Regular Expressions

Needles in a Haystack

You come across regular expressions at every step of the way on Unix systems. But what exactly are they, and how do you use them? BY MARC ANDRÉ SELIG

In short, regular expressions are simply placeholders for specific strings. *Regexps*, as the abbreviation goes, are used for a variety of search operations – for example, in most text editors, as well as in your favorite Unix **scripting language**.

Innumerable special functions and exceptions make regular expressions a slightly dry experience, or even a daunting prospect. If you are still struggling with the basics, I really would like to wish you the best of luck – believe me, it is worth the effort!

Encounters with Regexps

If you make regular use of the command line, you will definitely use *grep* for searching in files. This tool even derives its name from regular expressions. The ***ed*** command *g/re/p* prints every line in a file that contains the regular expression *re* – and that is exactly how *grep* works.

grep requires at least one search expression as a command line argument. The target file(s) for the search operation are supplied as an additional argument. Imagine you want to create a list of all the users on your system whose name starts with the letter "t" and who use *bash* as their **login shell**. You can type

```
grep -i ^t.*bash /etc/passwd
```

to display them on screen.

If you do not supply a file name in the *grep* command line, the tool simply

TABLE 1: USEFUL *GREP* OPTIONS

Command	Description
-i	Ignore case
-l	Display on the files where the regular expression exists – do not display the the lines of text
-v	Reverse search: Display only lines of text that *do not* contain the expression
-3	Display the line containing the expression and three lines before and after that line
-A 3	Display the line containing the expresssion and three lines after that line
-B 3	Display the line containing the expresssion and three lines before that line

searches standard input. This is quite useful if you need to process another program's output. Imagine you wanted to search for the Konqueror processes belonging to the user *mas*; you could type the following command:

```
ps -auwx | grep mas.*konqueror
```

The forwards / and ? reverse search functions in the *vi* text editor can also be regular expressions. However, this would tend to cause confusion in *emacs*, where incremental search operations are involved. In this case you need to explicitly search for a regexp.

The command for a standard incremental search is M-C-s, that is [Ctrl-Alt-S]. M-x [Alt-X], or search-forward-regexp is slightly more straightforward: In this case you first enter the complete search expression and then perform the search.

Some people's first experience with regexps comes from *perl*, but most other scripting languages (such as Tcl, Python and PHP) can handle them. There are some lesser known regexp consumers such as the *sed* or *awk* languages, but even the C++ GUI toolkit, Qt, which you can use to write KDE programs can handle regular expressions.

Building Blocks

In the case of simple regular expressions, you merely enter the string you want to search for. You can type most ASCII characters directly: Even a single character can be a search string, albeit a primitive one.

Our next lesson is: There are some metacharacters that enable special functions. The most commonly used metacharacter is the period, which is a wildcard for any other character. The regexp `be..` will search for *be* followed by any two characters. This would allow you to find both "bear" and "bean" and even "bed" (followed by a space character). However, what it would not find is a lonely "bed" without the trailing space character.

The ^ character searches for a new line. This kind of character is referred to as an *anchor* as it anchors the search expression at the beginning of a new line. You could therefore type *^bear* to search for the word "bear" at the beginning of a new line. The dollar sign *$* does the same thing at the end of a line. So if you want to search for the last bean in a line you would need to type *bean$*.

Since regular expressions are normally evaluated line by line, additional characters prepended to ^ or appended to $ normally make little sense. Languages like Perl sometimes allow exceptions in this case – we will get back to that subject later.

Regular expressions are really useful when you use combinations! *^One$* will match any lines that contain only the word "One" and nothing else. You can type ^$ to find empty lines where the new line character is immediately followed by the end of line character.

Warning: Constructs such as ^ and $ do not represent characters in the target files but instead find spaces between two characters.

```
grep ^a$ textfile
```

for example, will output exactly two characters for each match, the letter "a" and a new line, although the regular expression comprises three characters.

Alternatives and Repetitions

Sometimes you do not know exactly what you are looking for. If multiple occurrences of a character (or an expression to be more precise) are permissible, you can use one of the following operators: The asterisk * indicates that multiple occurrences of a character are permissible. *a** thus represents one or multiple "a"s – or even an empty string that contains exactly zero "a"s and thus complies with the requirement "any number". *Mo*rs* thus matches "Moors" or "Mooooooooors" but also "Mrs".

