Bash: Strings and Arrays

Prof. Dr. Peter Braun

15. Oktober 2024

1 Titlepage

Bash: Strings and Arrays

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Goals of this Unit

Goals of this Unit

- You learn how to use regular expressions
- You learn how to use strings in shell scripts
- You learn how to use arrays in shell scripts

Welcome to this unit, in which we will continue looking at the programming of shell scripts. Today's focus is on dealing with texts and the variables in which texts are stored. You have already noticed that dealing with text documents is a common use case for scripts. We will, therefore, start by looking at the concept of regular expressions in more detail. With such regular expressions, we can find places in a text document or extract information from a text document using patterns. We will then deal with variables and, in particular, variables in which character strings are stored. These can be, for example, words found in a text document. Or, and this is also a very common use case, simply the names of files. File names, for example, often need to be changed. We search for and replace certain parts of file names, and there are special commands in shell programming that can be used to describe such operations in an extremely concise way. And finally, let's look at how to deal with arrays in scripts.

3 Regular Expressions

Regular Expressions

- Formal language to describe character sequences
- Often used in scripts to find files
- Also used to find lines in text files

Regular expressions are a formal language within computer science that we can use to describe sequences of characters. We use a regular expression to describe a pattern we can search for in a text document. Imagine, for example, that we have a text document with all the words in the German language, and we now want to know which words start with the letter A and end with an E. We can translate this pattern into a regular expression and then use Linux commands to search for this pattern in a text document with all the words. Imagine the case of wanting to delete all files with a certain file extension in a directory. We translate this requirement that a file should end with a specific suffix into a regular expression, search for this pattern using a particular command, and then call the command to delete files. With the help of regular expressions and the possibility of concatenating Linux commands, this task can be solved in just one line.

4 Elements of Regular Expressions

Elements of Regular Expressions

- Matching character literal: `a`
- Matching character literals: `hello`
- Matching every character `.`
- Matching specific characters: `[acf]`
- Matching specific ranges: `[a-e]`

A regular expression, therefore, describes a pattern for a sequence of characters. If, when checking this pattern in a text document or a character string, a place is found where this pattern matches, then we call this a "match."Regular expressions can best and most simply be explained by starting with the elementary building blocks. In other words, the smallest patterns that can be described with a regular expression. Once you understand these elementary building blocks, you can combine them to form larger regular expressions. Simple or short regular expressions are still easy to understand. But larger or longer regular expressions are practically unreadable. When you use regular expressions in your scripts, you should, therefore, get into the habit of always explaining the meaning of the expression in colloquial language. Let's start with the smallest elementary building blocks. The regular expression that describes exactly one letter, in our example, the lowercase A, can be seen in the list next to me in the first position. You write the lowercase A. If you were to check this expression against the German word "Hallo," the result would be, yes, that fits. I have found the letter A in the word "hallo." If you were to check this regular expression against the English word "hello," the result would be, no, it doesn't match. You can now check not only one letter but also any sequence of letters. You can see this example in the second line next to me. So, the English word "Hello" would fit this regular expression, but not the German word "hallo." The third building block in this row consists of a dot. This dot stands for all characters. The dot is not to be understood as the dot but as a placeholder for all characters. These types of special characters are also known as meta symbols. In the next line, you can then see an example of a regular expression in which we allow the letters A, C, and F. And if you want to check not only individual characters but practically a series of consecutive characters, you can also abbreviate this with the syntax in the last example next to me.

This regular expression, therefore, always matches if the input contains any character from A to E. Let's look at how to use regular expressions on the command line in Linux with the "egrep" command.

5 Screencast 1

Screencast 1

- mit egrep Beispiele zeigen für Matchen von Zeilen in einer Datei
- und auch mit find und dateinamen

I would like to demonstrate using this grep or egrep command. And we will have a look at the egrep command which is a version of grep that is able to process extended regular expressions. So as you can see here on the main page we have a lot of tools in this let's say family of grep like grep and egrep and fgrep and they all have special features and we will use egrep for this extended regular expressions. I would like to show some tests from the slide. I will use here this text file which contains hello world and one benefit of this egrep command is that it produces a nice looking output so when it matches a pattern the characters are shown in red color here and you have also a lot of configuration possibilities here. You can search for a pattern or you can search for the opposite of a pattern and so on. So the first thing I would like to show is we are searching for a very simple character match. So we are looking for the capital H here in this text file and you can see this is now printed in red color and we can show the opposite by searching for a character that is not part of this file which is the lowercase h and you can see there is no output. We can also search for a complete word here and then the word is printed in red color and we can search for the full stop which matches every character as a regular expression so all the characters are now printed in red color. We can also use a group here so we use square brackets and we look for the A and the E and then these two characters are printed in red color and we can also search for a range with a beginning and an end character so A-E and the result is now the same because randomly there are no other characters from this range.

