Pattern Matching and Substitution in bash

R (Chandra) Chandrasekhar

2023-02-28 | 2023-03-07

The arcane powers of the bash shell

The programs, `sed`, `awk`, `grep`, and `perl` have been the traditionally used tools for matching and manipulating lines and strings in Linux. But the bash shell [1, 2] also embodies powerful pattern-matching and substitution capabilities [3–6], many of which are relatively unknown and unused. This blog gives practical examples for using these powerful, but somewhat understated features, to achieve common tasks efficiently and tersely, directly from within `bash` itself.

Extended globbing

Globbing is the unflattering term—an abbreviation for *global*—used to denote an operation to extract files satisfying certain conditions [7–9]. It is applicable also to the `bash` command line. For our purposes, depending on the sort of matching we perform, it will be sometimes necessary to set `shopt -s extglob` after the shebang line [10].

Parsing filenames

A fully qualified filename consists of a `path`, a `basename`, and an `extension`. While not all filenames are encountered in their full glory, it helps to decompose any given filename into its constituent parts to help with housekeeping functions on a machine running `bash`—for example, to facilitate searching, sorting, renamimg, and other file-related functions.

The canonical filename

A canonical filename should comprise these components:

1. a *path* with the forward slash / as the separator[1] between elements denoting the path;
2. a *filename* in two parts:
   (a) comprising a *basename* which appears immediately after the *last* / character; and
   (b) a *file extension* that occurs after the basename and immediately after a `.` or period character.

[1] The separator in Microsoft Windows is a *backslash*, \, but since we are discussing `bash`, running on Linux machines, it is the *forward slash*, /, that is our character of interest.

/my_path/is/quite/long/basename.ext is a canonical filename—hereafter referred to as fullname—where the above-named structural elements are as follows:

1. path: /my_path/is/quite/long
2. basename: basename
3. extension: ext
4. filename: basename.ext

Parsing the fullname

Our next task is to dissect the canonical filename into its above components using pattern-matching in bash:

```
#!/bin/bash
#file-parse.sh
shopt -s extglob

fullname=”/my_path/is/quite/long/basename.ext”
echo ”fullname is $fullname”

#
# Extract $path
# Approach from the right until the _first_ `/`
# is encountered and throw away everything
# from the _right_ end up to and including that `/`.
#
path=”${fullname%/*}”
echo ”path is $path”

#
# Extract $filename
# Approach from the left until the _last_ `/` character
# is encountered and throw away everything
# from the _left_ end up to and including that `/`.
#
filename=”${fullname##*/}”
echo ”filename is $filename”

#
# Extract $ext
# Approach from the _left_ until the _last_ `.` character
# is encountered and throw away everything
# from the _left_ end up to and including that last `.`.
#
ext=”${fullname##*.}”
echo ”extension is $ext”
```

```
#
# Extract $basename
# This requires trimming strings from both
# the left and the right of $fullname
# and requires _two_ steps if start with $fullname.
#
# Instead, we use $filename, which is already available,
# and excise the extension to get $basename.
#
# For this, we approach from the _right_ until we encounter
# the _first_ `.` character and throw away everything
# from the _right_ up to and including that first `.`.
#
basename=”${filename%.*}”
echo ”basename is $basename”
```

Mnemonics behind the # and % symbols

The use of the symbols # and % in the pattern matching expressions might seem arbitrary or whimsical. For a start, they do not conform to the usual delimiters ^ and $ for the beginning and end of a line. Because we are dealing with strings rather than lines here, those symbols are not used.

One other point to keep in view constantly is to avoid looking at bash pattern matching solely through the lens of regular expressions [11–13]. There are some similarities, but the two are not identical.

So, what's the dope on # and %? These two symbols have been chosen for their near universal usage as a prefix and suffix respectively. It is customary to write #1 for "number one", and 20% for "twenty percent", where you will notice that the # is written as a *prefix* and the % is written as a *suffix* to the number.

