Bash: Variables and Arithmetics

Prof. Dr. Peter Braun

15. Oktober 2024

1 Titlepage

Bash: Variables and Arithmetics

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 variables in shell scripts
- You learn how to do arithmetic operations
- You learn how to use `for` and `while` loops
- You learn how to use `if` and `switch` statements
- You learn how to use functions to structure code

Welcome to this last unit of the introduction to programming shell scripts with the Bash programming language. In this unit, we will look into the concept of variables in more detail. We must also discuss how to describe basic arithmetic operations in this programming language. You will find that the possibilities, or rather the type of syntax, take some getting used to. There is also a major limitation. This is because we cannot work directly with real numbers using the normal means of Bash programming, but only with integers. We will then introduce you to the usual control structures, such as branches and loops. And at the end, we will briefly show you how to divide a script into individual functions, for example, to make the source code easier to read and reuse sections of code.

3 Variables

Variables

- Define a variable: `VARIABLE=value` (no spaces)
- Access the value of a variable: `$VARIABLE`
- Sometimes necessary: `${VARIABLE}`
- Bash does not have a concept of data types

We had this already in the last units, but once again shortly. The Bash programming language has a concept of variables but no concept of data types. Unlike in other programming languages, you do not have to declare variables before you can use them. You write an assignment operation anywhere in your script, with the variable's name on the left-hand side and the value after the =. You should already know this from the last unit.

4 Constants

Constants

- Define a constant: `readonly CONST=value`
- If you want to change `CONST`, you will get an error

It is also possible to define constants, i.e., variables that can no longer be changed. Similar to other programming languages, constants are identified by an additional keyword. In the Bash programming language, this is the keyword "read-only," which you write before the assignment operation. If this constant is overwritten later, you will receive an error message during execution.

5 Scope

Scope

- Variables only exist in the shell they are defined in
- Variables from the shell are not visible in scripts
- Variables from scripts are not visible in sub-scripts
- To make a variable visible: `export VAR`

If you define variables or constants within a script, these variables or constants are only visible and, therefore, usable within this script. Once the script has been executed, these variables no longer exist. If you have already familiarised yourself with programming on the Linux command line, you may have already seen that variables can also be defined there. However, these variables on the command line are not readily visible within a script. This is also true in the other direction. If you define a variable within your script and call another script from your script, the variables are not visible there either. However, there is a way to bypass this restriction. The keyword “export” exists for this purpose. You can place this keyword directly before assigning a variable or after the variable has already been defined. Use the “export” keyword with the variable’s name as an argument. You can see an example of this in the last line next to me. Such an instruction then makes the variable visible in all called shell scripts. If you define a variable on the command line and make it visible using the Export keyword, you can also access these variables in your scripts.

6 Variable Substitution: Default Values

Variable Substitution: Default Values

- `${VAR:-value}`
- If `VAR` is set and **not** `null`, use it
- If `VAR` is set **but** null, use `value`
- If `VAR` is not set, use `value`

- `${VAR-value}`
- If `VAR` is set and **not** `null`, use it
- If `VAR` is set **but** null, use `null`
- If `VAR` is not set, use `value`

Now, we come to a somewhat complex topic to be found in the literature under the keyword variable substitution. It is a special syntax for handling or accessing the value of variables, which is very cryptic at first glance but ensures that you can save yourself a lot of "if" statements. Firstly, let's look at the use case of so-called default values. Let's assume that there is a variable with the name VAR in your script, and the value comes from an argument that the user has specified when calling the script. You now have to continue working with this variable within the script. However, you must determine whether the user specified the argument when calling the script. Normally, you would work with an IF statement in which you check whether the variable has already been filled with a value. If yes, continue working with this value. If not, continue working with a default value. In the Bash programming language, you can express this very compactly with just a few characters and without an IF statement. Please look at the upper block next to me and the first line. You will recognize the usual notation for accessing the value of a variable with dollars and the curly brackets in front of the variable name and at the end of the line. There is now a colon and a minus sign after the variable name. A value follows this. This value is usually a literal, for example, a number or a character string. However, it is also possible to access another variable. We describe the meaning of this line with the three bullet points below. If the variable exists and is not null, we continue working with the variable in the result. The expression in the first line will then be the variable's value. A variable is only null if it contains the empty string. We use the default value if the variable exists but contains no valid value. The third case is simple. If the variable does not exist, we continue working with the default value. The second block below differs from the first case by only one character in the expression. We omit the colon after the variable name and only write the minus sign. This slightly changes the

meaning of this expression. The first and third cases are identical. In the second case, however, we do not continue to work with the default value; we work with the empty string.