6 Character Classes

Character Classes

- Digits: `\d`
- Digit, letter, underscore: `\w`
- Whitespace: `\s`
- No digits: `\D`
- No digit, letter or underscore: `\W`
- No whitespace: `\S`

Now, let's move on to a few more meta symbols that we can use to describe classes of characters in a very compact form. The first example in the list next to me represents all digits, i.e., the characters between zero and nine. Instead of this meta symbol \d, we could also have used the notation from the last slide, "zero – nine" in square brackets. The meta symbol \W describes words, whereby words are defined here as all letters, all digits, and the underscore. White space refers to characters that are contained in the text document but are not visible on the screen or are not directly visible. This includes, first of all, the space character, the tabulator, and the character with which we mark the end of a line and a line break. The remaining three examples next to me, in which the letters are written uppercase, always describe exactly the opposite. For example, capital D describes all characters that are not digits. Let's look at a few examples of this live on the computer.

7 Screencast 2

Screencast 2

- todo

I would like to demonstrate the character classes that we introduced on the slide before. And for this character classes we have another text file that has two lines now and we also have not only characters but also digits in this text file and also white space characters. So the space character belongs to the class of white space. And now we have a little problem here because the egrep command that we used in the last video, in the last screencast, this does not work correctly here under Linux when using these character classes. So we must go back to this grep command and we introduce a new option here, dash capital P to enable using these character classes. So we first match for the character class of digits and this only shows one line and this is the line where we have digits and then it marks the digits red. When we go for words it matches both lines. So words match everything here. And the last class that we can test is white spaces and white space characters are only in the first line, not in the second. And of course we cannot mark a space character red here but you can imagine this line is shown because it matches the spaces, the two spaces that are in this line. So now we can invert the classes by using the capital letters, so capital D, show lines without digits and now the second line is shown, that's clear, but the first line is also shown but the digits are not marked red but all the other characters here. Then we have not words, capital W and this matches only the first line because there we have spaces in which are not characters. And the last class is then not white space, so capital S for not white space and this matches both lines because it matches then normal characters and also digits.

8 Quantifiers

Quantifiers

- Matching zero or once: ?
- Matching zero or many: *
- Matching one or many: +

So far, we have only been able to describe regular expressions that check individual characters or fixed sequences of characters. In practice, however, you often want to allow or check whether characters or sequences of characters occur at all or perhaps more than once. Let's look at the first example. The question mark indicates that the regular expression in front of it may only occur once. Let's assume you want to check whether the name of the month of January occurs in a document. However, you want to allow both the short spelling with the three letters JAN and the full name. You could then put the last part of the name January, i.e., everything from the letter U onwards, in round brackets and add a question mark. These letters could, therefore, occur, or they may occur, but they do not necessarily have to occur to find the month of January in the document. With the second example, the asterisk, we describe that the regular expression before it may occur as often as desired. This also includes the case that the regular expression does NOT match. Let's assume you are looking for all lines in a text document that begin with the word hello and then contain any other characters. The regular expression first contains the sequence with the letters from the word hello, followed by a dot, where every character is accepted, and then an asterisk to express that any number of characters may now be added. The last example describes the plus symbol. This is used to require that the regular expression before it may occur at least once but then any number of times. This is needed, for example, to describe words. A word must contain at least one letter but can be any length.

9 Groups

Groups

- Parts of a regular expression can be grouped
- Example: `(\w+)@(\d*)`
- Group 1: the prefix before `@`
- Group 2: the suffix after `@`

If the regular expressions become larger, it makes sense to bring more structure into the expressions. Brackets are used for this, which is very similar to the concept of brackets in arithmetic expressions. Please take a look at the example next to me. This regular expression initially requires a character from the class of characters of type W, i.e., digits, letters, or the underscore. There should be at least one such character or as many as you like. We have just learned the meaning of the plus character. The @ character must follow this. Then, there can be any sequence of digits. The asterisk means that none or any number of characters of this type may appear. With the brackets, we can access the parts of this matching word found separately within our script. This is needed, for example, if you want to extract parts from a text document to process the data found. So you could say, for example, give me everything you saw in the first group and save it in a variable. Let's take another look at the concept live on the computer.