In the bash pattern-matching we have encountered so far, we are matching elements in a string, *and throwing away the matching portion*, using some known delimiter. When we match from the left, we use # because it is a prefix. Likewise, when we match from the right, we use %, which is a suffix. In both cases, we stop at the first match from whichever direction we are starting the search for the match. The single # and % therefore denote lazy matching.

The symbol ## means we deal with the *longest* substring from the left that matches: a case of greedy matching. The same applies to %% where we stop at the longest matching substring from the right.

If you look carefully, you will see that—apart from the anchor character(s)—we do not care about what we are throwing away. We can therefore denote these "don't care" characters with the *, which is a wildcard that denotes zero or more characters. What *is* important to us, though, is the *delimiter* that anchors the string that we are trimming off.

This delimiting character will be placed to the right of the * when used with # or ##, and it will be placed to the left of * when used with % or %%. You will notice that the / and . characters obey this simple, logical placement rule in the code above. In both cases, the anchoring delimiter is also trimmed off.

To summarize:

1. When we use # or ##, we discard a substring to the left of the anchor; and
2. When we use % or %%, we discard a substring to the right of the anchor.

Minefields to beware of

The pattern-matching capabilities in `bash` throw up unexpected results when the assumptions made above are not fulfilled. The structure of the `fullname` is one such. What happens if our assumptions are false?

Filenames without an extension

There are occasions when, for a variety of reasons, filenames might not have extensions. In such cases, we might rightfully expect the extension to be a null or empty string. But is that what happens in practice? Let us try a simple experiment. You could fire up a bash terminal and run what follows interactively.

```
#!/bin/bash
shopt -s extglob
fullname=$HOME/myPDFfile
ext=”${fullname##*.}”
echo ”Extension is $ext.”
```

The result is:

```
Extension is /home/chandra/myPDFfile.
```

Surely, you did not expect the extension to be the fullname of the file. Yet, that is what we get. Though unexpected, is it yet logically correct?

Imagine you are moving from left to right until you hit the *last* . character. When you do, you discard whatever is to the left of the . along with that character itself. If there is no . character, you do not stop and you do not discard anything. So, you are left with what you started with. *But, although logical, that is not the intent.* The error arises from the unfulfilled assumption that `fullname` contains a dot character, followed by alphanumeric characters that denote the extension.

One way to overcome this issue is to test for a period or dot character in the original `fullname` string [14–16]. If there is no . character, we set the extension to the empty string. Otherwise, we set it to what we get by the pattern-matching we have discussed. The corrected routine for the extension should thus run:

```
#!/bin/bash
# ext.sh
shopt -s extglob

fullname=”/path/myPDFfile”

if [[ "$fullname” =~ \. ]]
then
  echo $?
  ext=”${fullname##*.}”
else
  echo $?
  ext=””
fi
echo ”Extension is $ext.”
```

Note that the `=~` sign is a *regular expression* operator that has been inducted from `perl` into `bash` [17]. The `[[...]]` expression returns a `0` for `true` and a `1` for `false`, which may sound contrary to expectations, but that is the correct behaviour in `bash`. This may be seen by pre-pending `echo $?` to each branch of the `if` conditional.

Moreover, when matching, the left side is a string that should be double quoted to avoid errors, while the right side is a regular expression, or a constant that is either escaped as in `\.`, or in single quotes, like `'.'` [15]. If a plain `.` is used, with no “protection”, it will match any single character in accordance with regex rules, and we risk getting the wrong result. It is attention to every small detail that ensures success with `bash` scripts. You also learn patience on the way. ☺

Filenames with multiple dot characters

I have encountered occasions where the `fullname` of a file contains multiple `.` characters. In such cases, we must adopt a convention that the extension is what occurs to the *right of the rightmost* dot character. We will avoid pathological cases like a filename *ending* with a `.` character. If these additional assumptions hold, our pattern-matching for the extension will return the correct result.

