18.1. A Brief Introduction to Regular Expressions

An expression is a string of characters. Those characters having an interpretation above and beyond their literal meaning are called *metacharacters*. A quote symbol, for example, may denote speech by a person, *ditto*, or a meta-meaning [1] for the symbols that follow. Regular Expressions are sets of characters and/or metacharacters that match (or specify) patterns.

A Regular Expression contains one or more of the following:

- *A character set*. These are the characters retaining their literal meaning. The simplest type of Regular Expression consists *only* of a character set, with no metacharacters.

- *An anchor*. These designate (*anchor*) the position in the line of text that the RE is to match. For example, ^, and $ are anchors.

- *Modifiers*. These expand or narrow (*modify*) the range of text the RE is to match. Modifiers include the asterisk, brackets, and the backslash.

The main uses for Regular Expressions (*RE*s) are text searches and string manipulation. An RE *matches* a single character or a set of characters -- a string or a part of a string.

- The asterisk -- * -- matches any number of repeats of the character string or RE preceding it, including *zero* instances.

  "1133*" matches `11 + one or more 3's`: *113*, *1133*, *1133333*, and so forth.

- The *dot* -- . -- matches any one character, except a newline. [2]

  "13." matches `13 + at least one of any character (including a space)`: *1133*, *11333*, but not *13* (additional character missing).

  See Example 16-18 for a demonstration of *dot single-character* matching.

- The caret -- ^ -- matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.

- The dollar sign -- $ -- at the end of an RE matches the end of a line.

  "XXX$" matches XXX at the end of a line.

  "^$" matches blank lines.

- Brackets -- [...] -- enclose a set of characters to match in a single RE.

  "[xyz]" matches any one of the characters *x*, *y*, or *z*.

  "[c-n]" matches any one of the characters in the range *c* to *n*.

  "[B-Pk-y]" matches any one of the characters in the ranges *B* to *P* and *k* to *y*.

  "[a-z0-9]" matches any single lowercase letter or any digit.

  "[^b-d]" matches any character *except* those in the range *b* to *d*. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to ! in a different context).

  Combined sequences of bracketed characters match common word patterns. "[Yy][Ee][Ss]" matches *yes*, *Yes*, *YES*, *yEs*, and so forth. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" matches any Social Security number.

- The backslash -- \ -- escapes a special character, which means that character gets interpreted literally (and is therefore no longer *special*).

  A "\$" reverts back to its literal meaning of "$", rather than its RE meaning of end-of-line. Likewise a "\\" has the literal meaning of "\".

- Escaped "angle brackets" -- \<...\> -- mark word boundaries.

  The angle brackets must be escaped, since otherwise they have only their literal character meaning.

  "\<the\>" matches the word "the," but not the words "them," "there," "other," etc.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
```

```
 This is line 3, another line.
 This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

The only way to be certain that a particular RE works is to test it.

```
TEST FILE: tstfile                              # No match.
                                                # No match.
Run   grep "1133*"  on this file.               # Match.
                                                # No match.
                                                # No match.
This line contains the number 113.              # Match.
This line contains the number 13.               # No match.
This line contains the number 133.              # No match.
This line contains the number 1133.             # Match.
This line contains the number 113312.           # Match.
This line contains the number 1112.             # No match.
This line contains the number 113312312.        # Match.
This line contains no numbers at all.           # No match.
```

```
bash$ grep "1133*" tstfile
Run   grep "1133*"  on this file.              # Match.
 This line contains the number 113.             # Match.
 This line contains the number 1133.            # Match.
 This line contains the number 113312.          # Match.
 This line contains the number 113312312.       # Match.
```

- **Extended REs.** Additional metacharacters added to the basic set. Used in egrep, awk, and Perl.

- The question mark -- ? -- matches zero or one of the previous RE. It is generally used for matching single characters.

