How can I use parameter expansion? How can I get substrings? How can I get a file without its extension, or get just a file's extension?

See FAQ 100 for a string manipulation tutorial; this page contains some of the same information, but in a more concise format.

Parameter Expansion covers the substitution of a variable or a special parameter by its value. There are various operations that can be performed on the value at the same time for convenience. The full set of capabilities can be found in the bash manpage, or see the reference or the bash hackers article. It can be hard to understand parameter expansion without actually using it, so we're going to begin with some examples.

The first set of capabilities involves *removing a substring*, from either the beginning or the end of a parameter. Here's an example using parameter expansion with something akin to a hostname (dot-separated components):

```
parameter     result
-----------   ------------------------------
${NAME}       polish.ostrich.racing.champion
${NAME#*.}           ostrich.racing.champion
${NAME##*.}                         champion
${NAME%%.*}   polish
${NAME%.*}    polish.ostrich.racing
```

And here's an example of the parameter expansions for a typical filename:

```
parameter     result
-----------   ------------------------------------------------------
--
${FILE}       /usr/share/java-1.4.2-
sun/demo/applets/Clock/Clock.class
${FILE#*/}     usr/share/java-1.4.2-
sun/demo/applets/Clock/Clock.class
${FILE##*/}
Clock.class
${FILE%%/*}
${FILE%/*}    /usr/share/java-1.4.2-sun/demo/applets/Clock
```

US keyboard users may find it helpful to observe that, on the keyboard, the "#" is to the left of the "%" symbol. Mnemonically, "#" operates on the left side of a parameter, and "%" operates on the right. The glob after the "%" or "%%" or "#" or "##" specifies what pattern to *remove* from the parameter expansion. Another mnemonic is that in an English sentence "#" usually comes before a number (e.g., "The #1 Bash reference site"), while "%" usually comes after a number (e.g., "Now 5% discounted"), so they operate on those sides.

You cannot nest parameter expansions. If you need to perform two expansions steps, use a variable to hold the result of the first expansion:

```
# foo holds: key="some value"
bar=${foo#*=\"} bar=${bar%\"*}
# now bar holds: some value
```

Here are a few more examples (but *please* see the real documentation for a list of all the features!). I include these mostly so people won't break the wiki again, trying to add new questions that answer this stuff.

```
${string:2:1}   # The third character of string (0, 1, 2 = third)
${string:1}     # The string starting from the second character
                # Note: this is equivalent to ${string#?}
${string%?}     # The string with its last character removed.
${string: -1}   # The last character of string
${string:(-1)}  # The last character of string, alternate syntax
                # Note: string:-1 means something entirely
different; see below.

${file%.mp3}    # The filename without the .mp3 extension
                # Very useful in loops of the form: for file in
*.mp3; do ...
${file%.*}      # The filename without its extension (assuming there
was
                # only one extension in the first place...).
${file%%.*}     # The filename without all of its extensions
${file##*.}     # The extension only.
```

Bash 4

Bash 4 introduces some additional parameter expansions; toupper (^) and tolower (,).

```
# string='hello, World!'
parameter     result
-----------   -----------------------------------------------------
--
${string^}    Hello, World! # First character to uppercase
${string^^}   HELLO, WORLD! # All characters to uppercase
${string,}    hello, World! # First character to lowercase
${string,,}   hello, world! # All characters to lowercase
```

Parameter Expansion on Arrays

BASH arrays are remarkably flexible, since they are so well integrated with the other shell expansions. Virtually any expansion you can carry out on a scalar can equally be applied to a whole array. Remember that quoting an array expansion using @ (e.g. `"$@"` or `"${cmd[@]}"`) results in the members being treated as individual words, regardless of their content. So for example, `arr=("${list[@]}" foo)` correctly handles all elements in the `list` array.

First the expansions:

```
$ a=(alpha beta gamma)  # our base array
$ echo "${a[@]#a}"      # chop 'a' from the beginning of every
member
lpha beta gamma
$ echo "${a[@]%a}"      # from the end
alph bet gamm
$ echo "${a[@]//a/f}"   # substitution
flphf betf gfmmf
```

The following expansions (substitute at beginning or end) are very useful for the next part:

```
$ echo "${a[@]/#a/f}"   # substitute a for f at start
flpha beta gamma
$ echo "${a[@]/%a/f}"   # at end
alphf betf gammf
```

We use these to prefix or suffix every member of the list:

```
$ echo "${a[@]/#/a}"    # append a to beginning
aalpha abeta agamma     #    (thanks to floyd-n-milan for this)
$ echo "${a[@]/%/a}"    # append a to end
alphaa betaa gammaa
```

This works by substituting an empty string at beginning or end with the value we wish to append.