Filenames without a path

Before attempting to extract a `path`, we must check for the presence of a / in the `fullname` string. Otherwise, we risk getting the same errors as with missing extensions. The following script should be self-explanatory by now. Again, note that we either need to make the / character a literal, enclosed by single quotes, or we must escape it with a backslash, \. Note that this time, the forward slash is enclosed by single quotes; I find \/ both inelegant and somewhat perplexing.

```
#!/bin/bash
# path.sh
shopt -s extglob

```

```
fullname=”myPDFfile.pdf”

if [[ ”$fullname” =~ '/' ]]
then
  echo $?
  path=”${fullname%/*}”
else
  echo $?
  path=””
fi
echo ”Path is $path.”
```

The generalized filename parser

The revised file for parsing a filename into its components therefore needs to be augmented with these tests if it is to be robust and general. Note also that if no input is given, there can be no meaningful output; so we have to test that there is at least one command-line argument. The file `parsefilename.sh` embodies the improvements we have discussed and is made available here for completeness. The usual disclaimers about software merchantability are implicit! ☺

Prettifying non-standard filenames

We have now concluded the first part of processing a filename, and are ready to proceed to the second part, which is prettifying a filename. Although filenames containing spaces, tabs, and non-alphanumeric characters *can* be processed in Linux—when enclosed by single quotes—the natural etiquette in Linux file naming is not to use such non-standard characters.

But what happens if we are bequeathed files having such names? Renaming them one-by-one, by hand, will be laborious and even impractical. How may we automate the renaming of such files—to result in filenames that are both meaningful and Linux-friendly? That is what will occupy us for the rest of this blog.

From cacophony to harmony

The original file naming convention in Linux is that there will be *no spaces* or other non-alphanumeric characters, *except for the underscore* character _ in a filename[2] [18]. This is because spaces are used as input field separators (IFS) to break up a string into its components: something known as *word splitting* [19].

But not all filenames respect this nomenclature of alphanumeric plus underscore characters alone. What if you encountered a file named so: `El??Condor    _Pasa%^!.mp3`. How would you sanitize it into something that could be easily processed by Linux when supplied as an argument?

[2] The - character (dash or hyphen) assumes many roles: as the standard input and standard output, as a prefix to an option for commands, as a range specifier in regular expressions like [a-z], etc. So, a filename should not start or end with the - character; its position elsewhere in a filename should not cause problems.

This set me developing a simple script to convert all non-compatible characters into acceptable characters so that the end result would be a sanitized, Linux-compatible filename that still retained some of its meaning. Here is my thought process as an algorithm:

1. Retain underscores _ unchanged;
2. Replace every other non-alphanumeric character by a -; and
3. Replace strings of consecutive - characters from the previous step by a single - character.
4. Neither the first nor the last character of the modified filename shall be a - character.

The standard and most obvious way to do this is by using regular expressions and a tool such as `sed` or `awk` or `perl`. Moreover, the POSIX character classes such as `[:space:]`, `[:blank:]`, `[:punct:]` hold the key to concisely including all characters that need to be substituted with dashes. This was the trajectory I followed initially.

Using sed to sanitize a filename

How might the unusual filename (or string)

`El??Condor   _Pasa%^!.mp3`

which contains three spaces, and five punctuation characters, be sanitized using `sed`[3]?

Typically, `sed` works on text within a file. But we may also pass strings to `sed` as literals rather than an input file. Rather than stumble through the tedious path I took to success, I record below the final `sed` one-liner that I assembled. Note that we will henceforth use only the basename of this filename in all examples.

```
sed -r ”s/([[:space:]]|[[:punct:]])+/-/g” «< 'El??Condor   _Pasa%^!'
```

which gives the result:

```
El-Condor-Pasa-
```

Note that *multiple* spaces and punctuation characters have been replaced by *single* hyphens or dashes. The + sign in the expression confers this behaviour. The fact that we want to change *both* spaces and punctuation is the reason for the | alternation sign which might be loosely looked at as a logical OR. The g parameter at the end (for global) means that *all* such occurrences will be substituted. The `[[:space:]]` and `[[:punct:]]` incantations are called POSIX character classes [11]. The option `-r` is given to `sed` to confer the regular expression matching behaviour we are after. And the «< allows an input to be given immediately to `sed` from the command line rather than from a file. The s refers to a substitution.

