COMP 2718: Regular Expressions

By: Dr. Andrew Vardy

Outline

- Introduction
- `grep` - **G**lobal **R**egular **E**xpression **P**rint
- Metacharacters and Literals
- Crossword Example
- Bracket Expressions
- POSIX Character Classes
- Basic Vs. Extended Regular Expressions
- Alternation
- Quantifiers
- Examples
- {} - Matches an Element a Specified Number of Times
- Regular Expressions in Shell Scripts
- `grep1.sh` - `grep` as a Shell Script
- `BASH_REMATCH`

Introduction

A **regular expression** is a symbolic notation for finding patterns in text. They are useful in many ways:

- Searching for patterns
- Validating data (e.g. is a given phone number properly formatted)
- Syntax highlighting

Most programming languages provide support regular expressions.

Regular expressions are similar to globbing, but also very different. Some of the same characters are used, but their meaning differs. So be aware if you are using globbing or regular expressions—**they are not the same**.

grep - **G**lobal **R**egular **E**xpression **P**rint

One can work with regular expressions using many different tools and programming languages. We will start by using grep for illustration.

So far we have used grep only to search for fixed strings. e.g.

```
$ ls /usr/bin | grep zip
```

Actually, "zip" is being used here as a regular expression—just a very straightforward one. Here is the form of the grep command:

```
grep [options] regex [file...]
```

Lets look at some of grep's options, then dive into regex. . .

Option	Description
`-i`	Ignore case. Do not distinguish between upper and lower case characters. May also be specified `--ignore-case`.
`-v`	Invert match. Normally, `grep` prints lines that contain a match. This option causes `grep` to print every line that does not contain a match. May also be specified `--invert-match`.
`-c`	Print the number of matches (or non-matches if the `-v` option is also specified) instead of the lines themselves. May also be specified `--count`.
`-l`	Print the name of each file that contains a match instead of the lines themselves. May also be specified `--files-with-matches`.
`-L`	Like the `-l` option, but print only the names of files that do not contain matches. May also be specified `--files-without-match`.
`-n`	Prefix each matching line with the number of the line within the file. May also be specified `--line-number`.
`-h`	For multi-file searches, suppress the output of filenames. May also be specified `--no-filename`.

We'll follow some examples from chapter 19 of the textboook. First, we create some files do perform pattern matching on:

```
$ ls /bin > dirlist-bin.txt
$ ls /usr/bin > dirlist-usr-bin.txt
$ ls /sbin > dirlist-sbin.txt
$ ls /usr/sbin > dirlist-usr-sbin.txt
$ ls dirlist*.txt
dirlist-bin.txt dirlist-sbin.txt dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

In the following, note the behaviour of `grep` with...

- no options matches + filenames
- `-l` just filenames with matches
- `-L` filenames **without** matches

```
$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover

$ grep -l bzip dirlist*.txt
dirlist-bin.txt

$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Metacharacters and Literals

In the searches above we used the regular expression "bzip". These four characters are considered **literal characters**. The power of regex's comes by using the following **metacharacters**:

```
^ $ . [ ] { } - ? * + ( ) |
```

The meanings of these characters are totally different from their usage in `bash`. **Therefore, you need to enclose regular expressions in quotes to prevent the shell from expanding them.**

The Any Character: .

A dot or period represents any single character. e.g.

```
$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
gunzip
[...more results follow... "zip" is not included]
```

The Anchors: ^ (start of line) and $ (end of line)

^ means the start of the line and $ means the end of the line.

```
$ grep -h '^zip' dirlist*.txt zip
zipcloak
[...]
zipsplit

$ grep -h 'zip$' dirlist*.txt gunzip
gzip
unzip
[...]
zip

$ grep -h '^zip$' dirlist*.txt
zip
```

Crossword Example

Lets say you are solving a crossword and looking to answer the following question:

> *What's a five letter word whose third letter is 'j' with last letter 'r'?*

The following provides some answers:

```
$ grep -i '^..j.r$' /usr/share/dict/words
Gujar
Kajar
Major
major
```

Bracket Expressions

A list of characters in square brackets means a single-character match to one of those characters. For example:

```
$ grep -h '[bg]zip' dirlist*.txt
```

```
bzip2
bzip2recover
gzip
```

This matches any line containing “bzip” or “gzip”.

Except for caret (^) and dash (-) other metacharacters lose their special meaning in a bracket expression.

Negation

If `^` is the first character in a bracket expression, it means logical negation. The match must **not** include any of the subsequent characters. e.g.

```
$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
[...none containing "bzip" or "gzip"...]
```

Traditional Character Ranges

A range of characters or digits can be specified within a bracket expression in the form `[x-y]` where `x` is the first possible character and `y` is the last possible character. e.g.

```
$ grep -h '^[A-Z]' dirlist*.txt
```

This matches filenames that start with capital letters.

