Part 3: String Processing and Regular Expressions
SCRAPS OF WORDS

MIRKO DÖLLE

We laid down the fundamentals of character strings in the last issue. This time though, we will not be settling for simple dry runs or length specifications, but will be mixing our strings vigorously together.

Let's start with separating parts of a string, let's say *Hello, you beautiful world*. Even if some of you might not necessarily wish to subscribe to this point of view, we are going to expand it a bit more. Firstly we break down our phrase:

```
#!/bin/bash
set="Hello, you beautiful world"
echo "${set:0:5} ${set:21}"
```

The result is our old friend 'Hello world', which we have derived from our phrase with the command "${variable:offset:length}". This command delivers a section (**Substring**) of the specified character string.

The offset states how many characters from the beginning we are starting. In the case of Hello it starts with the first character, thus with an offset of zero, while to reach the first letter of world, we have to leap over 16 characters – hence, here we have an offset of 16. If you like, you can also count back from the end of a string, as in the following example:

```
#!/bin/bash
set="Hello, you beautiful world"
echo "${set:0:5} ${set: -5}"
```

Whenever you specify a negative offset, the count is done forwards from the end. But be careful with colons and minus signs: If these are not separated by a blank space, Bash recognises the command "${variable:-string}", which we discussed in the last instalment of Programming Corner.

In the example, we have come across both forms of application of the substring function. Variable and offset must always be specified; without the length specification the substring will be derived from the specified position to the end. Knowing this, we are now able to convert:

```
#!/bin/bash
set="Hello, you beautiful world"
echo "${set:0:5} ${set: -5}${set:5:2} [you]
are so ${set:11:9}."
```

One great advantage of this method is that we always have to know the position of the words – if we were to use Hi instead of Hello, our whole program would go into a spin. In fact we really want to swap whole words and not just letters, so we have to recognise the limits of words.

Breaking down into words

To do this, we can use the special variable mentioned in the last issue, *IFS* – this includes all the characters which are to apply as separators between program names and parameters. The standard rule is that these are blanks, tabulator and Enter. For our purposes, only the blank space is the required separator between two words. The following program separates our phrase into words using blank spaces and stores each word in a different variable. The result is then the same as in the previous listing, i.e. our converted phrase:

```
#!/bin/bash
IFS=" "
set="Hello, you beautiful world"
set -- $set
echo "$1 $4, $2 [you] are so ${3:0:5}."
```

The fourth line is new, the *set* command. Without specifying parameters it lists all variables which have been set. Otherwise *set* can be used to change various settings for the run-time of Bash. Here we are using the third domain of application: We are filling our parameter variables (**positional parameters**), which we would not otherwise be able to alter, with new values.

The double minus characters as first parameter are important. *set* has its own range of options, which all begin with a minus sign followed by a letter. The double minus is defined as the end character of all options, which means that nothing following it can be a parameter. Any minus signs that may occur in our text are thus not misinterpreted.

Using the variable IFS we have told *set* that parameters are separated by blank spaces. This means that for *set* every individual word is a parameter; these are filed in sequence in our special variables from *$1* on upwards. By the way, it is not possible to fill only individual ones of these variables or to start with a specified number.

Bite-size arrays

Now we can try out the other action of the command *${variable:offset:length}*, namely in connection with arrays. To do this, we are using the array *$**, which contains all parameter variables such as *$0*, *$1* and *$2*:

```
#!/bin/bash
set -- Hello world - you are so beautiful.
echo ${*:1:2}
```

The result is our old friend, 'Hello world'. When using arrays we obtain a portion of the elements

Substring: *Part of a character string.*
Offset: *Displacement with respect to the zero point or start. There can basically be positive and negative offsets, negative values being counted back from the end.*
Positional parameters: *In the variables $0, $1, $2 etc. all parameters of a program command are stored separately. Nothing can be assigned to them directly – reloading is possible only with set.*

instead of, as with character strings, a part of the letters. In the example we start with element number 1 and fish out a total of two elements. Arrays normally start at element 0. But the program name of Bash is in *$0*, so we find the first element of the parameter variables, exceptionally, in *$1*.