But, are we satisfied with our result? Not really, on two counts:

1. We want to retain the underscore _ character unchanged, if and when it occurs in the original string. An underscore in our original string has disappeared.
2. We do not want a terminal hyphen in the modified filename, as in this case.

[3] The underscore does not count as a punctuation character because it is not replaced.

The second is easier to fix first. We will resort to the start-of-line anchor ^ and the end-of-line anchor $ to eliminate (possible) initial and terminal hyphens. Since sed can work consecutively on the original string, or its modified variant, we can simply chain the three substitutions using pipes so:

```
sed -E ”s/([[:space:]]|[[:punct:]])+/-/g”  «< 'El??Condor   _Pasa%^!' | \
sed ”s/^-//” | sed  ”s/-$//”
```

to get

```
El-Condor-Pasa
```

almost as desired. The -E option is POSIX-compliant and needed to deal with extended regular expressions; in GNU sed it is synonymous with the -r option. The | character indicates that the input for the second and third sed substitutions is the output from the previous sed command. The + denotes multiple consecutive instances of spaces and punctuation characters and the g denotes performing the substitution globally, i.e., as many times as the conditions require. The // means that there is no replacement character.

The requirement to pass underscores unchanged is more serious, because we need to modify the first sed replacement. Because the [[:punct:]] class also includes the underscore character, it is a sticky business to keep all the underscores but replace every other punctuation symbol by a dash. In fact, it negates the very notion and convenience of a POSIX character class.

What we want is some operation like a set difference, for which the regex syntax is not available for sed. One *could* enumerate all punctuation symbols and exclude only _ from that list, and use that class instead of [[:punct:]], but this approach strikes me as particularly ham-fisted.

A better and more felicitous way is to invert the requirement and *preserve* the alphanumeric and underscore characters alone, and *replace everything else* by a dash:

```
sed -E ”s/([^A-Za-z0-9_])+/-/g” «< 'El??Condor   _Pasa%^!' | \
sed ”s/^-//” | sed  ”s/-$//”
```

which gives:

```
El-Condor-_Pasa
```

Although it looks awkward—having a - followed by a _—this is exactly the desired output given our transformation rules. Note that the ^ character is used in the first line as a *negation* of a character class, and in the second line as a start-of-string anchor. It is this overloading of meanings on a single symbol that leads to difficulties in understanding such expressions.

Can it be done in bash?

But what about the nagging refrain, “Why not do it all in bash itself, using pattern matching”? So, rather than considering how to do this in perl, etc., I hacked my way through several iterations of trying to perform the substitution in bash itself.

Pattern-matching, substitution, and substring removal

The expression [A-Za-z0-9_] has a rather fortuitous abbreviation as a POSIX character class in bash: it is denoted by [[:word:]]. The pattern-matching/replacement expression in bash therefore becomes:

```
#!/bin/bash
shopt -s extglob
#
filename='El??Condor   _Pasa%^!'
#
# The character class `[:word:]` includes all
# alphanumeric characters and the underscore.
# `^[:word:]` is the negation of this condition.
# So, we replace _all_ non-alphanumeric characters
# and non-underscores with the dash.
#
# The `+` sign though placed at the beginning rather than the end
# has the same meaning as in the `sed` expression.
# The `//` after $filename denotes multiple replacements
# just like the terminal `g` in the `sed` substitution expression.
#
newname=”${filename//+([^[:word:]])/-}”
#
# We then trim off the first character in $newname in case
# it begins with a `-` character.
# Nothing happens if the first character is not a `-`.
#
newname=”${newname/#-}”
#
# Next, we trim off the last character in $newname
# if it matches a dash.
# Nothing happens if the last character is not a `-`.
#
newname=”${newname%-}”
echo ”$newname”
```

It bears noting that:

(a) there is no wildcard character * before the - in the first substring expression;

(b) there is no wildcard character * after the - in the second substring expression;

(c) consequently, in both cases, we are matching a *single* initial or terminal- character, with no ill effects if either or both are missing; and

(d) we may assign the possibly truncated variable newname to itself.