In contrast to this, the plus character + repeats a character – it must occur at least once, but can occur more often. The regular expression *jj+n* will search for two "j"s, one of which can be repeated, followed by an "n". Thus this expression will match "jjn" or "jjjjjjjjn", but not "jn".

Most of the time repetition wildcards such as + and * are not used in the context of specific characters but to repeat the period that represents any other character. The regular expression .* thus represents any number of occurrences of any character, i.e. it represents any string. The regexp *Jo.*nes* can thus represent any line that contains "Jo" followed by "nes" anywhere in the line – this matches "Jones", "Johannes" but also "Joe bought some fresh meaty bones for his dog".

.* will always search for a match that is as long as possible. The expression *a.*b* in the string "abcabcabc" thus matches "abcabcab". If you do not need this, you can just add a question mark in perl. .*? will search for a match that is as short as possible, so *a.*?b* will simply find "ab" in "abcabcabc".

You can use a single question mark *?* to represent optional occurrences: The preceding letter can occur exactly once or not at. *ab?c* represents "ac" or "abc".

The option of defining the number of repetitions is more rarely used, and we will be discussing it for the sake of completeness only. If you need this option, you use braces that contain the minimum and maximum counts. *a{1,7}* represents one through seven "a"s. *a{,7}*

GLOSSARY

Script language: *A programming language typically used for authoring (mainly smaller) programs (scripts) that do not need to be converted into an executable format by a compiler, but are interpreted and executed by an interpreter when the source file is called. The most common examples are shells (such as Bash) or Perl.*

ed: *The classic Unix line editor that does not allow you to edit a whole file, in contrast to modern text editors, but provides commands that can be applied to a single line or multiple lines.*

Login shell: *The shell that presents itself to a user when he or she logs on via the command line. The system administrator uses an entry in the /etc/passwd file, which also includes the user name, to specify what file this should be. (Modern Linux systems normally no longer store encrypted passwords in /etc/passwd but shadow them in the /etc/shadow file.)*

GUI Toolkit: *A program library that provides functionality for authoring graphic user interfaces (GUIs), for example, classes for windows, scroll boxes, menu bars. The most common GUI toolkits for Linux, Qt for C++ and GTK for C, also provide a range of classes and functions for other purposes.*

means at the most seven, and *a{4,}* at least four "a"s.

You can use the pipe character | to use alternative search expressions. Thus,

```
grep -E '(bus|train|plane)' ↩
vehicles.txt
```

will only show public transport vehicles. The *grep* flag *-E* tells the search tool to expect an "extended" regular expression. Without this option the tool will interpret this as a normal search string.

Character Ranges

A *character range* is an expression that represents multiple characters, e.g. only letters or accented characters. This is not used very often for interactive tasks with *emacs* or *vi*, but character ranges are quite useful in scripting languages – for example, you can define a range of valid input characters when you are programing **CGIs**.

Ranges are defined by square brackets surrounding the valid characters. *[abc]* includes "a", "b" or "c". You can combine ranges like the one just mentioned with other metacharacters: *[abc]+* thus matches "a", "aa", "abababc", etc.

Dashes (alias minus signs) facilitate the definition of larger ranges. *[a-z]* represents any lower case letter. *[a-zA-Z0-9]* represents any alphanumeric character. If the dash itself is to be included in the range, you will need to prepend the range with this character. The range *[-+a-z]* comprises lower case letters and the plus and minus signs.

If the first character within the square brackets is a circumflex accent ^, the definition of the range is inverted, i.e. the expression only match strings where these characters do *not* occur. To search for any characters with the exception of "Z", you can use *[^Z]*. If you want to find any lines in a file that do not start with "Y" or "Z" (and are not empty), you would type *^[^YZ]*. The circumflex at the start of the regexp indicates a new line which must be followed by any character apart from Y or Z.