- The plus -- + -- matches one or more of the previous RE. It serves a role similar to the *, but does *not* match zero occurrences.

```
# GNU versions of sed and awk can use "+",
# but it needs to be escaped.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# All of above are equivalent.

# Thanks, S.C.
```

- Escaped "curly brackets" -- \{ \} -- indicate the number of occurrences of a preceding RE to match.

  It is necessary to escape the curly brackets since they have only their literal character meaning otherwise. This usage is technically not part of the basic RE set.

  "[0-9]\{5\}" matches exactly five digits (characters in the range of 0 to 9).

  Curly brackets are not available as an RE in the "classic" (non-POSIX compliant) version of awk. However, the GNU extended version of *awk*, **gawk**, has the `--re-interval` option that permits them (without being escaped).

  ```
  bash$ echo 2222 | gawk --re-interval '/2{3}/'
  2222
  ```

  Perl and some **egrep** versions do not require escaping the curly brackets.

- Parentheses -- **( )** -- enclose a group of REs. They are useful with the following "|" operator and in substring extraction using expr.

- The -- | -- "or" RE operator matches any of a set of alternate characters.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

Some versions of **sed**, **ed**, and **ex** support escaped versions of the extended Regular Expressions described above, as do the GNU utilities.

- **POSIX Character Classes. `[:class:]`**

  This is an alternate method of specifying a range of characters to match.

- **`[:alnum:]`** matches alphabetic or numeric characters. This is equivalent to **`A-Za-z0-9`**.
- **`[:alpha:]`** matches alphabetic characters. This is equivalent to **`A-Za-z`**.
- **`[:blank:]`** matches a space or a tab.
- **`[:cntrl:]`** matches control characters.
- **`[:digit:]`** matches (decimal) digits. This is equivalent to **`0-9`**.
- **`[:graph:]`** (graphic printable characters). Matches characters in the range of ASCII 33 - 126. This is the same as **`[:print:]`**, below, but excluding the space character.
- **`[:lower:]`** matches lowercase alphabetic characters. This is equivalent to **`a-z`**.
- **`[:print:]`** (printable characters). Matches characters in the range of ASCII 32 - 126. This is the same as **`[:graph:]`**, above, but adding the space character.
- **`[:space:]`** matches whitespace characters (space and horizontal tab).
- **`[:upper:]`** matches uppercase alphabetic characters. This is equivalent to **`A-Z`**.
- **`[:xdigit:]`** matches hexadecimal digits. This is equivalent to **`0-9A-Fa-f`**.

  POSIX character classes generally require quoting or double brackets ([[ ]]).

```
bash$ grep [[:digit:]] test.file
abc=723
```

```
# ...
if [[ $arow =~ [[:digit:]] ]]   #  Numerical input?
then       #  POSIX char class
```

```
  if [[ $acol =~ [[:alpha:]] ]] # Number followed by a letter? Illegal!
# ...
# From ktour.sh example script.
```

These character classes may even be used with globbing, to a limited extent.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

POSIX character classes are used in Example 16-21 and Example 16-22.

Sed, awk, and Perl, used as filters in scripts, take REs as arguments when "sifting" or transforming files or I/O streams. See Example A-12 and Example A-16 for illustrations of this.

The standard reference on this complex topic is Friedl's *Mastering Regular Expressions*. *Sed & Awk*, by Dougherty and Robbins, also gives a very lucid treatment of REs. See the *Bibliography* for more information on these books.

Notes

[1] A *meta-meaning* is the meaning of a term or expression on a higher level of abstraction. For example, the *literal* meaning of *regular expression* is an ordinary expression that conforms to accepted usage. The *meta-meaning* is drastically different, as discussed at length in this chapter.

[2] Since sed, awk, and grep process single lines, there will usually not be a newline to match. In those cases where there is a newline in a multiple line expression, the dot will match the newline.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here Document
line1
line2
EOF
# OUTPUT:
# [line1
# line2]



echo
```

```
awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# OUTPUT:
# line
# 1


# Thanks, S.C.

exit 0
```

Prev
Regular Expressions

Home
Up

Next
Globbing