Pattern Matching and Substitution in `bash`

R (Chandra) Chandrasekhar

2023-02-28 | 2023-03-01

The arcane powers of the `bash` shell

The `bash` shell [15] embodies powerful pattern-matching and substitution capabilities, many of which are relatively unknown [1,7,13,14]. The programs, `sed`, `awk`, `grep`, and `perl` have been traditionally used for matching and manipulating lines and strings in Linux.

But the pattern-matching and string manipulation capabilities of `bash` have grown steadily since version 2.02, which was released in 1998. 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.

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 and other file-related functions.

Extended globbing

Globbing is the unflattering term—an abbreviation for *global*—used to denote an operation to extract files satisfying certain conditions [2,16,20]. It is applicable also to the `bash` command line. For our purposes, it is useful and sometimes mandatory, to set `shopt -s extglob` after the shebang line.

A canonical filename

A canonical filename will comprise these components:

1. a *path* with the forward slash `/` as the separator 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.

`/my_path/is/quite/long/basename.ext` is a canonical filename where the abovenamed elements are as follows:

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

Parsing the filename

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 ${fullname%/*}"

#
# 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 ${fullname##*/}"

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



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

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 through the lens of regular expressions [8,9,19]. 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.*

One way to overcome this issue is to test for a period or dot character in the original fullname string [6,12,18]. 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 into bash [10]. It returns a 0 for true and a 1 for false, which may sound contrary to expectations, but that is the correct behaviour. This may be seen by appending echo $? above to each branch of the if conditional.

Moreover, when matching, the left side is double quoted while the right side is either escaped as in \. or is a literal, like '.' [6]. If a plain . is used, with no "protection", it will natch any 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. In the process, you also learn patience. ☺

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.

```
#!/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. The file `parsefilename.sh` embodies the improvements we have discussed and is available here for completeness.

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 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 meaningful as well as Linux-friendly? That is what will occupy us for the rest of this blog.

From cacophony to harmony

The 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. This is because spaces are used as input field separators (IFS) to break up a string into its components: something known as *word splitting* [5].

But not all files 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?

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 its meaning. Here is my thought process as an algorithm:

1. Replace all non-standard characters with dashes -.
2. Replace consecutive spaces, or other non-alphanumeric characters, by a *single* dash.

3. Retain underscores, _, unchanged.

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 pathological filename (or string) `El??Condor  _Pasa%^!.mp3`—which contains spaces and punctuation—be sanitized using `sed`?

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 [9]. 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.

Now, 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.

The second is easier to fix first. We will resort to the end-of-line anchor $ to identify the terminal hyphen after the first replacement. Since `sed` can work consecutively on the string, or its modified variant, we can simply chain two substitutions using pipes so:

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

to get

```
El-Condor-Pasa
```


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 this second sed substitution is the output from the previous sed command.

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 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 grossly inelegant.

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/-$//"
```

which gives:

```
El-Condor-_Pasa
```

Although it looks awkward—having a - followed by a _—this is exactly the desired output given our transformation rules. One caveat is that this expression will only work with ASCII characters.

Can it be done in bash?

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

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.
#
newname="${filename//+([^[:word:]])/-}"
```



```
#
# We then extract the last character in the string newname
# and check whether it matches the dash character.
# If it does, we strip it off, to get the finalname.
#

lastchar="${newname: -1}"
if [[ "$lastchar" =~ '-' ]]
then
  finalname="${newname::-1}"
fi
echo "$finalname"
```

The syntax for substring extraction in bash is `${parameter:offset:length}` where `offset` is measured starting from `0` at the extreme left [7,11,14].

(a) Note especially the 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.

(b) 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}"`.

To explore further

The interested reader is referred to the official documentation online for a comprehensive explanation of the dazzling features of *parameter expansion* [17] and *substring removal* [3] 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 [4]. It contains a wealth of authoritative links that will speedily dispel your doubts.

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] 2022. Pattern matching. Retrieved 4 March 2023 from https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

[2] —. 2014. History of bash globbing. Retrieved 5 March 2023 from https://unix.stackexchange.com/questions/136353/history-of-bash-globbing

[3] —. 2021. Parameter expansion [bash hackers wiki]. Retrieved 5 March 2023 from https://wiki.bash-hackers.org/syntax/pe#substring_removal

[4] —. 2023. The bash hackers wiki: Recommended shell resources. Retrieved 5 March 2023 from https://wiki.bash-hackers.org/start#recommended_shell_resources

[5] lhunath aka Maarten Billemont and Greg Wooledge aka GreyCat. 2018. Word splitting. Retrieved 5 March 2023 from https://mywiki.wooledge.org/WordSplitting?highlight=%28spaces%29%7C%28word%29%7C%28splitting%29

[6] Mark Byers. 2010. Test for a dot in bash. Retrieved 4 March 2023 from https://stackoverflow.com/a/2745096

[7] Mendel Cooper. 2014. Parameter substitution. Retrieved 4 March 2023 from https://tldp.org/LDP/abs/html/parameter-substitution.html

[8] Zach Gollwitzer. 2020. Intro to bash regular expressions. Retrieved 5 March 2023 from https://dev.to/zachgoll/intro-to-bash-regular-expressions-4d2p

[9] Jan Goyvaerts. 2019. Regex tutorial—POSIX bracket expressions. Retrieved 5 March 2023 from https://www.regular-expressions.info/posixbrackets.html

[10] Kusalananda. 2017. Bash test: What does ”= ” do? Retrieved 4 March 2023 from https://unix.stackexchange.com/a/340485/11610

[11] Linuxize. 2019. How to check if a string contains a substring in bash. Retrieved 4 March 2023 from https://linuxize.com/post/how-to-check-if-string-contains-substring-in-bash/#google_vignette

[12] lmcanavals. 2013. Test if a string has a period in it with bash. Retrieved 4 March 2023 from https://unix.stackexchange.com/a/63374/11610

[13] Mitch Frazier. 2019. Pattern matching in bash. Retrieved 28 February 2023 from https://www.linuxjournal.com/content/pattern-matching-bash

[14] Cameron Newham and Bill Rosenblatt. 1998. String operators. Retrieved 4 March 2023 from https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch04s03.html

[15] Cameron Newham and Bill Rosenblatt. 2005. *Learning the bash shell* (3rd ed.). O’Reilly.

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

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

[18] Mark Reed. 2017. Bash script pattern matching. Retrieved 5 March 2023 from https://stackoverflow.com/a/44688520

[19] Abhinav Tiwari. 2023. How to write regular expressions?—GeeksforGeeks. Retrieved 5 March 2023 from https://www.geeksforgeeks.org/write-regular-expressions/

[20] Wikipedia. 2023. Glob (programming). Retrieved 28 February 2023 from https://en.wikipedia.org/w/index.php?title=Glob_(programming)&oldid=1133836865

