The Deep, Dark Secrets of Bash

By Ben Okopnik

"There are two major products that come out of Berkeley: LSD and UNIX. We don't believe this to be a coincidence." -- *Jeremy Anderson*

Deep within the `bash` man page lurk terrible things, not to be approached by the timid or the inexperienced... Beware, Pilgrim: the last incautious spelunker into these mysterious regions was found, weeks later, muttering some sort of strange incantations that sounded like "nullglob", "dotglob", and "MAILPATH='/usr/spool/mail/bfox?"You have mail":~/shell-mail?"$_ has mail!"" (He was immediately hired by an Unnamed Company in Silicon Valley for an unstated (but huge) salary... but that's beside the point.)

<Shrug> What the heck; I've already gone parasailing and scuba-diving this month (and will shortly be taking off on a 500-mile sail up the Gulf Stream); let's keep living La Vida Loca! :)

Parameter Expansion

The built-in parsing capabilities of `bash` are rather minimal as compared to, say, `perl` or `awk`: in my best estimate, they're not intended for serious processing, just "quick and dirty" minor-task handling. Nevertheless, they can be very handy for that purpose.

As an example, let's say that you need to differentiate between lowercase and capitalized filenames in processing a directory - I ended up doing that with my backgrounds for X, since some of them look best tiled, and others stretched to full-screen size (file size wasn't quite a good-enough guide). I "capped" all the names of the full-sized pics, and "decapped" all the tiles. Then, as part of my random background selector, "bkgr", I wrote the following:

```
fn=$(basename $fnm)                     # We need _just_ the filename
[ -z ${fn##[A-Z]*} ] && MAX="-max"      # Set the "-max" switch if true
xv -root -quit $MAX $fnm &              # Run "xv" with|without "-max"
                                        #   based on the test result
```

Confusing-looking stuff, isn't it? Well, part of it we already know: the [-z ...] is a test for a zero-length string. What about the other part, though? In order to 'protect' our parameter expansion result from the cold, cruel world (e.g., if you wanted to use the result as part of a filename, you'd need the 'protection' to keep it separate from the

other characters), we use curly brackets to surround the whole enchilada. **$d** is the same as **${d}** except that the second variety can be combined with other things without losing its identity - like so:

```
d=Digit
echo ${d}ize        # "Digitize"
echo ${d}al         # "Digital"
echo ${d}s          # "Digits"
echo ${d}alis       # "Digitalis"
```

Now that we have it isolated from the world, friendless and all alone... oops, sorry - that's "_shell_ script", not "horror movie script" - I lose track once in a while... Anyway, now that we've separated the variable out via the curly braces, we can apply a few tools incorporated in `bash` (capable little bugger, isn't it?) to perform some basic parsing of its value. Here is the list: (For this exercise, let's assume that $parameter="amanuensis".)

${#parameter} - return length of the parameter value.
EXAMPLE: `${#parameter} = 10`

${parameter#word} - cut shortest match from start of parameter.
EXAMPLE: `${parameter#*n} = uensis`

${parameter##word} - cut longest match from start of parameter.
EXAMPLE: `${parameter#*n} = sis`

${parameter%word} - cut shortest match from end of parameter.
EXAMPLE: `${parameter%n*} = amanue`

${parameter%%word} - cut longest match from end of parameter.
EXAMPLE: `${parameter%%n*} = ama`

${parameter:offset} - return parameter starting at 'offset'.
EXAMPLE: `${parameter:7} = sis`

${parameter:offset:length} - return 'length' characters of parameter starting at 'offset'.
EXAMPLE: `${parameter:1:3} = man`

${parameter/pattern/string} - replace single match.
EXAMPLE: `${parameter/amanuen/paralip} = paralipsis`

${parameter//pattern/string} - replace all matches.
EXAMPLE: `${parameter//a/A} = AmAnuensis` (For the last two operations, if the pattern begins with #, it will match at the beginning of the string; if it begins with %, it will match at the end. If the string is empty, matches will be deleted.)

There's actually a bit more to it - things like variable indirection, and parsing arrays - but, gee, I guess you'll just have to study that man page yourself. :) Just consider this as motivational material.

So, now that we've looked at the tools, let's look back at the code -

```
[ -z ${fn##[A-Z]*} ]
```

Not all *that* difficult anymore, is it? Or maybe it is; my thought process, in dealing with searches and matches, tends to resemble pretzel-bending. What I did here - and it could be done in a number of other ways, given the above tools - is to match for a max-length string (i.e., the entire filename) that begins with an uppercase character. The [-z ...] returns 'true' if the resulting string is zero-length (i.e., matched the [A-Z]* pattern), and $MAX is set to "-max".

Note that, since we're matching the entire string, ${fn%%[A-Z]*} would work just as well. If that seems confusing - if _all_ of the above seems confusing - I suggest lots and lots of experimentation to familiarize yourself with it. It's easy: set a parameter value, and experiment, like so -

```
Odin:~$ experiment=supercallifragilisticexpialadocious
Odin:~$ echo ${experiment%l*}
supercallifragilisticexpia
Odin:~$ echo ${experiment%%l*}
superca
Odin:~$ echo ${experiment#*l}
lifragilisticexpialadocious
Odin:~$ echo ${experiment##*l}
adocious
```

...and so on. It's the best way to get a feel for what a certain tool does; pick it up, plug it in, put on your safety glasses and gently squuueeeze the trigger. Observe all safety precautions as random deletion of valuable data may occur. Actual results may vary and *will* often surprise you.