So finally, a quick example of how you might use this in a script, say to add a user-defined prefix to every target:

```
$ PFX=inc_
$ a=("${a[@]/#/$PFX}")
$ echo "${a[@]}"
inc_alpha inc_beta inc_gamma
```

This is very useful, as you might imagine, since it saves looping over every member of the array.

The special parameter `@` can also be used as an array for purposes of parameter expansions:

```
${@:(-2):1}             # the second-to-last parameter
${@: -2:1}              # alternative syntax
```

You can't use `${@:-2:1}`, however, because that collides with the syntax in the next section.

Portability

The original Bourne shell (7th edition Unix) only supports a very limited set of parameter expansion options:

```
${var-word}             # if var is defined, use var; otherwise,
"word"
${var+word}             # if var is defined, use "word"; otherwise,
nothing
${var=word}             # if var is defined, use var; otherwise, use
"word" AND...
                        #   also assign "word" to var
${var?error}            # if var is defined, use var; otherwise
print "error" and exit
```

These are the only completely portable expansions available.

POSIX shells (as well as KornShell and BASH) offer those, plus a slight variant:

```
${var:-word}             # if var is defined AND NOT EMPTY, use var;
otherwise, "word"
similarly for ${var:+word} etc.
```

POSIX, Korn (all versions) and Bash all support the `${var#word}`, `${var%word}`, `${var##word}` and `${var%%word}` expansions.

ksh88 does not support `${var/replace/with}` or `${var//replace/all}`, but ksh93 and Bash do.

ksh88 does not support fancy expansion with arrays (e.g., `${a[@]%.gif}`) but ksh93 and Bash do.

Examples of Filename Manipulation

Here is one Posix-compliant way to take a full pathname, extract the directory component of the pathname, the filename, just the extension, the filename without the extension (the "stub"), any numeric portion occurring at the end of the stub (ignoring any digits that occur in the middle of the filename), perform arithmetic on that number (in this case, incrementing by one), and reassemble the entire filename adding a prefix to the filename and replacing the number in the filename with another one.

```
FullPath=/path/to/name4afile-00809.ext   # result:   #
/path/to/name4afile-00809.ext
Filename=${FullPath##*/}                             #   name4afile-
00809.ext
PathPref=${FullPath%"$Filename"}                     #   /path/to/
FileStub=${Filename%.*}                              #   name4afile-
00809
FileExt=${Filename#"$FileStub"}                      #   .ext
FnumPossLeading0s=${FileStub##*[![:digit:]]}         #   00809
FnumOnlyLeading0s=${FnumPossLeading0s%%[!0]*}        #   00
FileNumber=${FnumPossLeading0s#"$FnumOnlyLeading0s"} #   809
NextNumber=$(( FileNumber + 1 ))                     #   810
FileStubNoNum=${FileStub%"$FnumPossLeading0s"}       #   name4afile-
NewFullPath=${PathPref}New_${FileStubNoNum}${FnumOnlyLeading0s}${Nex
tNumber}${FileExt}
                        # Final result is:           #
/path/to/New_name4afile-00810.ext
```

Note that trying to get the directory component of the pathname with `PathPref="${FullPath%/*}"` will fail to return an empty string if `$FullPath` is `"SomeFilename.ext"` or some other pathname without a slash. Similarly, trying to get the file extension using `FileExt="${Filename#*.}"` fails to return an empty string if `$Filename` has no dot (and thus no extension).

Also note that it is necessary to get rid of leading zeroes for `$FileNumber` in order to perform arithmetic, or else the number is interpreted as octal. In the example above, trying to calculate `$(( FnumPossLeading0s + 1 ))` results in an error since "00809" is not a valid number. If we had used "00777" instead, then there would have been no error, but `$(( FnumPossLeading0s + 1 ))` would result in "1000" (since octal 777 + 1 is octal 1000) which is probably not what was intended. See ArithmeticExpression.

Quoting is not needed in variable assignment, since WordSplitting does not occur. On the other hand, variables referenced inside a parameter expansion need to be quoted (for example, quote $Filename in PathPref=${FullPath%"$Filename"}) or else any * or ? or other such characters within the filename would incorrectly become part of the parameter expansion (for example, if an asterisk is the first character in the filename --try FullPath="dir/*filename").

The example above fails to compensate if the result of the arithmetic operation, `$NextNumber`, has a different number of digits than the original. If `$FullPath` were "filename099" then `$NewFullPath` would have been "New_filename0100" with the filename being one digit longer.