

Regular Expressions

Dr. Leigh Brookshaw

18th April 2006

Abstract

Regular expressions are used for advanced context sensitive searches in text files. They are used in many advanced editors to search for and replace strings (such as `vi` and `emacs`), in parser programs and are available in many programming languages.

The UNIX system defines two standards of regular expressions – “Basic Regular Expressions” (BRE) and “Extended Regular Expressions” (ERE). Though “Basic Regular Expressions” are considered obsolete many popular UNIX utility programs (such as `grep`, `sed` and `ed`) use “Basic Regular Expressions” by default. To add to the confusion many scripting languages (such as Perl, Python and PHP) define their own regular expression syntax. Fortunately all of them follow the same principles and the differences are usually the addition of greater functionality. Once you understand the basic idea, it is easy to learn the details of the different dialects.

This document makes no attempt at completeness — rather it covers the basics of UNIX regular expression syntax that is the basis for the many RE variants

After reading through this document and attempting all the exercises you should be able to construct your own regular expressions as needed.

Contents

1.1	Introduction	3
1.2	Characters	6
1.3	Simple Strings	6
1.4	BRE Meta-characters	9
1.4.1	Period	9
1.4.2	Square Brackets	11
1.4.3	Asterisk	15
1.4.4	Caret and Dollar	17
1.4.5	Quoting Special Characters	19
1.5	ERE Meta-characters	21
1.5.1	Plus	21

1.5.2	Question Mark	22
1.5.3	Braces	24
1.5.4	Parentheses	27
1.5.5	Or	28
1.5.6	Back References	30
1.6	Replacement String	31
1.6.1	Delimiters	32
1.6.2	Ampersand	33
1.6.3	Quoted Digits	34
1.7	Further Reading and References	35

1.1 Introduction

A regular expression describes a pattern or a sequence of characters. An *expression* is not something to be interpreted literally. It is something that needs to be evaluated. A *regular expression* is also an expression that has to be evaluated and it has to be evaluated one character at a time!

Programs that accept regular expressions¹ must first evaluate the syntax of the regular expression (the regular expression is in effect compiled) to produce a pattern. The pattern is then compared against a string (for example a line of text). To see if the string contains a matching substring the first character of the pattern is compared to the first character of the string. If there is a match the second character of the pattern is compared to the second character of the string. If the current pattern character fails to match the current string character then the matching starts again at the start of the pattern but now starting one character along in the string. Example 1.1 illustrates this procedure by trying to match the pattern `ege` on a string.

The power of regular expressions comes from the fact that patterns are not restricted to literal characters. Regular expressions define a set of meta-characters. Each meta-character has a special meaning in a regular expression, that is, it can represent characters other than itself. Unfortunately, some of the the meta-characters used in regular expressions are meaningful to the shell (the wild-card characters `*` and `?` for instance) this can be rather confusing. It is important to remember a regular expression is a feature of the command that uses the expression and has nothing to do with the shell. This means that all regular expressions should be quoted when being passed to a command on the command line. This will stop the shell from trying to interpret the characters in the regular expression.

¹Any program can make use of regular expressions by using the POSIX Regular Expression functions available in the UNIX system. See `man regex`.

Note: Regular expressions are interpreted by the command not by the shell. Quoting the regular expression ensures that the shell isn't able to interpret the meta-characters in its own way. Always quote regular expressions!

Example 1.1: Interpreting a regular expression.

Consider the following input line (the `_` character represents a space):

```
This_is_a_test_of_regexprs.
```

and the following pattern:

```
ege
```

Patterns are matched against strings a character at a time starting at the left most character in the following manner —

```
This_is
ege
```

1. In this example there is no match between the first character of the input line and the first character of the pattern. Since there is no match, the next line of the input line is compared to the first character of the pattern.

```
This_is
ege
```

2. There is no match between the second character of the input line and the first character of the pattern. Continue.

```
_test_
ege
```

3. The first match between a character on the input line and the first character of the pattern occurs in the word `test`. Since this is a match the second character of the pattern is compared to the next character in the input line.

```
test  
ege
```

4. The second character in the pattern does not match the next character in the input line.

```
test  
ege
```

5. Return to the first character of the pattern and start the comparison again with the next character in the input line.

```
regexps.  
ege
```

6. The next match of the first character of the pattern occurs in the word `regexps.`

```
regexps.  
ege
```

7. Since there is a match the second character in the pattern is compared to the next character in the input line.

```
regexps.  
egex
```

8. Now the third and final character in the pattern is compared to the next character of the input line. This is also a match. So this input line contains a substring that successfully matches the pattern.