Predefined Ranges

Most libraries containing regular expressions define short forms for common ranges, and can save you some typing. As you might have guessed from the lack of enthusiasm in the last sentence, you should not expect global standardization …

You will find an overview of some of the most important pre-defined ranges in Table 2. The table includes the so-called **POSIX** character classes, which most engines accept – such as *grep* or perl. You will also find the perl short forms, which are somewhat cryptic to understand but easier to type.

The big advantage of these pre-defined ranges in comparison to homegrown definitions such as *[a-zA-Z]* is that pre-defined ranges normally allow you to use a **locale**, i.e. if you work in a French locale, accented characters count as alphanumeric characters.

The second big advantage does not really apply to Linux: Theoretically you could have a character set that is incompatible to ASCII, where the letters of the alphabet are out of sequence or not correctly sorted. In this case "[a-z]" might not include all the lower case letters (which it should), but instead include some special characters (which it should not). *[[:lower:]]* is guaranteed to contain lower case letters only, no matter what character set is in use.

Special Characters

You can use special characters (such as umlauts, tabs, control characters, etc.) for most implementations of regular expressions. To do so, just enter them directly as a regexp. If this is impractical (because you cannot distinguish a tab from a space in a program listing), you can use the notation common to C and most shells, for example *\t* for a tab. If you need to search for a backslash, type the character twice: \\.

Groups

Before we get down to practical cases, let us look at another important construct. Repetition characters such as * always apply to the character, or to be more precise, the expression that immediately precedes them. Thus, *abc** repeats only the "c"; this regexp will match "abcc", but not "abcabc".

However, you can use parentheses to group a regexp. *(abc)* effectively means the same as *abc*, but the internal workings of the search operation are entirely different: *abc* comprises three search expressions, "a", "b", and "c", which must be found in sequence. *(abc)*

TABLE 2: PRE-DEFINED RANGES IN POSIX AND PERL

POSIX Character Class	Short form for range in perl	Description
[[:digit:]]	\d	"Number": a number between 0 and 9
[^[:digit:]]	\D	"not a number": anything apart from digits
[[:alpha:]]		"alpha": letters (including local accented characters and similar)
[[:alnum:]]		"alphanumeric": letters and numbers
[[:word:]]	\w	"word": alphanumeric character or underscore "_" (not in POSIX!)
[^[:word:]]	\W	"non word": not alphanumeric or the underscore
[[:lower:]]		lower case letters
[[:upper:]]		upper case letters
[[:punct:]]		punctuation marks
[[:space:]]	\s	"whitespace": space character, tab or new line, POSIX includes the rare vertical tab
[^[:space:]]	\S	"non whitespace": anything apart from space, tab or new line
[[:blank:]]		"vertical whitespace": space or tab

GLOSSARY

CGIs: *Scripts or compiled programs that are stored on a web server, call a specific web page when launched and generate a HTML document "on the fly" (dynamically). CGI is short for "Common Gateway Interface".*

POSIX: *An attempt to standardize typical Unix functionality and definitions (IEEE Standard 1003.1). Most Linux programs can be made POSIX compatible if required, although this may mean doing without some of the more advanced functionality.*

Locale: *POSIX supports automatic adapting of programs to the local environment. One obvious example would be displaying local translations of system messages or man pages. The locale also includes local formats for pages, time or date values, measurements, or preferred paper sizes, and of course information the function fulfilled by specific sections of the character set – that is, whether character 196 should represent an "Ä" or a non-printable character.*

TABLE 3: PERL VARIABLES FOLLOWING A SUCCESSFUL SEARCH

Name	Description
$&	The last string found, i.e. what ever matched the regexp.
$` (backtick)	Part of a string before the match.
$´ (Forward tick)	Part of a string after the match. After a successful search the entire search string is split into $´$&$´.
$+	The last matching group. If a regular expression comprises multiple group constructs, where some are optional, you can use this to access the contents of the last group.

contains the group "abc" as an individual search expression to which other functions can be applied: *(abc)** repeats the whole group. This expression will match both "abcabcabc" and "abc" or even an empty string, but not "abcc".