Parameter State

There are times - say, in testing for a range of error conditions that set different variables - when we need to know whether a specific variable is set (has been

assigned a value) or not. True, we could test it for length, as I did above, but the utilities provided by `bash` for the purpose provide convenient shortcuts for such occasions: (Here, we'll assume that our variable - $joe - is unset or null.)

${parameter:-word} - If parameter is unset, "word" is substituted.
EXAMPLE: `${joe:-mary} = mary` ($joe remains unset.)

${parameter:=word} - If parameter is unset, set it to "word" and return it.
EXAMPLE: `${joe:=mary} = mary` ($joe="mary".)

${parameter:?word} - Display "word" or error if parameter is unset.
EXAMPLE:

```
Odin:~$ echo ${joe:?"Not set"}
bash: joe: Not set
Odin:~$ echo ${joe:?}
bash: joe: parameter null or not set
```

${parameter:+word} - "word" is substituted if parameter is not unset.
EXAMPLE:

```
Odin:~$ joe=blahblah
Odin:~$ echo ${joe:+mary}
mary
Odin:~$ echo $joe
blahblah
```

Array Handling

Another built-in capability of `bash`, a basic mechanism for handling arrays, allows us to process data that needs to be indexed, or at least kept in a structure that allows individual addressing of each of its members. Consider the following scenario: if I have a phonebook/address list, and want to send my latest "Sailor's Newsletter" to everyone in the "Friends" category, how do I do it? Furthermore, say that I also want to create a list of names of the people I sent it to... or some other processing... i.e., make it necessary to split it up into fields by length, and arrays become one of the very few viable options.

Let's look at what this might involve. Here's a clip of a notional phonebook to be used for the job:

Name	**Category**	**Address**	**e-mail**
Jim & Fanny Friends	Business	101101 Digital Dr. LA CA	fr@gnarly.com
Fred & Wilma Rocks	friends	12 Cave St. Granite, CT	shale@hill.com

```
Joe 'Da Fingers' Lucci   Business  45 Caliber Av. B-klyn NY tuff@ny.org
Yoda Leahy-Hu            Friend    1 Peak Fribourg Switz.   warble@sing.ch
Cyndi, Wendi, & Myndi    Friends   5-X Rated St. Holiday FL 3cuties@fl.net
```

Whew. This stuff obviously needs to be read in by fields - word counting won't do; neither will a text search. Arrays to the rescue!

```
#!/bin/bash

# 'nlmail' sends the monthly newsletter to friends listed
# in the phonebook


# bash would create the arrays automatically, since we'll
# use the 'name[subscript]' syntax to load the variables -
# but I happen to like explicit declarations.

declare -a name category address email

# Count the number of lines in "phonelist" and loop that
# number of times

for x in $(seq $(grep -c $ phonelist))
do
    x=$(($x))                           # Turns '$x' into a number
    line="$(sed -n ${x}p phonelist)"    # Prints line number "$x"
    name[$x]="${line:0:25}"             # Load up the 'name' variable
    category[$x]="${line:25:10}"        #  Etc.,
    address[$x]="${line:35:25}"         #       etc.,
    email[$x]="${line:60:20}"           #            etc.
done
# Continued below ...
```

At this point, we have the "phonelist" file loaded into the four arrays that we've created, ready for further processing. Each of the fields is easily addressable, thus making the stated problem - that of e-mailing a given file to all my friends - a trivial one (this snippet is a continuation of the previous script):

```
# Continued from above ...
for y in $(seq $x)
do
    # We'll match for the word "friend" in the 'category' field,
    # make it "case-blind", and clip any trailing characters.

    if [ -z $(echo ${category[$y]##[Ff]riend*}) ]
    then
        mutt -a Newsletter.pdf -s 'S/V Ulysses News, 6/2000' ${email[$y]}
        echo "Mail sent to ${name[$y]}" >> sent_list.txt
    fi
done
```

That should do it, as well as pasting the recipients' names into a file called "sent_list.txt" - a nice double-check feature that lets me see if I missed anyone.

The array processing capabilities of `bash` extend a bit beyond this simple example. Suffice it to say that for simple cases of this sort, with files under, say, a couple of hundred kB, `bash` arrays are the way to go. For my own curiosity, I created a list of names that was just over 100kB, using the "phonelist" from the above example -

```
for n in $(seq 300); do cat phonelist >> ph_list; done
```

- and ran it on my aging Pentium 233/64MB. 24 seconds; not bad for 1500 records and a "quick and dirty" tool.

Note that the above script can be easily generalized - as an example, you could add the ability to specify different phone-lists, criteria, or actions, right from the command line. Once the data is broken up into an easily-addressable format, the possibilities are endless...

Wrapping It Up

`bash`, besides being very capable in its role as a command-line interpreter/shell, boasts a large number of rather sophisticated tools available to anyone that needs to create custom programs. In my opinion, shell scripting suits its niche - that of a simple yet powerful programming language - perfectly, fitting between command-line utility usage and full-blown (C, Tcl/Tk, Python) programming, and should be part of every *nix user's arsenal. Linux, specifically, seems to encourage the "do it yourself" attitude among its users, by giving them access to powerful tools and the means to automate their usage: something that I consider a tighter integration (and that much higher a "usability quotient") between the underlying power of the OS and the user environment. "Power to the People!" :)

Until next month -
Happy Linuxing!

Quote of the Month

"...Yet terrible as UNIX addiction is, there are worse fates. If UNIX is the heroin of operating systems, then VMS is barbiturate addiction, the Mac is MDMA, and MS-DOS is sniffing glue. (Windows is filling your sinuses with lucite and letting it set.)

You owe the Oracle a twelve-step program."
--*The Usenet Oracle*

References

The "man" pages for `bash`, `builtins`