Dirname and basename are home-built

Now let's turn to a new example, and separate the absolute path and the file name from each other – "/usr/X11R6/bin/X" should turn into "/usr/X11R6/bin" and "X". There are ready-made programs to do this, *dirname* delivers us the directory and *basename* the file name. Now you need to copy both commands using the resources of Bash. To do this, we can apply the method of substring function we have just learnt to arrays:

```
001  #!/bin/bash
002  IFS="/"
003  set -- $1
004  file="${*: -1:1}"
005  IFS=" "
006  set -- ${*:1:$[$#-1]}
007  IFS="/"
008  directory=/"$*"
009  IFS=" "
010  echo directory: $directory
011  echo file: $file
```

This listing needs some explanation, even if no new commands occur in it – a bit of thought has gone into these eleven lines, which you will not immediately grasp at first glance. This is a good example of a listing requiring exhaustive documentation.

Path separation with IFS

At this point we are going to make use of the special variable *IFS*, whose content will be interpreted as separator between program name and parameters. In the second line, we set *IFS* as equal to slash – now the parameters of a program command are no longer separated from each other by blank spaces, but by slashes. This is also where the unusual notation of the third line comes from: The command *set* receives, as first parameter, the double minus, and after this comes our path specification. The slashes seen there are interpreted as separators between the parameters, so that we then have *$1* to *$5*, where *$1* is empty.

The trick for determining the file name is that it must stand immediately behind the last slash – so it is in the last parameter variable. In line four it is stored under *file*. We still have to define the directory: This is written – even if without slashes – in the previous parameter variable. So we need them all but the last one. To get rid of the variable with the file name, we set *IFS* on the blank space. Line six now provides us with a part of our parameter variables, namely the first to the second-to-last ones ("$#" is the number of parameter variables, 1 is deducted from it) separated by spaces, and stores them afresh using *set* under *$1* to *$3*.

Constructing a path specification

Now we want the directory name separated, not with blank spaces, but in the usual way with slashes. To do this we exploit the fact that *"$*"* delivers all parameters – in our case *$1* to *$3* – one after the other, separated by the first characters from *IFS*.

We set *IFS* as equal to slash once again and with line eight we get back the readable directory name. But in the conversion the first slash has gone astray, which is why we must place it in front. Line nine is still very important, even if it is not apparent at first glance. If the slash was in line 10 in *IFS*, then *echo* would be given back our path (also separated by slashes) as individual parameter – and *echo* always displays parameters separated by spaces.

Pattern recognition

The final example serves not only as consolidation but is also intended to show you how to make life considerably easier for yourself by using the right commands. The following listing produces exactly the same result as the previous one:

```
#!/bin/bash
directory="${1%/*}"
file="${1##*/}"
echo directory: $directory
echo file: $file
```

Now we are dealing with two new commands, *${variable%pattern}* and *${variable##pattern}*. Both work by **pattern matching**. The pattern in line two is "/*" and stands for a character string, which starts with a slash and is then followed by as many characters as you like. In this case, as many as you like means: from none up to an infinite number. Bash has additional control symbols for pattern matching, so-called **wildcards** – the most important ones are the question mark and star. The question mark stands for any character at all, while the star covers as many characters as you like.

The command itself searches the variable from back to front and checks when the pattern matches for the first time – thus when, seen from the back, a slash occurs and either nothing or something is behind it. Then it deletes the pattern it has just found, meaning the slash and any characters behind this are removed.

We use this to get the directory. We know that the program name stands directly after the last slash, and have to remove this to get the directory. The last slash itself is – as in the previous example – also removed.

To get the file name, we must do the exact opposite: We remove everything as far as the last

pattern matching: *pattern comparison, where a pattern consisting of* ***wildcards****, special characters and normal characters is compared with a character string.*
wildcards: *Joker characters, for example question mark and star. These stand for any character or as many characters as you like. These can be used to form complex patterns and* ***regular expressions****.*

slash. To do this we have turned the pattern around, we are now seeking any number of characters, behind which a slash stands. The command *${variable##pattern}* searches from the front, beginning after the first match of the pattern. The effect of the double hash is that it is not satisfied with the first match, but tries to remove as much as possible at once. In the example of *path=/usr/X11R6/bin/X*, *${Path#*/}* would settle for "/" (the star here stands for no characters, followed by the slash), while *${Path##*/}* greedily removes "/usr/X11R6/bin/" (the star stands for "/usr/X11R6/bin", followed by the slash).