The Taming of the Shell

One important reason for the "popularity" of regular expressions is the fact that the incredible confusion that using them can cause. Regular expressions are so difficult to read when they start to get more complex. To prove a point, here is an example from the *perlfaq6* man page:

```
/\*[^*]*\*+([^/*][^**]*\*+)*/|↩
("(\\.|[^"\\])*"|'(\\.|[^'\\])↩
*'|\n+|.[^/"'\\]*)
```

This monstrosity is supposed to find comments in C programs at the same time ignoriing possible comment characters in strings. No, I have not tried it, and no, experts cannot really "read" expressions like that, although they might be able to piece together the eventual outcome.

Various other complexities can make your life miserable. Regular expressions are used in thousands of different programs, and of course each program has its own implementation with proprietary features and a small smattering of exceptions. The first trap you tend to fall into is backslashes, particularly in combination with parentheses to provide group definitions. As we have seen, *(abc)+* searches for repetitions of the "abc" string, such as "abcabc" or "abcabcabc". Unfortunately you often have to enable special characters like parentheses or the plus by adding a slash. If you use perl, the correct regexp would be *(abc)+*. However, in *grep* you would need to type *\(abc\)\+*. *egrep* or *grep -E* searches for "extended regular expressions" and understands *(abc)+* as is.

The shell really likes backslashes – in fact so much so that it eats them for breakfast. Would you like a demonstration? The command :-

```
echo \(abc\)\+
```

displays "(abc)+". See what I mean, the shell has "eaten up" the backslashes. So you need to prevent backslashes from disappearing into the depths of the shell by adding another backslash. *grep \\\(abc\\\)\\+* works, but it might be easier and more readable to add quotes instead: *grep '\(abc\)\+'* has the same effect and is clearer.

Differences in Libraries

I have already mentioned that different programs will deal with regular expressions in different ways. If you only have to deal with a single tool when authoring a perl script, for example, this probably will not cause you too many headaches. But if you switch tools, or use multiple tools simultaneously, you may find yourself facing a few issues.

The differences mainly concern two points. Does the program expect a "simple" or "extended" expression? This boils down to the question of whether you need to enable special functions, such as + or parentheses, by prepending a backslash \, or assume that the functions are enabled by default (and need to be *dis*abled using a backslash).

Rule of thumb: Most script languages use extended regular expressions, you can thus directly use the functions presented here. In addition to grep we can also use egrep, which understands extended regexps. Most editors and command line instructions however expect simple regular expressions. Generally if something does not work then try again but supplement backslashes at the strategic points.

The second issue is not such a big deal in real life situations. Sophisticated special functions are normally private extensions that, of course, will not be available in any other program. Perl, for example, will allow you to place comments in regular expressions and offers special functions for virtual expressions (predictive expressions that check whether a certain string *follows* the regexp without the text needing to be part of the regexp) – this would make no sense in *grep*.

Backlinks to Expression

As previously discussed, parentheses are used to define groups of characters. So far we have only used these groups to repeat strings.

But the genuine task for these groups is completely different: A group can define a substring that you can refer to later. When the program finds an expression in parentheses, it will store

TABLE 4: MODIFIERS FOR REGULAR EXPRESSIONS IN PERL

Tailing the last slash	Description
/i	Non case-sensitive search.
/s	The period also applies to new lines in the string where you are searching.
/m	The string you are searching in can contain multiple lines (similar to *grep*), where ^ and $ always represent the start or end of a line in the string.
/g	On searching and replacing, do not stop at the first match, but continue through the whole string.

Listing 1: Sample file connections.txt

```
B      by     to P   by     to M
10:45  bus    11:52  train  13:05
10:49  train  11:19  train  12:05
11:45  bus    12:54  bus    15:10
12:45  bus    13:51  train  15:05
13:49  train  14:19  train  15:05
```

the expression, allowing you to call it in a different part of the program.

Example: The regexp *(bus|train)* matches both a bus and a train. Since the expression is enclosed in parentheses, the software records any matches – "bus" or "train". The string is stored in a variable with a serial number between 1 and nine for the sake of simplicity. Variable 1 comprises the contents of the first group, variable 2 the content of the second group, and so on.

You can use the \1 ff. construct and to refer to these variables. Let's use the file *connections.txt* from Listing 1. This contains a list of the connections from Berlin via Paris to Madrid.