1.2 Characters

As used in this exposition of regular expressions, a *character* is any character *except* a *newline* character. Most characters represent themselves within a regular expression. A meta-character is one that does not represent itself. If you need a meta-character to represent itself see the section on “Quoting Special Characters” (§1.4.5)

1.3 Simple Strings

The most basic regular expression is a simple string that contains no meta-characters. A simple string matches only itself.

Example 1.2:

Pattern: `ring`

Meaning: matches the substring `ring`

Examples: ring, spring, string, gringo, hearing

Pattern: `or_not`

Meaning: matches the substring `or_not`

Examples: or not, poor nothing, poor note

Pattern: `toll`

Meaning: matches the substring `toll`

Examples: atoll, extoll, ayatollah, tollgate

Pattern: `sword`

Meaning: matches the substring `sword`

Examples: crossword, swordfish, password

Pattern: `tour`

Meaning: matches the substring `tour`

Examples: contour, tourniquet, tourist, detour, entourage

Exercise 1.3: Finding patterns with egrep

The UNIX command `egrep`² is a good tool for experimenting with regular expressions. Egrep searches an input file or files (or standard input if no files are given) for lines containing a substring that match the given pattern. By default, egrep prints the lines that contain the matching substring.

²Egrep understands EREs while the older program `grep` only understands BREs. On some UNIX systems only `grep` is available.

UNIX systems normally store a list of dictionary words in the file `/usr/share/dict/words`³. (Some UNIX systems may use the directory `/usr/dict`.) Write commands of the form

```
egrep 'pattern' dictionary-file
```

to locate words in the dictionary. For example, the command

```
egrep 'fred' /usr/share/dict/words
```

will find all the words that contain the substring “fred”.

Using `egrep` and the `words` file find—

- All words containing the substring “aea”.
- All words containing the substring “eea”.
- All words containing the substring “yz”.
- All words containing the substring “kyr”.
- All words containing the substring “rtb”.

Exercise 1.4: The command `egrep` has many options that will change its default behaviour. Read the UNIX man page for `egrep` and answer the following questions:

- `egrep` by default, is case sensitive. What flag will make `egrep` case insensitive?

³If this file does not exist on your system under Debian then you should use `aptitude` or `synaptic` to install the package `wbritish`.

- `egrep`'s default action is to output the line of the file that contains a matching substring. What is the flag that will change the output to the number of lines that contain a matching substring?
 - `egrep` will output the lines that contain a substring that matches the pattern. What is the flag that will change this default behaviour so that `egrep` will output only lines that do not match?
 - Which option to `grep` will allow it to interpret EREs?
-

1.4 BRE Meta-characters

Meta-characters make regular expressions powerful and useful. They also add to the complexity of regular expressions. When meta-characters are used within regular expressions the resulting pattern will be able to match more than one string of characters.

A regular expression that includes meta-characters always attempts to match the longest possible string of characters as close as possible to the start of the input line.

1.4.1 Period

A period will match any character.

Example 1.5:

Pattern: `_.alk`

Meaning: matches all substrings that start with a space followed by any character followed by `alk`

Examples: will talk,
may balk

Pattern: `.zin.`

Meaning: matches all substrings with a character before and after `zin`

Examples: sizing, gazing, Mazzini, breeziness

Pattern: `Z..s`

Meaning: matches all substrings with two characters between the characters `Z` and `s`

Examples: Zeus, Zens

Exercise 1.6: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `ing.`
 - `z.z`
 - `.z..z`
 - `.q.u.`
-

1.4.2 Square Brackets

Square brackets, `[. . .]` define a set of characters that matches any single character within the square brackets. If the first character following the left square bracket is a caret `^`, the square brackets define a set of characters that matches any single character **not** within the brackets.