10 Screencast 3

Screencast 3

- todo

To demonstrate this pattern matching using groups I would like to show you another Linux command. The command is named sed and this stands for stream editor. So sed is an editor but not comparable to vi. Vi is an interactive editor where you can see the document and you can modify it. Sed is a non interactive editor where you can only use commands on the command line to modify a text document. The text document is given as a parameter here and then the commands are executed on this document and the output is printed to standard out. And you can then redirect the output for example into a new file. I only show you here some very basic examples because it's only about this concept of groups today. In the later unit you will have the chance to learn sed in more detail. So we want to search. This is now an example for commands. So s stands for search. After the first slash we now write the pattern that we are looking for. So the pattern should start with a capital H. Then we have characters A to Z as a character class here or as a character range. The star stands for any number and we want to replace this by high. And you can see here the result. So when a word starts with capital H the whole word is replaced by high. In the last line nothing happens because there is no word with a capital H in this line. Next we demonstrate how to use groups. So we again start to search for something and now we open a group. And the group is written in parentheses and because this is a valid character we have to escape this. So now we open a group and we copy now the pattern. H as beginning then A to Z as any number. And we finish the group. We end the group. And this is replaced by the group number one itself. So any group that matches is assigned a number and by this backslash one we print the first group. And as you can see nothing has changed here because we search for something but then we print the same in the output. So nothing changes here. And next we are going to change this and we insert an A before and after the group. And now you can see

that SED has really done something with the output. So there was a pattern matching and in the result we have now these lowercase a letters before and after the group that was found. So what was the goal of this demonstration? So what is our goal here in this demonstration? We want to modify this text document and we want to work with this or we want to replace words that start with a capital H. But we want to delete lines that do not have any words with a capital H at the beginning. So we have to modify now the pattern a little bit and we have to use some new flags. It's called flags and options here. So first we extend the pattern. After the group we now allow further characters in this pattern. So after the word there might be now more characters in that line and we only print what we found as a word here. So the first line is clear, hello in German, but not the digits and not the word world. In the second line nothing has changed because it's only one word here. And the third line is still shown. So next we activate a flag called P that prints everything that was matched. And with dash N we omit everything else. So by this we can now get the result that we want. Only the two lines with a capital H words are shown.

11 Alternatives

Alternatives

- Two alternatives are separated by `|`
- Example: `(foo|bar)`

We are almost through with the basic rules for writing regular expressions. What you still need, however, is a way to describe an alternative. For example, you want to search for either one word or the other in a text document. To do this, use the vertical bar.

12 Special Characters

Special Characters

- Beginning of line: ^
- End of line: $
- New line: \n

Last but not least, three special meta-symbols need to be used frequently. Imagine, for example, that you want to search a text file for words that begin with a certain word. You no longer want to find lines where a certain symbol appears somewhere in the line, but it must be at the beginning or end. To do this, use this circumflex symbol, which looks like a roof, to mark the start of the line. Use the $ to mark the end of the line. Often, you also need the special character of backslash and N to mark a line break, for example, if you are looking for a pattern extending over two lines.

13 Exercise 1: Regular Expressions

Exercise 1: Regular Expressions

- Use file `/usr/share/dict/american-english`
- Print all words starting with the letters `ab`
- Print all words ending with `pp`
- Print all words with letter `h` at the second position
- Print all words starting with `a` and `b` or `c` at the second position

Now, we come to the first exercise. If you have Linux installed on your computer, you should find the American English file in the directory next to me. This file contains all the words of the English language. Now, please develop regular expressions to fulfill the following requirements. Firstly, you should display all words on the screen that begin with the letters A and B. Then, display all words that end in PP. In the next exercise, please display all words with an H in the second position. Finally, please display all words that begin with an A and have either a B or C in the second position. When you have finished or want to see the solution, please click the button.

14 Screencast 4

- todo

We first check if the file, this dictionary is available and yes here we can see it. We can also count the number of lines which is equal to the number of words in this file and this is more than 100,000 words here. So we start with the first exercise. We want to display only words that start with a b. So we translate this to a pattern. The circumflex stands for the beginning of a line. Then we have the two characters a b and then we have the dot and any number for the rest of the line. And this gives us this result here. The second exercise was about words that end on double p. So we start with any characters and then at the end pp and the dollar stands for the end of the line. And this number is very small, only six words here. And next we want to show all words that have an h at the second position. So we modify the pattern again. We start with one character, so one full stop here. Then the letter h and any more characters. And we can use more here to display this page by page. And here you can see that all these words have an h at the second position. And the next exercise was about words that start with an a and then have a b or a c at the second position. So we change the pattern again. We start now with an a and now we use a range. So square brackets here and with b and c. And also here we can use more. And here you can see we have letters a b at the beginning and then a c at the beginning. Another way to write this would be to use these pipe symbols here as an alternative. And the result is the same here.

