ..... Accent marks, etc.
`\p{Z}` ..... Spaces, etc.
`\p{S}` ..... Dingbats and symbols
`\p{N}` ..... Numeric characters
`\p{P}` ..... Punctuation characters
`\p{C}` ..... Everything else

Unicode sub-properties

`\p{Ll}` ..... Lower-case letters
`\p{Lu}` ..... Upper-case letters
`\p{Mn}` ..... Non-spacing mark (accents, ...)
`\p{Sm}` ..... Math symbol (−,+,÷,≤,...)
`\p{Sc}` ..... Currency symbol (¥,¢,€,$,£,...)
`\p{Nd}` ..... Decimal digit
`\p{Nl}` .... Letter number (mostly Roman numerals)
`\p{Pd}` ..... Dash punctuation
`\p{Ps}` ..... Open punctuation ([, {, ⟪, ...)
`\p{Pe}` ..... Close punctuation (], }, ⟫, ...)
`\p{Cc}` .. ASCII and Latin-1 control characters (`TAB`, `CR`, `LF`, ...)

Unicode negated properties

`\P{⋯}` ..... Perl/Java/.Net
`\p{^⋯}` ..... Perl

Unicode blocks and scripts

`\p{InCyrillic}` ..... Cyrillic character (Perl, Java)
`\p{IsCyrillic}` ..... Cyrillic character (.Net)
`\p{Latin}` ..... Latin character
`\p{Greek}` ..... Greek character
`\p{Hebrew}` ..... Hebrew character

POSIX character classes

`[:alnum:]` ..... Alphanumeric characters
`[:alpha:]` ..... Alphabetic characters
`[:blank:]` ..... Space and tab
`[:cntrl:]` ..... Control characters
`[:digit:]` ..... Numeric characters
`[:graph:]` ..... Non-blank characters
`[:lower:]` ..... Lower-case alphabetic characters
`[:print:]` ... `[:graph:]` and the space character
`[:punct:]` ..... Punctuation characters
`[:space:]` ..... All whitespace characters
`[:upper:]` ..... Upper-case alphabetic characters
`[:xdigit:]` .... Hexadecimal digits (`[0-9a-fA-F]`)

POSIX collating sequences

`[.span-ll.]` ..... Match “ll” as single character
`[.ch.]` ..... Match “ch” as single character
`[.eszet.]` ..... Match “ss” (“ß”) as single character

POSIX character equivalents

`[[=a=][=n=]]` Character *equivalents* for *a* and *n* (a, á, à, ä, ..., n, ñ, ...)

Special characters and other shorthands

`\`*`char`* ..... Literal *char*
`\t` ..... Tab (`HT, TAB`)
`\n` ..... New line (`LF, NL`)
`\r` ..... Carriage return (`CR`)
`\f` ..... Form feed (`FF`)
`\a` ..... Alarm (`BEL`)
`\e` ..... Escape (think troff) (`ESC`)
`$$` ..... Literal $ (.Net)
`\l` ..... Lower-case next char
`\u` ..... Upper-case next char
`\L`*`text`*`\E` ..... Lower-case *text*
`\U`*`text`*`\E` ..... Upper-case *text*
`\`*`num`* ..... Octal escape
`\x`*`num`*`, \x{`*`num`*`}` ..... Hex escape
`\u`*`num`*`, \U`*`num`* ..... Unicode escape
`\c`*`x`*`, \c`*`X`* ..... `CTRL-`*`X`*
`\N{`*`U+263D`*`}` ..... Unicode character
`\N{`*`name`*`}` ..... Character by Unicode *name*

Class set operations

`[[a-z]&&[^aeiou]]` .. Class 1 except class 2 (Java)
`[[a-z]-[aeiou]]` ..... Class 1 except class 2 (.Net)

Regular Expressions Efficiency

Notation: $worse \rightarrow better$

General guidelines

- Avoid recompiling the *re*.
- Use non-capturing parentheses (but check they are indeed faster in your case!).
- Split into multiple *re*'s (or literal text search) if it can make the process faster.
- Use leading anchors (`^`, `\A`, `$`, `\Z`, etc.) to reduce the number of locations where the *re* is evaluated.
- Use atomic grouping and possessive quantifiers to avoid unnecessary backtracking (especially important in non-matching cases).
- Put the most-likely alternatives first (only affects traditional NFA engines).

Lazy quantifiers may be slower than greedy

Because they must jump between checking what the quantifier controls with checking what comes after. Example: `"(.*?)"` to match a double-quoted string. Compare with the example in the next page.

Expose anchors

`^this|^that` → `^(?:this|that)`.