A hyphen can be used to indicate a range of characters. If the hyphen is the first or last character within the square brackets it no longer represents a range but its own character value

Within the square brackets the meta-characters backslash, asterisk and the dollar sign (all described below) lose their special meaning. A caret is only a meta-character within the square brackets when it is the first character after the left square bracket. The right square bracket can appear as part of the character set only if it is the first character after the left square bracket.

Example 1.7:

Pattern: `[wW]ord`

Meaning: match all substrings `Word` and `word`.

Examples: `Wordsworth`, `foreword`, `swordfish`

Pattern: `t[aeiou].k`

Meaning: matches all substrings that start with `t` followed by a vowel, any character, and a `k`

Examples: `undertook`, `ticks`, `stockholder`, `cantankerous`

Pattern: `Chapter_[5-9]`

Meaning: matches all substrings beginning with `Chapter`, followed by a space then a digit in the range 5 to 9.

Examples: Chapter 7, Chapter 96, Chapter 30

Pattern: `[^a-zA-Z]`

Meaning: matches any character that is not a letter

Examples: \$100, Chapter 1, myaddress@myisp.com

Pattern: `[Qq][^aeiou]`

Meaning: matches all substrings that start with Q or q followed by a character that is not a vowel.

Examples: Compaq's, Iqbal, Iraq's

Pattern: `[_-]`

Meaning: matches all substrings that contain an underline or a dash.

Examples: meta-character, variable_name

Exercise 1.8: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `.'s`
 - `[W-Y]....'s`
 - `[A-Z][^a-z]`
 - `[Kk][^aeiou']`
-

Character Classes

Square brackets, `[...]` define a set of characters or a “Character Class” that matches any single character within the square brackets. In order to accommodate non-English environments, the square brackets have been enhanced with the ability to match characters not in the English alphabet.

Generic character classes consist of a keyword bracketed by `[: and :]`. The keywords describe different classes of characters such as alphabet characters, control characters etc.

Table 1.1 lists the standard generic character classes⁴. The characters that the generic character classes define are locale dependent. For example, in an English speaking locale the following regular expressions are equivalent:

`[[:alpha:]]` is equivalent to `[A-Za-z]`

But in a Russian speaking environment they would not be equivalent as `[[:alpha:]]` would also include the characters `к ж Я Ж`

Example 1.9:

- Pattern:** `[[:space:]][[:upper:]]`
- Meaning:** match all substrings that start with a whitespace character followed by one uppercase character,
- Examples:** `_`This is a match, The word Melbourne is a match
- Pattern:** `0x[[:xdigit:]][[:xdigit:]]`
- Meaning:** match all byte length hexa-decimal substrings
- Examples:** `0xff, 0x10, 0x2c, 0xdd`

⁴A locale can provide other character classes

Table 1.1: Standard character classes

Class	Matching Characters
<code>[:alnum:]</code>	Printable characters (includes whitespace)
<code>[:alpha:]</code>	Alphabetic characters
<code>[:blank:]</code>	Space and tab characters
<code>[:cntrl:]</code>	Control characters
<code>[:digit:]</code>	Numeric characters
<code>[:graph:]</code>	Printable and visible (non-space) characters
<code>[:lower:]</code>	lowercase characters
<code>[:print:]</code>	printable characters (includes whitespace)
<code>[:punct:]</code>	Punctuation characters
<code>[:space:]</code>	Whitespace characters
<code>[:upper:]</code>	Uppercase characters
<code>[:xdigit:]</code>	Hexadecimal digits

Exercise 1.10: Put into words the meaning of the following patterns. Give examples.

Test your examples by piping your test string to `grep`—for example:

```
echo 'TESTSTRING' | egrep 'PATTERN'
```

if the `TESTSTRING` is returned then the pattern matched a substring in the test string.

Provide alternate strings that have their character classes replaced with character ranges.

- `[[:upper:]][[:punct:]]`

- `[[:punct:]]`
 - `[[:digit:]][[:digit:]][:./][[:digit:]][[:digit:]]`
 - `[[:punct:]][[:blank:]][[:upper:]]`
-

1.4.3 Asterisk

An asterisk represents *zero or more* occurrences of the regular expression pattern that precedes it. The regular expression pattern can include any meta-characters.