Pattern comparison from back and front followed by removal comes in the following variants: a moderate (${variable%pattern} and ${variable#pattern}) and a greedy (${variable%%pattern} and ${variable##pattern}) variant.

But back to our program. In the second line we get the directory in which, searching from the back, we want to have the pattern "/*" removed. In our case, thus "/X" is deleted, leaving "/usr/X11R6/bin". In line three we use the greedy method and search from the beginning to have the pattern "*/" deleted – "/usr/X11R6/bin/" is thereby excepted, which leaves "X" as the file name.

Regular expressions

There is also a third way of separating the file name from the directory, using **regular expressions**:

```
#!/bin/bash
file="${1//#*\//}"
directory="${1/%\/[^\/]//}"
echo directory: $directory
echo file: $file
```

The second line is the greedy version of the "Search/Replace" command, usually written as *${variable//searchpattern/replacement}*. Our search pattern is "#*V", which looks complicated at first glance. The hash at the beginning means that the following pattern must appear at the start of the variable. The star stands for any sequence of characters, and "V" is nothing but a protected slash – otherwise it would be misinterpreted as the start of the replacement.

So the search pattern "#*/" is effective for any character string followed by a slash, which must stand at the start of the variable. Since this is the greedy version of the command, in "/usr/X11R6/bin/X" it lays claim to the character sequence "/usr/X11R6/bin/". The pattern found will now be exchanged for the replacement – in our case, this is empty (two slashes in succession), hence the matching substring is removed. This leaves "X", the file name.

The next line provides us with the directory. A search is made for the pattern "%V[^V]", where again the slashes protected by backslashes can be seen – the simplified form of the search pattern is "%/[^/]". The percentage character means that the following pattern must stand at the end of the variable, in order to match. In the square brackets there is a series of characters which may occur as an alternative – "[123]" means that 1, 2 or 3 matches at this point. In our case a caret and the slash stand in the brackets, but the caret itself has a special position. If it stands at the beginning, the following listed characters must *not* come before the brackets. "[^/]" thus means *all characters as far as the slash*. The whole pattern together thus matches a character string beginning with a slash, then contains any characters (with the exception of an additional slash) and stands right at the end of the variable.

This notation, which is admittedly not very enlightening, becomes necessary because when doing a Search/Replace, evaluation always starts from the beginning. A "/*" would have encompassed"/usr/X11R6/bin/X" in the third line, despite using the moderate method, and the result would have been to leave an empty string. It was only by knowing that the name of the program stands after the last slash, so it contains any characters with the exception of the slash, that we were able to solve this case by search/replace. ■

***Regular expression:** Also called "regex" for short, is the generic term for pattern. Text patterns are described by a regular expression, almost like a little programming language. This means regular expressions can be used for both searching and also replacing text patterns.* ■

Commands for string processing

${#variable}	Length of the variable in characters.
${variable:?string}	outputs string, when variable is empty or does not exist.
${variable:-string}	Result is string, when variable is empty or does not exist, otherwise variable is sent back.
${variable:=string}	string is assigned variable, when variable is empty or does not exist, otherwise variable is sent back.
${variable:+string}	Result is string, if variable exists and is not empty, otherwise nothing is sent back.
${variable:offset}	Delivers the content of variable from position offset to the end. If variable is an array, all elements from offset to the end of the array are given back.
${variable:offset:length}	Delivers length characters of the content of variable from position offset. If variable is an array, length elements from element offset are returned.
${variable:#pattern}	Removes the smallest matching pattern from variable (moderate). Searches from front to back.
${variable:##pattern}	Removes the largest matching pattern from variable (greedy). Searches from front to back.
${variable:%pattern}	Removes the smallest matching pattern from variable (moderate). Searches from back to front.
${variable:%%pattern}	Removes the largest matching pattern from variable (greedy). Searches from back to front.
${variable/pattern}	Searches variable from front to back and removes the first matching pattern.
${variable//pattern}	Searches variable from front to back and removes all matching patterns.
${variable/pattern/string}	Searches variable from front to back and replaces the first matching pattern with string.
${variable//pattern/string}	Searches variable from front to back and replaces all matching patterns with string.