`abc$|123$` → `(?:abc|123)$` (only affects Perl).

re only tried at places where the anchor (i.e. `^`) matches.

Algebraic identities

Note: In this section all parenthesis are non-capturing. These are strictly theoretical equivalences.

`r|s` = `s|r` ..... commutativity for alternation†

`r|(s|t)` = `(r|s)|t` ..... associativity for alternation

`r|r` = `r` ..... absorption of alternation

`r(st)` = `(rs)t` ..... associativity for concatenarion

`r(s|t)` = `rs|rt` ..... left distributivity

`(s|t)r` = `sr|tr` ..... right distributivity

`rϵ` = `r` ..... identity for concatenation

`r*r*` = `r*` ..... closure absorption

`r*` = `ϵ|r|rr|...` ..... Kleene closure

`rr*` = `r*r` ..... `r+`

`(r*)*` = `r*`

`(r*|s*)*` = `(r*s*)*`

`(r*s*)*` = `(r|s)*`

`(rs)*r` = `r(sr)*`

`(r|s)*` = `(r*s)*r*`

† It is not the same in terms of performance for a NFA engine, where alternatives are tried in order of appearance.

Expose literal text

`this|that` → `th(?:is|at)`.

`-{5,7}` → `----{0,2}`.

The engine can use a fast substring search (like the Boyer-Moore or the Hume-Sunday algorithms) to find the literal text (i.e. `th`) and then match the rest of the *re*.

Loop unrolling

Useful to optimise a regex with the following structure: `(normal|special)*`. This can be changed to: `opening normal*(?:special normal*)* closing`, where `opening`, `normal`, `special` and `closing` are all regular expressions. It can be used to optimise any number of alternatives, like in `(re1|re2|...)*`.

Notes:

1. `normal` should be the most common case
2. the start of `normal` and `special` must not match at the same location.
3. `special` must not match the *empty string*.
4. `special` must be atomic.

See the examples.

Regular Expressions Examples 1

Match quoted string with escaped quotes

```
"([^\\"]++|\\.)*+"
```

Note: Possessive quantifiers prevent very long execution times (due to the nested quantifiers `+` and `*`) when there is no matching.
Loop-unrolling version:

```
"[^\\"]*+(?:\\.[^\\"]*+)*+"
```

Conditional expressions

- Backreference as condition: Match a word optionally wrapped in <···>:
  `(<)?\w+(?(1)>)`
- Lookaround as condition: Check for number only if prefixed with "`NUM:`":
  `(?(?<=NUM:)\d+|\w+)`

Branch reset

Always capture the number under `\1` (or `$1`):

```
(?|Num:(\d+)|Number:(\d+)|N=(\d+))
```

Fix floating-point rounding problems

Use at most 3 decimal places. Change numbers like 3.27600000002828 into 3.276; or 4.120000000034 into 4.12:

```
s/(\.\d\d[1-9]?+)\d+/$1/g
```

Extract file name from path

Perl:

```
$path =~ m{([^/]*)$};
$file = $1;
```

Adding thousand separators to a number

Using lookaround:

```
s/(?<=\d)(?=(\d\d\d)+(?!\d))/,/g
```

Without using lookaround (in Perl):

```
while ($text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g)
{
  # Just repeat until no match
}
```

Match continuation line

Match a single "logical" line split into multiple lines by adding `\` at the end of each split. Example:

```
SRC=a.c b.c c.c \
    d.c e.c
```

```
^\w+=([^\n\\]|\\.)*
```

Loop-unrolling version:

```
^\w+=            # Leading field and '='
(                # Capture complete line
  (?> [^\n\\]* ) # normal
  (?>
    \\.          # special
    [^\n\\]*     # normal
  )*
)
```

Match a date (month and day)

```
(?|
  (Jan|Mar|May|Jul|Aug|Oct|Dec)
  (31|[123]0|[012]?[1-9])
  |
  (Apr|Jun|Sep|Nov)
  ([123]0|[012]?[1-9])
  |
  Feb
  ([12]0|[012]?[1-9])
)
```

Match time (am/pm)

```
(1[012]|[1-9]):[0-5][0-9] (am|pm)
```

Match time (24 hours)

```
[01]?[0-9]|2[0-3]
```

or

```
[01]?[4-9]|[012]?[0-3]
```

Regular Expressions Examples 2

Parse a CSV file

This regex matches each field in a CSV line, supporting fields with or without surrounding double-quotes. In the former case, a double quote is represented by a pair of double quotes:

```
(?:^|,)\s*
(?|
    "
    ( (?: [^"] | "" )* ) # Slow
    "
  |
    ( [^",]* )
)
```