Example 1.11:

Pattern: `ab*c`

Meaning: matches the substring starting with `a` followed by zero or more `b`s followed by `c`

Examples: `ac`, `abc`, `abbc`, `abbbbbc`

Pattern: `t.*ing`

Meaning: matches the substring starting with `t` followed by zero or more characters followed by `ing`

Examples: `thing`, `testing`, `thought of going`

Pattern: `[a-zA-Z]*`

Meaning: matches any substring composed of letters.

Examples: any substring without punctuation or numbers

Pattern: `_[A-Z][a-z]*_`

Meaning: matches any substring that begins with a space followed by a capital letter, followed by zero or more lowercase characters, ending in a space.

Examples: First word of a sentence, Bill , Kevin

Exercise 1.12: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `[[[:upper:]]].*[[[:upper:]]]`
- `[aeiou][aeiou]*'s`
- `k[aeiou]*k`
- `..*wall..*`

Longest Match Possible

A regular expression always matches the longest possible substring starting as close to the start of the input string as possible. For example, consider the following string

This rug is not what it once was, is it?

The regular expression `Th.*is` matches the substring

This rug is not what it once was, is

The regular expression `is.*n` matches the substring

is rug is not what it on
however the pattern `is[^rw]*n` matches the substring
is n

Exercise 1.13: The option `-o` modifies the output of the `grep` command so that only the matching substring is output.

Using the following command:

```
echo 'This is a long string for the exercise' | egrep -o 'PATTERN'
```

What is the `PATTERN` that will produce the following output:

- (a) `string for the exercise`
 - (b) `ng strin`
 - (c) `This`
 - (d) `_a_long_string_for_`
 - (e) `is_is`
-

1.4.4 Caret and Dollar

There are two meta-characters that allow you to specify the context in which a matching substring appears, either at the beginning of a line or at the end of a line. The caret `^` meta-character is a single-character regular expression indicating the beginning of the input line. The dollar sign `$` meta-character is a single character regular expression indicating the end of the input line. These are often referred to as “anchors” since they anchor, or restrict, the match to a specific location.

Example 1.14:

- Pattern:** `^T`
Meaning: Matches the T at the beginning of a line.
Examples: This is the ...
That time ...
- Pattern:** `^[0-9]`
Meaning: Matches a number at the beginning of a line.
Examples: 600, the number of ...
147 frogs found in the ...
- Pattern:** `^[0-9][0-9]*$`
Meaning: Matches any line that contains only digits.
Examples: 1
1234567
0050123456
- Pattern:** `^[:;]$`
Meaning: Matches any line that ends in a colon or a semi-colon.
Examples: See below:
Consider the following list;
-

Exercise 1.15: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `^[A-Z]..*'s$`
 - `^[Qq][^u]...$`
 - `^Fred`
 - `fred$`
-

1.4.5 Quoting Special Characters

Most meta-characters can be made to represent themselves by *quoting* them. To quote a meta-character the character is preceded by a backslash. Quoting a normal character has no effect.

Example 1.16:

Pattern: `[A-Z]..*\.`

Meaning: Match all substrings that start with a capital letter followed by one or more characters and end in a period.

Examples: Mrs., A sentence will also match.

Pattern: `*\.\c`

Meaning: matches the substring `*.\c`

Examples: `*.c`, `test*.c`, `myscript*.c`

Pattern: `\[...*]`

Meaning: matches the substring surrounded by square brackets

Examples: `[5]`, `array[20]`, `[place answer here]`

- Pattern:** `\\`
Meaning: Match any backslash
Examples: A newline character is `\n`, The tab character is `\t`
- Pattern:** `\[[1-9][0-9.]\]`
Meaning: matches any substring with a number surrounded by square brackets. The number cannot start with zero or a point.
Examples: `[123.45678]`, `[98793]`, `[7.]`
-

Exercise 1.17: Put into words the meaning of the following patterns. Give examples. Test your examples by piping your test string to `grep`—for example:

```
echo 'TESTSTRING' | grep 'PATTERN'
```

if the `TESTSTRING` is returned then the pattern matched a substring in the test string.

- `^\[.**\]$`
 - `\\\[[:alpha:]\]`
 - `^[A-Z].*\. $`
 - `_[1-9][0-9]\.[0-9][0-9]_`
-

1.5 ERE Meta-characters

The regular expression patterns discussed so far are used in “Basic Regular Expressions”. The basic set of meta-characters has been extended to create “Extended Regular Expressions”.