7 Variable Substitution: Is Variable Set?

Variable Substitution: Is Variable Set?

- `${VAR:+value}`
- If `VAR` is set and **not** `null`, use `value`
- If `VAR` is set **but** null, use `null`
- If `VAR` is not set, use `null`

- `${VAR+value}`
- If `VAR` is set and **not** `null`, use `value`
- If `VAR` is set **but** null, use `value`
- If `VAR` is not set, use `null`

The second use case concerns the opposite scenario. If a variable exists and is already filled with a valid value, then you do not continue working with this value but with the specified default value. In this case, the variable can be understood as a switch used to decide whether the expression accepts the specified alternative value or is set to the empty string. Let's first look at the case above and the first line. The variable name is now followed by the colon and a + sign. Let's go through the three possible cases again. If the variable exists and has a valid value, we continue working with the specified alternative value. In other cases, the expression in the first line results in the value zero, i.e., the empty string. Either the variable exists but does not have a valid value, or the variable does not even exist. The second block next to me is only slightly different from the first. In terms of syntax, the only difference is that the colon is omitted. In terms of meaning, there is only a difference in the second case. If the variable exists but does not have a valid value, the expression returns the alternative value here.

8 Variable Substitution: Redefine Variables If Not Set

Variable Substitution: Redefine Variables If Not Set

- `${VAR:=value}`
- If `VAR` is set and **not** `null`, use `VAR`
- If `VAR` is set **but** null, assign and use `value`
- If `VAR` is not set, assign and use `value`

- `${VAR=value}`
- If `VAR` is set and **not** `null`, use `VAR`
- If `VAR` is set **but** null, use `null`
- If `VAR` is not set, assign and use `value`

The third use case is now very similar to the first. It is a matter of checking whether a variable exists and has a valid value. If not, work should continue with a default value. However, the difference is that the specified variable is now overwritten with the default value. Firstly, please look at the upper block again and its first line. The variable name is now followed by a colon and an equal sign. This already looks like an assignment operation in some older programming languages like Pascal. If the variable exists and has a valid value, we continue to work with this value, and the variable's value does not change. In other cases, we continue to work with the specified default value, and the variable also has this default value as its new content. The second block below next to me differs in syntax again only by the missing colon, and the meaning varies only in the second case. If the variable exists but is perhaps in an empty string, then we continue to work with this empty string. As I said, the use case is identical to the first one. The only difference is that the variable may be overwritten with the specified default value here.

9 Variable Substitution: Error Messages

Variable Substitution: Error Messages

- `${VAR:?value}`
- If `VAR` is set and **not** `null`, use `VAR`
- Otherwise, exit with error code `1`

The last use case is now very simple. Here, we want to check whether a given variable has a valid value. If so, we continue working with this value. And in all other cases, the execution of the script is terminated with an EXIT code 1. This notation is always used if the value of a variable is necessary for further processing in a script; otherwise i.e., if no value is available, the script must be canceled immediately. This notation is used, for example, when checking the necessary arguments that the user must have provided when calling a script. If no arguments are provided, the script can also be canceled immediately.

10 Exercise 1: Testing Script Arguments

Exercise 1: Testing Script Arguments

Set variable `NAME` to value of `$1` so that:

- If `$1` is not valid, use a default value
- If `$1` is not valid, exit the script

Set variable `NAME` so that:

- If `$1` is valid, set `NAME` to `"file.txt"`

In the first exercise for today, please experiment with these relatively complicated substitution methods. Please write three scripts for the three tasks you see next to me. In all cases, the script should accept exactly one argument. In the first case, a variable with the name NAME is to be defined to contain the value of this first argument. However, if the first argument is not set at all or equals the empty string, the variable Name should be set to the given default value. Please think about the default value yourself. The second script should check the first argument with such a substitution rule and terminate the script if the argument is not valid. Finally, the third script should set the variable NAME to the value FILE.TXT if the first argument is valid. // When you have finished the task or want to see the solutions, please click the button.