Now when I'm travelling, I try to avoid walking from the bus station to the train station. So I really just want to view the connections that allow me to travel the whole trip by bus or by train. I can use the following syntax to do so:

```
egrep '(bus|train).*\1' ⮐
connections.txt
```

The regexp first looks for "bus" or "train" and saves the results in variable 1. Any string can follow this (".*") provided the content of variable \*1* also occurs. So if "bus" occurs, the line must also contain a second instance of the word "bus", in the case of "train" the word "train" must occur twice.

Listing 2: Output from egrep command

```
10:49 train 11:19 train 12:05
11:45 bus   12:54 bus   15:10
13:49 train 14:19 train 15:05
```

The output is shown in the listing above (Listing 2).

Backward References in Perl

Perl also handles *backward references* perfectly. Within a regular expression the content of the groups is stored in the special numerical variables \1 through \9.

Additionally, perl stores regular perl variables that you can use to access matches outside of the regular expression. The content of \1 is thus placed in the perl variable *$1*, and the content of \2 in *$2*, etc.

The following is a quick example from a primitive Parser:

```
/^\s*From\s+(\w+)\s+Type\s+⮐
(\w+)\s+Seq\s+(\d+)\s+DB\s+⮐
(\w+)\s*$/i
    or the "Invalid pattern in ⮐
packet: \"$_\"";
my ($from, $type, $seq, $db) = ⮐
($1, $2, $3, $4);
```

Now the variables *$from*, *$type*, *$seq* and *$db* contain the matching strings. If you type

```
From mas Type ACK Seq 4219 DB ⮐
Pharma
```

thus *$from = "mas"*, *$type = "ACK"*, *$seq = 4219* and *$db = "Pharma"*.

Additional Perl Functions

Besides groups and the like, perl uses additional variables after performing a search. Table 3 provides an overview.

The perl regexp engine is without a doubt one of the quickest available for Unix. A few practical perl one-liners will allow you to save some work on the command line.

The following line allows perl to function as a substitute for *grep*, outputing any lines that include the regular expression *abc*:

```
perl -ne 'print if /abc/' ⮐
filename.txt
```

Perl can also be used as a substitute for *sed*. In our example the regexp *abc* is replaced by the *def* string and the results are displayed on screen:

```
perl -pe 's/abc/def/g' ⮐
filename.txt
```

The next command is similar, however, here we are replacing *abc inside* the file *def* and creating a backup copy in *filename.txt ~* :

```
perl -pi~ -e 's/abc/def/g' ⮐
filename.txt
```

You will normally use *m/regexp/* or simply */regexp/* for simple search operations in perl. To search and replace, use *s/regexp/replacetext/* instead. You can also use one of the modifiers found in Table 4 instead of the last slash.

Troubleshooting

Complex regular expressions rarely work on your first attempt. Troubleshooting regexps costs time and nerves – make sure you have an ample supply of coffee or cola!

The first step in the command line should be to prepend the *echo* command. Remember, the shell might devour some of your characters. You can soon get to the bottom of this issue by displaying the command while it is executing. But even in scripts you would be well advised to display the regular expression, the matches or at least part of these. You should place commands that allow this in your script. The perl debugger permits interactive debugging of the regular expressions.

If that does not help, simply split up the expression into smaller sections. Check whether all the parts really do what you intended them to. And most importantly – stay calm!.

Prospects

This short article should provide you with an impression of the capabilities and possibilites, but also the complexity of regular expressions. If you want to delve deeper into this field, there is only one way to go: practise.

Reading or dissecting complex regexps that you come across in scripts is both instructive and frustrating. You might prefer to keep using regular expressions. The first time you need to perform two similar searches in short succession, you might find yourself wondering if a single regexp would also have done the trick.

Also have a look at the *grep* und *perlre* man pages and your perl documentation *man perltrap* and *man perlfaq6*.

THE AUTHOR

Marc André Selig spends half of his time working as a scientific assistant at the University of Trier and as a medical doctor in the Schramberg hospital. If he happens to find time for it, his currenty preoccupation is programing web based databases on various Unix platforms.