1.5.1 Plus

A plus represents *one or more* occurrences of the regular expression pattern that precedes it. The regular expression pattern can include any meta-characters.

Example 1.18:

Pattern: `ab+c`

Meaning: matches the substring starting with a followed by one or more bs followed by c

Examples: abc, abbbc, abbbbbbbbbc

Pattern: `[aeiou]+z`

Meaning: matches one or more vowels followed by the character z

Examples: woozy, maze, embezzle, doze

Pattern: `z[a-y]+z`

Meaning: matches the substring beginning with and ending in z, with one or more letters in between but not z.

Examples: Velázquez, Gonzalez, Dzerzhinsky, zigzag

Pattern: `[Cc]hapter[_]+[0-9]+`

Meaning: Match the substring starting with the word `Chapter` or `chapter` followed by one or more spaces followed by one or more digits.

Examples: `Chapter_1`, `chapter_23`, `Chapter_05`

Exercise 1.19: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `.+wall.+`
 - `^[A-Z].+'s$`
 - `^[A-Z]+$`
 - `w[aeiou]+s`
-

1.5.2 Question Mark

A question mark represents *zero or one* occurrences of the regular expression pattern that precedes it. The regular expression pattern can include any meta-characters.

Example 1.20:

Pattern: `[+-]?[0-9]+\.[0-9]+`

Meaning: matches the substring starting with an optional plus or a minus, followed by one or more digits, followed by an optional point which can be followed by one or more digits. This means that if a point exists it must be surrounded by digits.

Examples: 12345, -12345, +4.7, 3.14159

Pattern: `y[aeiou]?z`

Meaning: matches the substring starting with `y` and ending in `z` with an optional vowel between them.

Examples: Byzantium, Kyrgyzstan, Soyuz

Pattern: `x.?y`

Meaning: matches the substring beginning with `x` and ending with `y`, with zero or one character in between.

Examples: oxygen, sixty, max young, laxly

Exercise 1.21: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `.+wall?.`
 - `^[Qq]u?`
 - `y[aeiou]?y`
 - `y[^aeiou]?y`
-

1.5.3 Braces

The meta-characters that allow you to specify repeated occurrences of a character ($*+?$) indicate a span of characters of undetermined length. The brace meta-characters allow you to indicate a span of characters of specified length.

The braces enclose one or two arguments:

$\{n, m\}$

n and m are integers between 0 and 255. If you specify

$\{n\}$

by itself, then exactly n occurrences of the preceding character or regular expression will be matched. If you specify

$\{n\},$

then at least n occurrences will be matched. If you specify

$\{n, m\}$

then any number of occurrences between n and m will be matches.

Note:

The meta-character “?” is equivalent to “ $\{0, 1\}$ ”

The meta-character “*” is equivalent to “ $\{0, \}$ ”

The meta-character “+” is equivalent to “ $\{1, \}$ ”

Example 1.22:

Pattern: $[0-9]\{4\}-[0-9]\{4\}-[0-9]\{4\}-[0-9]\{4\}$

Meaning: Match a credit card number. Four digits, a dash, four digits, a dash, four digits, a dash, four digits

Examples: 1234-0002-7638-1234

Pattern: `00[135][0-9]{6}`

Meaning: Match a student number starting with 00 followed by one of 1, 3, 5, followed by six digits.

Examples: 0051234567, 003001234, 001010101

Pattern: `4631-[0-9]{4}`

Meaning: Match a USQ phone number

Examples: 4631-1234, 4631-2222, 4631-7890

Pattern: `[aeiou]{4,}`

Meaning: Match any substring of four or more consecutive vowels.

Examples: queueing, sequoia, Hawaiian, pharmacopoeia, obsequious, onomatopoeia

Note: Braces can be used in BREs — but to turn the braces into meta-characters they must be preceded by a “\”!

That is, the equivalent BREs of the examples above, are:

- `[0-9]\{4\}-[0-9]\{4\}-[0-9]\{4\}-[0-9]\{4\}`
- `00[135][0-9]\{6\}`
- `4631-[0-9]\{4\}`
- `[aeiou]\{4,\}`

This use of a backslash (to create a meta-character) is the reverse of standard behaviour and therefore extremely confusing.