Multiple ranges are also possible where the match must occur in one of the ranges. e.g.

```
$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

This matches any filename starting with letters or numbers. (I say 'filename' because that's what this command is dealing with, but in general regular expressions are for string matching.)

If you actually want to match to a literal dash character you should put it as the first entry in a bracket expression. e.g. The following matches any filename with containing '-', 'A', or 'Z'.

```
$ grep -h '[-AZ]' dirlist*.txt
```

POSIX Character Classes

Sometimes the traditional character ranges (e.g. `[A-Z]`) don't work. Recent versions of `bash` seem to be okay, but some programs can become broken because of confusion between different character orderings. Traditionally, the characters were ordered in **collation order**, the same ordering used in ASCII:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Yet this is different from the conventional **dictionary order**:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

To prevent confusion between these different orderings you can use the POSIX Character classes. These are the same classes that can be used in globbing...

Character Class	Description
`[:alnum:]`	The alphanumeric characters. In ASCII, equivalent to: `[A-Za-z0-9]`
`[:word:]`	The same as [:alnum:], with the addition of the underscore (_) character.
`[:alpha:]`	The alphabetic characters. In ASCII, equivalent to: `[A-Za-z]`
`[:blank:]`	Includes the space and tab characters.
`[:cntrl:]`	The ASCII control codes. Includes the ASCII characters 0 through 31 and 127.
`[:digit:]`	The numerals zero through nine.
`[:graph:]`	The visible characters. In ASCII, it includes characters 33 through 126.
`[:lower:]`	The lowercase letters.
`[:punct:]`	The punctuation characters. In ASCII, equivalent to: `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]`
`[:print:]`	The printable characters. All the characters in `[:graph:]` plus the space character.
`[:space:]`	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed. In ASCII, equivalent to: `[ \t\r\n\v\f]`
`[:upper:]`	The uppercase characters.
`[:xdigit:]`	Characters used to express hexadecimal numbers. In ASCII, equivalent to: `[0-9A-Fa-f]`

The POSIX character classes provide protection against different character orderings, but they don't support partial ranges such as `[A-M]`.

What is POSIX?

The IEEE (Institute of Electrical and Electronics Engineers) developed a standard for application programming interfaces (APIs), shell, and utilities for Unix-like systems. The name for this standard is Portable Operating System Interface (POSIX). POSIX-compliance means that a feature is likely to work the same on different Unix-like OS's (e.g. Unix, Linux, Mac OS X).

See `https://en.wikipedia.org/wiki/POSIX`.

Basic Vs. Extended Regular Expressions

The POSIX standard separates **basic regular expressions (BRE)** from **extended regular expressions (ERE)**. BRE just uses the following metacharacters:

```
^ $ . [ ] *
```

ERE adds the following (to be discussed below):

```
( ) { } ? + |
```

By default `grep` uses BRE. To use ERE with `grep` just specify the `-E` option.

Alternation

ERE adds support for **alternation** which is the ability to choose one of a set of matches. The matches are separated with the '|' or pipe character.

```
$ echo "AAA" | grep -E 'AAA|BBB'
AAA

$ echo "BBB" | grep -E 'AAA|BBB'
BBB

$ echo "CCC" | grep -E 'AAA|BBB'
$
```

Alternation can be applied on more than two choices:

```
$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

Alternates can be combined with other metacharacters and sequences of matches.

Use parentheses characters ‘(’ and ‘)’ to separate the alternation:

```
$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

This matches filenames that begin with “bz”, “gz”, or “zip”. What would be matched if we left off the parentheses?

```
$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

This matches any string that starts with “bz” and *contains* “gz” or “zip”.

Quantifiers

There are four types of **quantifiers** that specify how many of the preceding **element** are matched. An element could be a single character or a group of characters (e.g. an alternation). Here are the quantifiers:

`?`	Match an element zero or one time (0, 1)
`*`	Match an element zero or more times (0, infinity)
`+`	Match element one or more times (1, infinity)
`{n,m}`	Match element n - m times (n, m). Other variations possible—see below.

Examples

e.g. Phone Numbers

Assume the following two forms of phone numbers are considered valid:

```
(nnn) nnn-nnnn
```

```
nnn nnn-nnnn
```

The following regex accepts both forms:

```
^\(?[0-9][0-9][0-9]\)?
[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

(Line above broken in two.)

Note that the parentheses are escaped (with \). The parentheses elements are followed with ? which means to match 0 or 1 times. In other words, they are optional.

e.g. Sentences

Lets assume that a sentence begins with a capital letter, is followed by any number of upper and lowercase letters and spaces, and ends with a period. Obviously, this is pretty crude. The following is a valid sentence by this criterion.

```
Sadsfnwe asdfnwe JasdfJ er.
```

Here is the regex to recognize such a sentence:

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

Note the use of `*` which means match any number of times (0, infinity). The period at the end is escaped because `.` is a metacharacter. The use of POSIX character classes make this look a bit messy. Using traditional character ranges it would look like this:

```
[A-Z][A-Za-z ]*\.
```

e.g. Words with single spaces between them

The following regex matches lines that consists of groups of alphabetic characters (i.e. words) separated by single spaces.