The line with the "slow" comment can be made faster using loop unrolling, with `normal = [^"]` and `special = ""`:

```
( (?> [^"]* ) (?> "" [^"]* )* )
```

Match IP address

Using Perl regex object for clarity:

```
$num = qr/[01]?\d\d?|2[0-4]\d|25[0-5]/;
  (?<!\d)
  ($num)\.($num)\.($num)\.($num)
  (?!\d)
```

Match an email address

```
\b(
  \w[-.\w]* (?# user name)
  \@
  [-a-z0-9]+(\.[-a-z0-9]+)*
  \.(com|org|net|gov|edu|info)
  (\.[a-z]{2})?
)\b
```

Match a URL

```
\<(
  (?:https?|ftp)://
  (?i: [a-z0-9] (?: [-a-z0-9]* [a-z0-9] )? \.)+
  (?|
    ((?-i: com|org|net|gov|edu|info)
     (?-i: [a-z]{2})?)
    |
    ((?-i: [a-z]{2}))
  )
  (:\d+)? (?# port number)
  \b
  (
    /
    [-a-z0-9_:@&?=+,.!/~*'%$]*
    (?<![.,?!]) (?# Not allowed at end)
  )?
)\>
```

Match closing XML tag

Using lazy quantifier: `<B>((?!<B>).)*?</B>`
Using greedy quantifier: `<B>((?!</?B>).)*</B>`
Loop-unrolling with `normal = [^<]` and `special = (?!</?B>)<`:

```
<B>                 # opening
    (?>[^<]*)       # any "normal"
    (?>             # any amount of
        (?!</?B>)   # if not <B> or </B>
        <           # one "special"
        [^<]*       # any normal
    )*
</B>                # closing
```

Balanced parentheses (static)

Match up to n levels of nested parentheses:

```
# Using Perl regex objects
my $level0 = qr/\(([^()]        )*\)/x;
my $level1 = qr/\(([^()]|$level0)*\)/;
my $level2 = qr/\(([^()]|$level1)*\)/;
...
my $balpar = qr/\(([^()]|$levelN)*\)/;

# Using string to build regex
my $balpar = "\(([^()])*\)"; # level 0
foreach (1..$n) {
  $balpar = "\(([^()]|$balpar)*\)";
}
```

Balanced parentheses (dynamic)

Match a function call with balanced parentheses:

```
my $balpar; # must be predefined
$balpar = qr/
  (?>
    [^()]+
    |
    \( (??{ $balpar }) \)
  )*
/x;

if ($text =~ m/\b(\w+)(\($balpar\))/) {
  print "function: $1, args: $2\n";
}
if (not $text =~ m/^ $balpar $/x) {
  print "mismatched parentheses\n";
}
```

Regular Expressions Usage in Programming Languages and Tools

Perl

```perl
# Highlight double words
# perl -w finddbl file.txt

# Chunk with dot-newline combination
$/ = ".\n"

while (<>) { # Put input "line" in $_
  next unless s{
    \b
    ( \w+ ) (?# grab word in $1 and \1 )
    (?# Any number of spaces or <TAGS> )
    (
      (?:
        \s
        |
        <[^>]+>
      )+
    )
    (\1\b) (?# match the first word again)
  }
  {\e[7m$1\e[m$2\e[7m$3\e[m}igx;

  # Remove unmarked lines
  s/^(?:[^\e]*\n)+//mg;

  # Insert file name
  s/^/$ARGV: /mg;

  # Print $_
  print;
}
```

Java

```java
import java.util.regex.*;

String text;

// Extract subject
Pattern re = Pattern.compile(
    "^Subject: (.*)",
    Pattern.CASE_INSENSITIVE);
Matcher m = re.matcher(text);

if (m.find()) {
  subject = m.group(1);
}

// Insert prefix
Pattern re = Pattern.compile("^(.*)$");
re.matcher(text).replaceAll(">> $1");
```

Python

```python
import re

r = re.compile("^Subject: (.*)",
               re.IGNORECASE)
m = r.search(text);
if m:
    subject = m.group(1)
```

.Net (C#)

```csharp
using System.Text.RegularExpressions;

string text;

// Extract subject
Regex re = new Regex(
    "^Subject: (.*)",
    RegexOptions.IgnoreCase);
Match m = re.Match(text);

if (m.Success) {
  subject = m.Groups(1).Value;
}

// Insert prefix
Regex re = new Regex("^(.*)$");
re.Replace(text, ">> ${1}");
```

Automated editing

```
sed -i .old -E 's/re/.../g' file
perl -p -i .old -e 's/re/.../g' file
```