This extension to BREs should be avoided and wherever possible EREs should be used.

For portability many scripts found on the UNIX system will use BREs exclusively.

Exercise 1.23: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `[aeiou]{3,3}`
 - `[aeiou]r{2,}[aeiou]{2,}r`
 - `m.{12,}m`
 - `^[[:alpha:]]{20,}$`
-

1.5.4 Parentheses

Parentheses, (. . .), are used to group regular expressions. Meta-characters that specify repeated occurrences (for example, *, + etc.) can also be applied to the regular expressions enclosed in parentheses.

Example 1.24:

- Pattern:** `([0-9]{4}-){3}[0-9]{4}`
Meaning: Match a credit card number. Four digits, a dash, four digits, a dash, four digits, a dash, four digits. (cf. the example above)
Examples: 1234-0002-7638-1234
- Pattern:** `(et){2}`
Meaning: Match the substring etet
Examples: dietetic
- Pattern:** `[Ll]ab(oratorie)?s`
Meaning: Match the substrings Laboratories, laboratories, Labs, labs
Examples: Labs, Laboratories, labs, laboratories, Sandia National Laboratories
- Pattern:** `w([aeiou].)+y`
Meaning: Match any substring starting with w followed by one or more substrings of a vowel followed by a character, ending with the character y
Examples: Bowery, awesomely, midwifery, widely, wisely

Note: As with braces parentheses can be used in BREs as meta-characters — if they are quoted. As with braces this is not a practice to be encouraged. But if portability is an issue then BREs should be used.

Exercise 1.25: Put into words the meaning of the following patterns. Give examples and test your answer by using `egrep` and the `words` file.

- `(ak+)+`
 - `(m[aeiou]+[^m]){2,}`
 - `(am..){2,}`
 - `^[aeiou][^aeiou]){6,}$`
-

1.5.5 Or

The vertical bar (`|`) meta-character can be used to specify a union of regular expressions.

Example 1.26:

Pattern: `[Cc]ompan(y|ies)`

Meaning: Match the substrings `Company` or `company` or `Companies` or `companies`.

Examples: `Company`, `company`, `Companies`, `companies`

Pattern: `UNIX|Unix|unix`

Meaning: Matches the substrings UNIX or Unix or unix.

Examples: UNIX, Unix, unix

Pattern: `w.*(del|dil)+y`

Meaning: Match any substring starting with w, followed by zero or more characters, followed by del or dil one or more times, ending with a y.

Examples: bawdily, dowdily, widely

Pattern: `(^\.\.)*[A-Z][a-z]*\.`

Meaning: Match the first word of a line if it starts with a capital letter, or match the first word of a sentence.

This is an attempt to find the first word of all sentences — but there will be problems.

Examples: The first word of a sentence and the line.

Mr. Jones (forgot about this possibility)
...the end. A new sentence ...

Exercise 1.27: Put into words the meaning of the following patterns. Give examples and test your answer by using egrep and the words file.

- `c(ious|uous|eous)$`
- `(pug|gup)`
- `(Jones|Smith)`

- `(si|va).+(us|ac)`
-

1.5.6 Back References

Back references allows you to reference within the same regular expression any of the parenthesised subexpressions that have been defined. A backslash followed by a nonzero digit (a quoted digit `\n`) represents the substring that the n-th bracketed regular expression matched. Bracketed regular expressions are numbered from the left by the positions of their opening parentheses.

Example 1.28:

Pattern: `(ke)[a-z]+\1`

Meaning: Match substrings that begin with `ke` and end in `ke`, with at least one character between the two patterns.

Examples: keepsake, Okefenokee, Rickenbacker, keystroke

Pattern: `[aeiou]b[^aeiou]+\1`

Meaning: Match substrings that begin with a vowel followed by the character `bb`, followed by any alpha character not a vowel, followed by the original matching vowel and the character `b`

Examples: hobgoblin, hubbub, hobnob

Pattern: `(^|_+)([A-Za-z]+)_+\2(_|$)`

Meaning: Match substrings with repeated words anywhere on a line.

Examples: This is a a repeated word.
Even Even at the beginning of a line
Even at the end of a line line

Pattern: `(([0-9]{2})-\2)\1`

Meaning: Match the substrings with two digits, a dash, four digits (the same two digits repeated), a dash, and the same two digits.