```
^([[:alpha:]]+ ?)+$
```

Note that the parenthesis are used here to establish an element (word and space) that must be repeated one or more times. + means match 1 to infinity times. This quantifier is used twice for the characters in the word to the (word and space) element. Note that the following strings would not match:

```
a b 9
abc  d
```

{} - Matches an Element a Specified Number of Times

Sometimes we want to specify the number of matches. The `{ }` quantifier achieves this and has the following options:

Specifier	Meaning
`{n}`	Match the preceding element if it occurs exactly *n* times.
`{n,m}`	Match the preceding element if it occurs at least *n* times, but no more than *m* times.
`{n,}`	Match the preceding element if it occurs *n* or more times.
`{,m}`	Match the preceding element if it occurs no more than *m* times.

We can use this to improve (by shortening) our regex for phone numbers:

```
^\(?[0-9][0-9][0-9]\)?
[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

To this:

```
^\(?[0-9]{3}\)?  [0-9]{3}-[0-9]{4}$
```

Regular Expressions in Shell Scripts

Recall the compound command `[[ ]]` that can act as a replacement for the `test` command and its single bracket `[ ]` form. The `[[ ]]` compound command enables an additional operator for extend regular expression matching:

```
[[ "$string" =~ regex ]]
```

This is true (exit status 0) if the `string` variable matches the regex. Somewhat surprisingly, the regex should **not be quoted**. If it is quoted, it is treated as a regular string. Space characters should be individually quoted or escaped. Consider the following examples:

```
$ if [[ 'asdf' =~ ^[a-z]+$ ]]; then echo "Yes"; fi
Yes
```

Now try adding a space.

```
$ if [[ 'asdf asdf' =~ ^[a-z]+$ ]]; then echo "Yes"; fi
$
```

No match, which makes sense. Lets modify our regex to include spaces:

```
$ if [[ 'asdf asdf' =~ ^[a-z ]+$ ]]; then echo "Yes"; fi
-bash: syntax error in conditional expression
```

Syntax error(s) because the right-hand side now has two parts. Try escaping the space character.

```
$ if [[ 'asdf asdf' =~ ^[a-z\ ]+$ ]]; then echo "Yes"; fi
Yes
```

Another strategy is to put the regex into a variable which can be quoted.

```
$ regex='^[a-z ]+$'
$ if [[ 'asdf asdf' =~ $regex ]]; then echo "Yes"; fi
Yes
```

`grep1.sh` - `grep` as a Shell Script

The following script will implement a simplified version of `grep` as a shell script. The first part of the script does the following:

- Prints a usage line and exits if there are 0 or more than 2 arguments.
- Accepts `$1` as the regex pattern
- If there are 2 arguments, uses `$2` as the file to search. Otherwise uses standard input, located at `/dev/fd/0`.

The second part of the script is a loop that checks for a match on each line of the file and increments a `count` variable for each match.

```
#!/bin/bash
if [ $# -eq 0 -o $# -gt 2 ]; then
    echo grep1.sh pattern "[ file ]"
    exit 1
fi
pattern=$1
source=/dev/fd/0    # Use stdin
if [ $# -eq 2 ]; then
    source=$2       # Use file $2
fi                  # instead of stdin

count=1
while read line; do
    if [[ "$line" =~ ${pattern} ]]; then
        echo "$count: $line"
    fi
    (( count = count + 1 ))
done < $source
exit 0
```

BASH_REMATCH

bash also provides a feature to access individual parts of a regular expression match. This is through the BASH_REMATCH variable. Actually, this is an **array variable**. ${BASH_REMATCH[0]} will hold the entire matched string (if it exists).

In order for BASH_REMATCH to be useful you have to define **capture groups** within the regex. These are defined with ( ). For example this regex defines three capture groups:

```
([a-z])([a-z])([a-z])
```

After the regular expression matching with the =~ operator, the matched elements (in this case, single characters) will be available through the following:

```
${BASH_REMATCH[1]}
${BASH_REMATCH[2]}
${BASH_REMATCH[3]}
```

The following example illustrates BASH_REMATCH:

```
$ [[ "asdf" =~ ([a-z])([a-z])([a-z]) ]]
$ echo ${BASH_REMATCH[1]}
a

$ [[ "asdf" =~ ([a-z])([a-z])([a-z]) ]]
$ echo ${BASH_REMATCH[2]}
s

$ [[ "asdf" =~ ([a-z])([a-z])([a-z]) ]]
$ echo ${BASH_REMATCH[3]}
d

$ [[ "asdf" =~ ([a-z])([a-z])([a-z]) ]]
$ echo ${BASH_REMATCH[0]}
asd
```