We have accomplished what we set out to do with the filename. The absence of the * in the expression newname=”${newname%-}” has morphed the pattern matching and substring removal we used for parsing filenames into a robust, removal of a terminal -, without the need to test if it is the last character in the string. To demonstrate the terseness of this approach, I give below the same operation, with a slightly longer syntax, that is also available to us in bash.

Using substring extraction

The syntax for substring extraction in bash is ${parameter:offset:length} where offset is measured starting from 0 at the extreme left [3, 6, 20].

```
#!/bin/bash
shopt -s extglob
#
filename='El??Condor   _Pasa%^!'
#
# The character class [:word:] includes all
# alphanumeric characters and the underscore.
# ^[:word:] is the negation of this condition.
# So, we replace all non-alphanumeric characters and non-underscores
# with the dash.
#
newname=”${filename//+([^[:word:]])/-}”
#
# Extract the first character in $newname and test if it is `-`.
# If so, remove it; else do nothing.
# Indexing starts with zero.
#
firstchar=”${newname:0:1}”
if [[ ”$firstchar” == '-' ]]
then
  newname=”${newname::-1}”
fi
echo ”$newname”
#
# Extract the last character in $newname and test if it is `-`.
# If so, remove it; else do nothing.
#
lastchar=”${newname: -1}”
if [[ ”$lastchar” == '-' ]]
then
  newname=”${newname::-1}”
fi
echo ”$newname”
```

The points to especially note here are:

(a) The expression `"${newname:0:1}"` denotes the substring of length 1 starting from the beginning of the string `$newname` is obviously the first character in that string. It may also be written as `"${newname::1}"`.

(b) There is a space between the `:` and the `-` in the expression `"${newname: -1}"`. This space is inserted to avoid ambiguity with *another* expression of the form `${parameter:-word}` which has a different function. The `-1` in `"${newname: -1}"` denotes the leftmost character in the string. Another way to write this is as `"${newname:0-1}"` [21].

(c) The final idiom used above is `"${newname::-1}"`, which is shorthand for `"${newname:0:-1}"`. This operation strips off the final character in the variable `"${newname}"`. Because it is positional in nature, rather than the result of a pattern match, we have to test whether the terminal character is indeed a `-`.

(d) We could also have used the `=~` sign for these tests since we are matching a *single* character. Nevertheless, it is better programming discipline to test for equality when dealing with a single character, as it is more specific.

It should be clear that the first version is clearer, less verbose, and less prone the error than the second one.

To explore further

The interested reader is referred to the official documentation online for a comprehensive explanation of the dazzling features of *parameter expansion* [22] and *substring removal* [23] in bash. For an admirable summary of features like parameter expansion, do also visit the clear and comprehensive BashGuide website.

If you are familiar with bash but require an aide-mémoire for some aspect of string matching or manipulation, the section with the heading "Recommended Shell resources" is the best place to start your search [24]. It contains a wealth of authoritative links that will speedily dispel your doubts, not to speak of saving you time.

Feedback

Please email me your comments and corrections.

A PDF version of this article is available for download here:

https://swanlotus.netlify.app/blogs/pattern-matching-in-bash.pdf

References

1. Cameron Newham and Bill Rosenblatt. *Learning the bash shell.* 3rd ed. O'Reilly, 2005.

2. Ryder, Tom. *Bash quick start guide: Get up and running with shell scripting with bash.* Packt, 2018.

3. Mendel Cooper. Parameter substitution. Online. 10 March 2014. [Accessed 4 March 2023]. Available from: https://tldp.org/LDP/abs/html/parameter-substitution.html

4. Mitch Frazier. Pattern matching in bash. Online. 15 April 2019. [Accessed 28 February 2023]. Available from: https://www.linuxjournal.com/content/pattern-matching-bash