Examples: 34-3434-34, 78-7878-78, 01-0101-01

Note: Due to the complexity of the logic of implementing back references and the possible ambiguities that can develop when interpreting them they are best avoided.

1.6 Replacement String

Regular expressions are used as search patterns within substitute commands in editors (`vi`, `emacs`, `sed`), in many scripting languages (Perl, Python, PHP, Javascript) and in programming languages (Java).

Note: The default action of All substitute commands is to substitute the first match that is found to the regular expression. This default action can be changed. How the default action is modified is implementation specific. In the examples below the default action is followed.

1.6.1 Delimiters

A character, called a *delimiter*, usually marks the beginning and end of a regular expression and the replacement string in a substitution command. The delimiter is always a meta-character for the regular expression it delimits (that is, it does not represent itself, but marks the beginning and end of the regular expression). Any character can be used as the delimiter of the regular expression. The delimiter most widely used is the forward slash.

Example 1.29: The following substitution commands could be used in `vi`.

Pattern: `/Smith/Jones/`

Meaning: Replace the substring `Smith` with the substring `Jones`.

Examples: `Mr. Smith` → `Mr. Jones`

Pattern: `/(UNIX|unix)/Unix/`

Meaning: Replace the substring `UNIX` or `unix` with `Unix`.

Examples: `The UNIX system` → `The Unix system`
`The unix system` → `The Unix system`

Pattern: `/_{2,}/_/`

Meaning: Replace two or more spaces with one space only.

Examples: `The__UNIX__system` → `The_ UNIX_ system`

Pattern: `/^[[:space:]]+//`

Meaning: Replace any white space characters at the beginning of a line with nothing! That is delete the white space at the start of a line. If

the line only contains white space this command will not delete the line.

Examples: `sed 's/ /_/' The UNIXsystem` → The UNIXsystem

1.6.2 Ampersand

Within a replacement string, an ampersand (&) takes on the value of the substring the regular expression pattern matched.

Example 1.30:

Pattern: `/[0-9]+/(&)/`

Meaning: Place parentheses around all integers of one or more digits

Examples: 100 → (100)
 ps2pdf → ps(2)pdf
 3.1415926 → (3).1415926 (maybe this is not what was wanted!)

Pattern: `/[0-9]*\.[0-9]+/(&)/`

Meaning: Place parentheses around all numbers of one or more digits

Examples: 0 → (0)
 ps2pdf → ps(2)pdf
 3.1415926 → (3.1415926)
 .7 → (.7)
 7. → (7). (Unexpected?)

1.6.3 Quoted Digits

As with back references (§1.5.6), a quoted digit in the replacement string represents the matching string for the equivalent bracketed regular expression. Bracketed regular expressions are numbered from the left by the positions of their opening parentheses.

Note: Unlike “back references”, quoted digits in the replacement string is far more useful and far less fraught with pitfalls.

Example 1.31:

- Pattern:** `/^_*([^_]+)_+([^_]+) (_ | $)_ / \2_ \1 /`
- Meaning:** Swap the first two words of a line. In the process remove any extra spaces.
- Examples:** First Word → word First
 Jones George → George Jones
 First word first → word Firstfirst (maybe this is not what was wanted!)
- Pattern:** `/([0-9]{2})\ / ([0-9]{2})\ / ([0-9]{4})\ / \2\ / \1\ / \3 /`
- Meaning:** reverse the month and day fields of a date
- Examples:** 02/12/2005 → 12/02/2005
 31/07/2006 → 07/31/2006
- Pattern:** `/([^ ,]+) , (.+)\ / \2 \1 /`
- Meaning:** Convert a substring of the form:
 Family-name, Given-name Initial
 into the form:

Given-name Initial Family-name

Examples: Smith, John → John Smith
Jones, Ingrid L. → Ingrid L. Jones

Note: Mixing “back references” with quoted digits in the replacement string is a recipe for disaster and only for the adventurous or foolhardy.

1.7 Further Reading and References

- A description of the POSIX Regexp standard can be found on the man page `regex(7)`.
- Any book on UNIX that incorporates a chapter on Regular Expressions.

©2005 Leigh Brookshaw
Department of Mathematics and Computing, USQ
(This file created: 18th April 2006)