15 Working with Strings

Working with Strings

- Length of a string: `${#VARIABLE}`
- Concatenate two strings

```
VAR1="Hello"
VAR2="you"
VAR3=${VAR1}:${VAR2}
```

- Beware the difference

```
VAR3=$VAR1and$VAR2
VAR3=${VAR1}and${VAR2}
```

Next, we look at some special features of the Bash programming language in connection with character strings. If you need the length of a character string, write Dollar hash and then the variable name and at the end the closed . Normal programming languages either have functions for this or, in the case of object-orientated programming languages, corresponding methods on the appropriate classes. Certain syntactic tricks are used in Bash programming. Use the instruction in the second block to join several strings together and assign the result to a new variable. You write the individual elements one after the other on the right-hand side of the assignment for the third variable. Variable three contains the character string “hello” followed by a colon and then “you” in the result. Here, you can also see an example of why using curly brackets is sometimes necessary when accessing the value of a variable. In the last example, the first line would attempt to access a variable named VAR1and. However, what was meant was that in the second line, the content of variable one should be concatenated with the word “and” followed by the content of variable two.

16 String Variables and Pattern Matching

String Variables and Pattern Matching

- Replace first occurence: `${var/pattern/replace}`
- Replace all occurrences: `${var//pattern/replace}`

You have just learned the basics of regular expressions, and now we will show you how to easily find and replace patterns in a variable with a string. You start with the dollar followed by the opening , then the variable name followed by a /. Now, write a regular expression that you want to search for in the content of the variable. This is followed by another / and then a sequence of characters that must be substituted instead of the one found. Finally, of course, you write the closing curly bracket again. You can then assign the result of this expression to another variable. However, with this notation, only the first occurrence must be replaced. If you want to replace every occurrence of this pattern, write 2 / instead of one / in front of the pattern. Because this is a very important function within the programming of scripts, let's take another look at it live.

17 Screencast 5

Screencast 5

- todo

I would like to demonstrate some examples for variable substitution and I will do this on the command line here but it works in the same way also in bash script files but on the command line it's a little bit more interactive now. So we first create a variable and we assign the value hello world to this new variable var. We can output this variable and now we start with the substitution. We want to replace in this variable var the string hello by the German word hallo. And this is the result, this is clear. We can also assign the result to a new variable w and now output this variable and that's the expected result. When we have this word hello multiple times in this variable we can use this substitution to replace it only once. So you can see here the first hello was replaced but the second not and when we use two slashes in the beginning all occurrences are replaced.

18 Remove Substrings by Pattern

Remove Substrings by Pattern

- `VAR1="Helloworld"`
- Cut at the beginning: `${VAR1#Hello}` returns `world`
- Cut at the end: `${VAR1%world}` returns `Hello`
- If the substring is not part of the given string, the original value is returned

A special case of the last search and replace function is the search and cut of character sequences. So, if you want to delete a character sequence at the beginning or end of a string, you can also work directly with the symbol hash or the percentage sign. Please take a look at the example next to me. The variable is assigned the value Hello World at the beginning. In the next line, we use the hash followed by the word hello to cut out this word and only get the word World in the result. So, we use the hash to search at the beginning of the string and cut out the result. Similarly, we use the percentage sign to cut out something at the end - the next example returns hello because we have cut out the word world at the end. The variable's original value is returned if the pattern you are looking for is not contained in the string.

19 Remove Substrings by Index and Length

Remove Substrings by Index and Length

- `VAR1="Helloworld"`
- `echo ${VAR1:2}` returns `lloworld`
- `echo ${VAR1:0:1}` returns `H`
- `echo ${VAR1:1:3}` returns `ell`

It is also very easy to cut out parts of a string based on the index position and length. Again, we start with a variable to which we assign the character string Hello World at the beginning. In the next line, we specify that we want the complete string only from index position two, which is the third character. Note that we start counting at zero. In the next example, we do not only define a start position but also a length after the second colon. So, in this case, we start at the first letter because we specify index position zero, and we want to have exactly one character from there. Therefore, the result of this expression is H. In the third example, we start at index position one, i.e., at the second character, and then include three characters in the result.