5. —. Pattern matching. Online. 26 September 2022. [Accessed 4 March 2023]. Available from: https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

6. Cameron Newham and Bill Rosenblatt. String operators. Online. 26 January 1998. [Accessed 4 March 2023]. Available from: https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch04s03.html

7. Wikipedia. Glob (programming). Online. 15 January 2023. [Accessed 28 February 2023]. Available from: https://en.wikipedia.org/w/index.php?title=Glob_(programming)&oldid=1133836865

8. Gilles Quénot. Unix. Run script across multiple dirs on specific files, where pathname has regex. Online. 5 March 2023. [Accessed 5 March 2023]. Available from: https://unix.stackexchange.com/a/738684/11610

9. —. History of bash globbing. Online. 10 June 2014. [Accessed 5 March 2023]. Available from: https://unix.stackexchange.com/questions/136353/history-of-bash-globbing

10. Krzysztof Kowalczyk. Extended globbing. Online. 30 January 2023. [Accessed 6 March 2023]. Available from: https://www.programming-books.io/essential/bash/extended-globbing-7c7bf6bd68b64f0e919716228ef9f3df

11. Jan Goyvaerts. Regex tutorial—POSIX bracket expressions. Online. 22 November 2019. [Accessed 5 March 2023]. Available from: https://www.regular-expressions.info/posixbrackets.html

12. Zach Gollwitzer. Intro to bash regular expressions. Online. 21 October 2020. [Accessed 5 March 2023]. Available from: https://dev.to/zachgoll/intro-to-bash-regular-expressions-4d2p

13. Abhinav Tiwari. How to write regular expressions?—GeeksforGeeks. Online. 2 March 2023. [Accessed 5 March 2023]. Available from: https://www.geeksforgeeks.org/write-regular-expressions/

14. Lmcanavals. Test if a string has a period in it with bash. Online. 1 February 2013. [Accessed 4 March 2023]. Available from: https://unix.stackexchange.com/a/63374/11610

15. Mark Byers. Test for a dot in bash. Online. 30 April 2010. [Accessed 4 March 2023]. Available from: https://stackoverflow.com/a/2745096

16. Mark Reed. Bash script pattern matching. Online. 22 June 2017. [Accessed 5 March 2023]. Available from: https://stackoverflow.com/a/44688520

17. Kusalananda. Bash test: What does ”=~” do? Online. 17 January 2017. [Accessed 4 March 2023]. Available from: https://unix.stackexchange.com/a/340485/11610

18. Peek, Jerry, Todino, Grace and Strang, John. *Learning the UNIX operating system*. 5th ed. O’Reilly, 2002.

19. Aka Maarten Billemont, lhunath and Aka Greg Wooledge, GreyCat. Word splitting. Online. 23 May 2018. [Accessed 5 March 2023]. Available from: https://mywiki.wooledge.org/WordSplitting?highlight=%28spaces%29%7C%28word%29%7C%28splitting%29

20. Linuxize. How to check if a string contains a substring in bash. Online. 19 July 2019. [Accessed 4 March 2023]. Available from: https://linuxize.com/post/how-to-check-if-string-contains-substring-in-bash/#google_vignette

21. thinker3. How to get the last character of a string in a shell? Online. 9 July 2013. [Accessed 7 March 2023]. Available from: https://stackoverflow.com/a/21635778

22. Chet Ramey. Bash reference manual: Shell parameter expansion. Online. 22 September 2020. [Accessed 5 April 2023]. Available from: https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Parameter-Expansion

23. —. Parameter expansion [bash hackers wiki]. Online. 10 December 2021. [Accessed 5 March 2023]. Available from: https://wiki.bash-hackers.org/syntax/pe#substring_removal

24. —. The bash hackers wiki: Recommended shell resources. Online. 5 March 2023. [Accessed 5 March 2023]. Available from: https://wiki.bash-hackers.org/start#recommended_shell_resources