11 Screencast 1

Screencast 1

- todo

So we solve this exercise in a new script where we implement all three commands step by step. The name is var substitution shell script. We start with the shebang line and the first command was to check if the first argument And here we can just use the number one to refer to the first argument. We check this and if it's not valid we have to use a default value. So we use the minus syntax here and we give the default value here as a string we output the contents of variable name and then we check the script first we have to set the permissions and then we call the script first without any arguments then we get the default value then with an argument and then this is the correct value. We continue with the next statement we comment out the first two lines here and then we can continue with the second case where we want to check if the first argument if it's not valid then we want to exit the script with an error message. So again we check here the first argument and now we use the question mark here as syntax and as value we define an error message and we output the contents of variable name so if it's valid then it's printed otherwise the error message is printed and first we call this with hello this is the correct output and then without any argument we get the error code and we can check the exit status here by using this special variable with question mark and this is one so the exit code was one. We continue with the next case and the last case and in the last case we want to use the argument one so the first argument like a switch so if it's valid then we define a variable to a specific value we assign a new value to a variable if the argument one is set so syntax for this is plus and the value that name should have then is file.txt. So now the value of the first argument is not important it's only important that it is a valid value. We check the script again so first with an argument then we have file.txt without an argument we have nothing.

12 Arithmetic

Arithmetic

- Command `expr` evaluates mathematic expressions
- Example: `VAR=$(expr 2 + 3)` (spaces necessary)
- Shorter way to write this: `$((2+3))`
- Example: `VAR=$((2+3))` (no spaces)
- Allowed operators: `+`,`-`, `*`, `/`, `>>`, `<<`, `&`, `|`
- You can't use float values
- Special command for this: `bc`

Let us now turn to the possibilities for mathematical operations in the Bash programming language. Under Linux there is the Expression command, abbreviated to E X P R, with which you can perform mathematical operations on the command line. The numbers and mathematical operators such as plus and minus are passed as arguments to this Linux command. It is important that you pay attention to the spaces to separate the arguments from each other. Within a script, you can call this command as part of a substitution by enclosing the statement in brackets and placing a $in front of it. Both on the command line and within a script, yo$ before it. It is now important to explicitly not write any spaces in the mathematical expression. You can see an example next to me in the center. The usual basic arithmetic operations, including integer division and the bit operators for left and right shifting and "logical and" as well as "logical or" are permitted as operators here. In the Bash programming language, a variable can only contain character strings or numbers but only the integers. In Bash, you cannot work with real numbers, i.e., floating point numbers. However, another command in Linux with the name BC makes this possible. BC is a separate programming language for mathematical expressions with arbitrary precision. In the second exercise, we will now take a closer look at the topic of mathematical operations.

13 Exercise 2: Script Arguments and Arithmetic

Exercise 2: Script Arguments and Arithmetic

- Create a new shell script with the name `add.sh`
- It should take two arguments
- Print the sum of the two arguments

The second task is simple. Please write a script that accepts two arguments. Both arguments should be numbers. The script should calculate the sum of the two numbers using the methods you have just learned about arithmetic operations and display the result on the screen. As I said, this exercise is very simple. You could also use the time to experiment with the other mathematical operators we discussed on the last slide. Please click the button when you have finished and want to see the solution.

14 Screencast 2

- todo

So I think this exercise was really easy. We just write the solution briefly. The script is named at dot sh. We start with a shi-ben line and then we can immediately start with the assignment statement. The variable is named sum and we use the arithmetic operation here of argument one and two and we output the sum. We change the permission here and then we start the script with two numbers and that's the correct result.