20 Remove Substrings by Index, Length, and Arithmetics

Remove Substrings by Index, Length, and Arithmetics

- `VAR1="Helloworld"`
- `${VAR1:1:$((${#VAR1}-2))}`
- Result: `elloworl`

You can also add a bit of arithmetic to this function for cutting out partial strings to take the length of the string into account. Again, we start with a variable with the value hello world. In the second line, we now specify that we want to extract a string beginning at the second position, i.e., the "e," and extending to the penultimate letter in this string. To do this, we have to determine the length of the character string in variable one, which we do with the hash, then subtract two and enter this as the length specification. To perform arithmetic operations easily, you must put the expression in double round brackets and write a $ in front of it.

21 Exercise 2: Working with Strings

Exercise 2: Working with Strings

- Convert file names: `FILE.jpg` to `FILE.png`
- Convert file names: `FILE.tex` to `FILE_v2.tex`
- Convert file names: `FILE_vN.c` to `FILE_vM.c`
  (where `N` is a number and `M` = `N` + 1)

In today's second exercise, we would like you to look at some typical use cases for working with strings. The first exercise is about renaming a file name. Let's assume you have saved a file name in a variable. The file ends with .jpg. You probably know this because you have only searched for files with this extension, for example. In your script, you now want to convert the JPG file into a PNG file; for this, you need the appropriate file name. So instead of .jpg, now .png. How do you do this if you can only use the string operations you have just learned? In the second task, you must change the file name again. The file name should have the character string underscore v 2 before the dot. This should now be a version number. Again, please search for the appropriate expression that only requires string operations. And finally, please increment a file by one number with a given version number. A file with _ v 2 should, be renamed to _ v 3. Please try to write the shortest possible instructions for this. When you have finished or want to view the solution, please click the button.

22 Screencast 6

- todo

Let's create a new script for this exercise. The name shall be rename.sh. We create a first variable here, file, which contains a filename, image.jpg. We want to replace the suffix jpg by the new suffix png. A new variable here, file renamed. We use this variable substitution using the percentage sign to delete what comes at the end. We delete this .jpg at the end of the filename. Then we have to replace this by png. We simply add png at the end. We print this to the console and let's check this. We have to change the permission first. Then we execute the script and this is the correct filename. For the second exercise, we continue the same script here. We create or we define a new value to this variable file, text.tech. And within this exercise, we want to add a version number. We remove the suffix .tech and we add a version number and the suffix. We print the new value to the console and we check this. This is also correct. For the third exercise, this now becomes a little bit more difficult. We start with a file named main underscore v9 dot c. This filename has a version number already. The first thing we do is we extract the part before the dot. Let's call this the prefix. We use again the percentage sign to remove the suffix here. The next step is we must extract the number. So we create a new variable here and we take the value prefix. We want to remove everything except the number. When we don't write anything after the second slash, this means we remove things here. Let's test this first by using an echo here. We execute the script and now you can see we have the number nine. Next we have to increase. First we extract the part before the number and we create a new variable for this. We name this simply part now. We take the prefix and now we remove all digits. We can write here the range from zero to nine. We could now use the plus symbol here or we use two slashes to remove all occurrences of digits in the string. We replace this by nothing. The last step is to increase the version number. We have this variable number already and we do now this

arithmetic thing that we already discussed with the dollar sign in two parentheses here. Number plus one. Now we can create the target file name which starts with part. Then we have the increased number. And at the end we have the suffix dot c. So let's check this. And that's the correct result.

23 Array Variables 1

Array Variables 1

- `VAR[3]="hello"`
- No need to declare an array beforehand
- `echo ${VAR[3]}` returns `hello`
- Undefined positions return the empty string

In the last topic for today, we will briefly look at the concept of arrays. Arrays are always used when we want to manage several pieces of data under the same name but with different index positions. As the Bash programming language has no concept of data types, you can also store wildly mixed strings or numbers in an array. Declaring an array or specifying a maximum number of index positions is also unnecessary. You can start, as in the first line next to me, and write the character string hello in the fourth position, i.e., index three. You will recognize the usual notation with square brackets here, just as in C or Java. The access then takes place using the normal notation for accessing variables, and you specify the index position again in the square brackets. You will receive an empty string if you access positions that still need to be assigned.