15 Command texttttest

Command `test`

- `test` is used when comparing string or numbers
- `test` exits with code `0` only if successful
- Long form: `test $VAR1 OP $VAR2`
- Short form: `[ $VAR1 OP $VAR2 ]`
- Operations for strings: `=`, `!=`, `<`, `>`
- Operations for numbers: `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`
- Several operations for files

Now, we come to the control structures. The programming language Bash offers control structures similar to C or Java. However, the syntax is slightly different. For both branching and loops, we first need a way to describe expressions that can be true or false. Under Linux, there is the "Test" command for this. Here, you specify the variables or values and the operator as arguments. In the case of comparison, the operator is, of course, always in the middle. However, there /are/ cases where the operator is at the beginning, and only one variable is specified. The Test command evaluates the expression, and the EXIT code of this command is zero if the comparison result is TRUE. Otherwise, the EXIT code is greater than zero. Within a script, you can either call this Test command directly as usual by substituting commands. Or you can use a shorter notation with square brackets, as in the example next to me. As operands, the Test command now allows either numbers or character strings or file names. It looks strange, but the operators must be picked according to the expected data types. For example, you can only use the equal sign for character strings. If you want to compare two numbers with each other, you must write minus EQ as the operator. You can also use the less than or greater than sign for character strings. However, here, you compare character strings according to their alphabetical order. The other mathematical comparison operators are derived from the usual abbreviations. NE stands for Not Equal, LT for lower than, and GT for greater than. There are also numerous operators that can be used in connection with files. But we'd better have a look at these live on the computer.

16 Screencast 3 Command texttttest

Screencast 3 Command `test`

- Please test the command in your terminal. Which exit code do you get? (`echo $?`)
- Comparing string:

```
["hello" = "world"]
["hello" != "world"]
[-z "$var1"] #var1 is empty
[-n "$var1"] #var1 is not empty
```

I would like to demonstrate using this test command and to make this very easy we will now use this test command not using this brackets sorry not using the test command itself but using this brackets syntax and we will do this on the command line immediately and not write a new script so we start with the simple comparison of two strings and we can check the exit status here the zero that means it's true so both strings are identical and the other case is then exit code one we can now extend this and use the syntax to define that this true string should be printed if the result of the comparison is correct and false otherwise and here you can see this works next we can compare numbers and first we start with very easy case here five equals five and then we change the numbers and then the result is false next I would like to show you that there is no variable named demo defined in this shell so let's assign a value to demo hello world and then we can check if the variable is set by using this dash n operator and you can see the value is set or the variable has a valid value and the last thing I would like to show you is about files so we create a file first because later we want to check if this file exists test 99.txt with a command touch we create an empty file and then we use the dash e syntax to check if this file which we pass here as a name file name if it does exist yes it exists and then we can change the file name to 100 and then you can see it does not exist so you have these kind of operators here when you have strings you can use just the double equal or not equal operator with numbers you have to use the special syntax this equal operator you can check if variable exists or does not exist and we can check if files exist or do not exist so that's all for this

17 Control Structures: Conditional

Control Structures: Conditional

```
if [ $# -lt 2 ]
then
  echo "Not enough arguments"
else
  echo "Everything is fine"
fi
```

Firstly, let's take a look at the IF statement. The necessary keywords will certainly be familiar to you from other programming languages. You start with the keyword IF followed by the expression that results in either true or false. The notation with the square brackets corresponds to what we have just learned with the Test command. On the next line, you write the keyword THEN, followed by the instructions to be executed if the condition is true. After the Else, the instructions will be executed if the condition is false. At the end, the statement is limited by the keyword F I. This is the character string IF in reverse order. It would be best to get used to this way of writing with the keyword THEN in a new line. It is also possible to write the keyword THEN directly after the condition, but then you must place a semicolon after the closing square bracket. On the Internet and in some books, you will also find the notation with double square brackets. This is also correct in most shells and offers minor advantages in some situations. However, as it only works on some UNIX variants, we will not discuss these special features in detail here. As in all other programming languages, further conditions can be included in this construct. The keyword for this is ELIF, followed by an additional condition. // In the example next to me, we compare the number of arguments with 2. The error message is displayed on the screen if fewer than two arguments have been passed. Otherwise, the message that everything is OK.

18 Control Structures: Switch-Case

Control Structures: Switch-Case

```
var=$(date +%a)
case "$var" in
  Mon) echo "Today is Monday" ;;
  Tue) echo "Today is Tuesday" ;;
  Wed) echo "Today is Wednesday" ;;
  Thu) echo "Today is Thursday" ;;
  Fri) echo "Today is Friday" ;;
  *) echo "Today is weekend"
esac
```

The so-called switch case statement offers a compact notation for many alternatives. The example next to me branches into different control paths depending on the current day of the week, which we obtain via the Linux command Date, and outputs corresponding messages for today's day. The use of the variable is enclosed in double quotes after the keyword Case. You see this very often in scripts if you want to make sure that character strings with spaces are also processed correctly. However, the individual alternatives are without double quotes and are delimited by the round closing bracket. This is followed by one or several further instructions to be executed in this case. Two semi-colons are used as delimiters for the next case. You will see the asterisk on the penultimate line followed by the closing parenthesis. This is the notation for all other cases. At the end, the keyword follows again in reverse letter order.

19 Switch Case Using Regular Expressions

Switch Case Using Regular Expressions

```
case "$1" in
  -[hH]|-help) echo "Help" ;;
  -[tT]|-test) echo "Test" ;;
  *) echo "Wrong option $1"
esac
```

Another example of such a switch case statement is shown on this slide. This example comes from the processing of arguments for a shell script. We want to show how compact the notation is in dealing with different alternatives—the first argument of the shell script is used here as a variable to decide on the branch. The notation of the alternatives with a vertical bar in the next line resembles regular expressions. Before the vertical bar, there are options for arguments with only one letter. After the vertical bar, the argument is interpreted as a complete word. However, a dash symbol is expected as the first character in both cases. In the left-hand case, the two permitted spellings for the letter H are shown in square brackets, one as a lowercase letter and one as an uppercase letter. All in all, this one line means that the word help should always be printed if the first argument of the shell script contains either dash small h or dash capital H or dash help.

20 Control Structures: For Loop

Control Structures: For Loop

```
for var in 1 2 3
do
  echo "Element " $var
done


for var in ${ARRAY[*]}
do
  echo "Element " $var
done
```

Now, we come to the first type of loop. The so-called for-loop is exactly what you would expect from other programming languages. It runs through a list or an array, element by element. After the keyword for, the name of the so-called loop variable is specified, which takes the current element's value from the list or array in each loop iteration. Between the keywords DO and DONE are the instructions that are to be executed for each element. You can then access the value of the loop variable here. Please take a closer look at the first line in the example above. The keyword "in" is followed by the sequence of values to be run through one after the other. The space character serves as a delimiter between the values. The example below is a little more realistic. Here, all elements of an array are run through. The somewhat strange notation with the square brackets and the asterisk corresponds exactly to the syntax required to access all elements of an array.

21 Control Structures: For Loops Over Number Ranges

Control Structures: For Loops Over Number Ranges

```
for ((i=0; i<=9; i++))
do
  echo "Hello World " $i
done
```

```
for i in 0..9
do
  echo "Hello World " $i
done
```

You can use the following notation to run through a specific number range. In this example, the loop variable is the I. We start with 0 and run through all numbers in sequence, including nine. The upper notation corresponds to the C programming language or Java syntax. The lower notation uses the two dots to define a number range. Please note the different types of brackets that are necessary in these cases.

22 Control Structures: For Loops Over File Sets

Control Structures: For Loops Over File Sets

```
for file in *.txt
do
  echo "Filename:  " $file
done


for name in $(find .  -name *.jpg)
do
  echo "JPG Bilddatei:  " $name
done
```

Now, two more examples are often found in scripts. In a loop, we want to process all files that match a certain pattern in sequence. In the top example next to me, you can see the simple notation in which you search for files with the suffix TXT in the first line of the loop using a pattern. In the second example below, you can see the solution for a very similar use case. Here, command substitution is used, and the result of a call to the Linux command FIND is used. The advantage is that you can control the search for files more finely with the FIND command than with the simple means in the example above.