24 Array Variables 2

Array Variables 2

- Define an array: `VAR1=("one" "two" "three")`
- `echo ${VAR[0]}` returns `one`
- Length of an array: `${#VAR1[*]}`
- Access all elements: `${VAR1[*]}`
- Copy an array: `VAR2=(${VAR1[*]})`

You can also define several index positions at once in the Bash programming language. To do this, write the values at the respective positions separated by and set in round brackets. In the third line next to me, you can see the notation for determining the length of an array. You use hash marks, as you do when determining the size of character strings, but you must also place the asterisk in square brackets after the variable name. If you want to access all elements, i.e., all index positions of an array, write an asterisk instead of a concrete number at the index position. You need this if you want to copy the content of an array, for example. You can see an example in the last line next to me. Make sure that the right-hand side of the expression still contains the parentheses. This is needed so that the result is an array, not just a character string.

25 Array slices

Array Slices

- Only return slices (parts) of an array
- You have to give a start index and a length
- `${VAR[@]:1}` returns all except the first
- `${VAR[@]:0:2}` returns the first two elements

And just as you can copy parts of a string, you can only copy sections of arrays. These sections are called slices, by the way. Again, a notation consists of an index position for the start of the slice and a length specification. If no length is specified, everything from the specified index position to the end of the array should be considered. You can see this in the third line next to me. Or, the normal case is that you also specify a length, as in the fourth line next to me. So, with this expression, you would get the first two elements of the array.

26 Exercise 3: Working with arrays

Exercise 3: Working with Arrays

- What happens if no index is given? `${VAR[@]::2}`
- Get the length of the string at index 1 of an array
- How to add an element to an array?
- Search and replace of array elements?

In the last exercise for today, we want you to do a bit of puzzling. Of course, you can also search the web for information if you get stuck. Firstly, please try out what happens if you omit the index specification when slicing. What kind of array will you get back? Perhaps think about what might make sense here before you try it out. And then test your guess. In the second exercise, let's assume you have an array with character strings. Which instruction would you use to find out the length of the string at position 1 in the array? In the third exercise, you are asked to add an element to an array. Should the new element be appended to the end of the array? And finally, in the fourth exercise, please find out how to search for and replace values in an array. In principle, you already know this syntax from a previous slide, but we have yet to apply it to arrays. When you have finished and want to see the solution, please click the button.

27 Screencast 7

Screencast 7

- clear
- `echo ${#Unix[3]}`
- `Unix=("${Unix[@]}" "AIX" "HP-UX")`
- `echo ${Unix[@]/Ubuntu/SCO Unix}`

We create a new script for this exercise named arrays.sh. We start with a shibang line as usual and first we create an array variable here. Let's write var equals and then we use the parentheses and we just write a list of words here. It's mixed German and English. We print the contents of this array but we use slicing for this. We write two colons and then number two. We only want to see the first two elements. We change the permission of this script here. We execute the script and then you can see only the two words are shown here. We use this colon syntax and we make use of the default values here. The next point was to return the length of the word at index position one. We use the hash symbol to get the length and we use the array syntax to access the first element at index one. That's four because the word at index position one is the German word Welt which has four characters. Next, before we continue, I want to show you a new syntax here. This is how to add new elements to an array. The new syntax is plus equal. It's an usual assignment operator that you know from other languages already. We add the new value Samstag, the German word for Saturday here. Beware of the parentheses that are necessary here. Let's double check. Now this new word is the last word in the array. Next, we want to search and replace. We access all elements of the array. Then we use the syntax with a slash to replace all a's by u's. We double check and here is the result.

28 Summary

Summary

- You learned to write and apply regular expressions
- You learned to use pattern matching for strings
- You learned to use arrays and slices in shell scripts

So, that's it again for today. In this second part of the introduction to programming Bash scripts, we mainly deal with texts and character strings. In the beginning, you learned what regular expressions are. Such regular expressions are used to find certain patterns in text files or character strings stored in variables. Then you saw that replacing character strings in variables is very easy, especially when replacing happens at the beginning or the end of a character string. This function is often needed when programming Bash scripts; you must know how to use these special functions. Finally, we briefly looked at the idea of arrays. This is very similar to the usual programming languages. Extracting sections from arrays is also available in all programming languages but is realized using functions. Syntactic tricks are used in Bash programming.

29 Literature

Literature

- Patrick Ditchen: Bash: Einstieg in die Shell-Programmierung. mitp, 2018.
- Jürgen Wolf: Shell-Programmierung. 5. Auflage, Rheinwerk, 2016.
- Sander van Vugt: Beginning the Linux Command Line, Apress, 2015.