23 Control Structures: While Loops

Control Structures: While Loops

```
i=0
while [ $i -lt 10 ]
do
  echo $i
  i=$((i+1))
done
```

As in other programming languages, there is also the WHILE loop in the Bash language. Next to me, you can see a simple example demonstrating the basic structure. A loop is always used if you can define an expression to keep the loop running. In contrast, a FOR loop should always be used when running through a list or an array. You formulate the condition for a while loop as an expression similar to an IF statement. The loop body is embedded in the two keywords DO and DONE. At the end of the loop body, note the necessary instruction with the arithmetic operation to increase the run variable by one.

24 Reading Files

Reading Files

- Process a file line by line
- The content is piped as input to `read`

```
while read line
do
  echo $line
done < $1
```

In practice, such a WHILE loop is often used when processing all lines of a text document in sequence. Here, we combine the Linux command READ, which you already know and which is normally used to process input from the user, with a redirect at the end of the WHILE loop. In argument one, we expect the name of a file. The content of this file is passed into the loop using the less than sign in the last line. The READ command at the beginning of the loop reads a line and saves it in the variable line. We can then access this variable within the loop.

25 Exercise 3: Reading Files

Exercise 3: Reading Files

- Write a script that accepts two arguments
- Process file that is given as the first argument
- Only print a line if the length is smaller than the second argument

Now, we come to the last exercise in today's unit. Please write a script that accepts two arguments. The first argument should be the name of a text file. The second argument should be a number. Read the text file line by line and check the line length. If the length is less than or equal to the specified number, i.e., the second argument of the script, then output this line on the screen. All longer lines are not displayed on the screen. When you have finished or want to view the solution, please click the button.

26 Screencast 4

Screencast 4

- todo

We start a new script for this last exercise for this unit. We expect that this script has two arguments. The first one is a file name and the second argument is a number. First we check the second argument. This number should express like a number of characters for the line length here. We want to have a default value. If the second argument is not defined, we use 20 as the default value. Then we run through all lines of the text file. The name is given as the first argument. We use a while loop here and we read line by line. In the loop we check the length of this variable line. We compare this lower than with the given length here. If the length is smaller than this value of length, then we echo the line. In this while loop we pipe in the text file. We change the permission as usual. We check this script with a text file that is prepared here, text 4. We start the script. First we only pass the name of the text file. We do not give a second argument here, so the default value should be used. All lines have less than 20 characters, so all lines are printed. Then with length 10 only the last line is printed, because this has 9 characters.

27 Functions

Functions

- Functions are reusable blocks with parameters
- Parameters are similar to script arguments: `$1`, `$2` ...
- Return value is set with command `return`
- Access return value by `$?`

```
foo() {
  echo "Hello World" $1
  return 10
}

foo test
echo $?
```

The last thing we want to do today is briefly touch on functions. Sections are reusable blocks of code that can be called with arguments and return a result. To define a function, write the name of the function followed by brackets. This is followed by the instructions of the function embedded in curly braces. The arguments with which the function is later called are accessible within the function via the special variable dollars followed by a number. The return value of a function is introduced with the keyword Return. The function is called by the given name, followed by the arguments. However, please note the difference to other programming languages. This is because we do not write the arguments in brackets. Using an assignment operator, we cannot simply pass the return value function to a variable. Instead, we access the return value as if we wanted to query the EXIT code of a Linux command. The special variable for this is called dollar and question mark. Please note that functions must first be defined in a script before use. Therefore, the first call of a function must always occur later in the source code than the definition of the function.

28 Summary

Summary

- You learned how to use variables in shell scripts
- You learned how to substitute variables
- You learned how to do arithmetic operations
- You learned how to use `for` and `while` loops
- You learned how to use `if` and `switch` statements

This was the last unit in our short introduction to programming using the Bash programming language. We looked at the concept of variables and constants. We then showed you how to solve certain cases when dealing with variables quickly and compactly. This involved, for example, the use of default values. You can use this notation to avoid branching very often or almost always. You have briefly seen the somewhat limited possibilities for arithmetic operations. We then looked at the typical control structures. You have learned about simple branching. Then, the complex switch case statement. Like all programming languages, the simple FOR-loop and the WHILE loop are available. In the end, we showed you how to structure source code with the help of functions. Literature