Tekstmanipulatie, week 14

1. `expr` and `bc`

In order to make easy calculations you can use the 'bc' command (bell's calculator). Type bc <RETURN> and you can immediately type in any expressions, like 3+4 or (45/3400)*100. In fact, similarly to the way we were writing short files by using 'cat', we are just using the fact that this command needs an input file, and if nothing else is specified, then it is the standard input. Therefore the program can be ended by ^d (CTRL + D: end-of-file). Or, alternatively, by ^c (CTRL + C: stop the running program).

Therefore why not doing things like:

```
echo 3+4 | bc
echo 23/46 | bc
```

Hey! Why is 23 / 46 = 0 ?! Because, if otherwise not specified, bc works with integers. Type ' scale = 4 ' to be able to receive your results with four decimals.
How to do this within one command line? You need an input file of two lines:

```
(echo scale = 4; echo 5/8) | bc
```

What does

```
echo 13 % 3 | bc
```

mean? The remainder of the division. And what is the problem with this one:

```
echo (13/26)*4 | bc
```

Try rather the following, and remember what you know about the escape characters:

```
echo \(13/26\)*4 | bc
```

What is the difference between `echo` and `cat`?

- `echo` sends to the standard output (or redirected standard output) its arguments, seperated by one space
- `cat` sends to the standard output (or redirected standard output) the content of the file(s) given as its argument(s), or (if no arguments are given) the standard input (or the redirected standard input).

You can find the same dichotomy among the commands dealing with mathematical expressions:

- `expr` outputs the value of the expression given as its arguments.
- `bc` outputs the value of the expression given in a file (mentioned as its argument) or given in its (maybe redirected) standard input.

Examples for `expr`:

```
expr 3 + 4
7

expr 3+4
3+4

expr \( 3 + 4 \) \/ 4
1

expr 2 * 3
expr: syntax error

expr '-2' \* 3
-6

expr 13 \% 3
1

expr 8 = 8
1

expr 15 = 2
0

expr \( 8 = 8 \) \& \( 3 = 3 \)
1

expr '(' 8 = 8 ')' '|' '(' 3 = 4 + 5 ')'
1
```

Remarks: The numbers, parantheses and arithmetic symbols are different arguments, therefore you should separate them by a space (if you don't: see the second example). Some out of the arithmetic symbols are metacharacters, therefore they should be protected using quotes or the escape character ('\') (what is the reason of the error message in the fourth example?). Division is understood as division of integers, and % refers to the modulo of the division. The last four examples show how logical statements are evaluated: 0 stands for the logical value FALSE, while 1 stands for the logical value TRUE. The '&' symbol means AND, '|' means OR. Check `man expr` for further possibilities (e.g. what happens if you use

these logical operations between numerals, and not between statements?).

The `expr` command, combined with back quotes (that is replaced by the shell with the output of the command line within the quotes) makes us an easier way to calculate type-token ratio or word-frequencies. How to calculate for instance the frequency of the word "the" in a given a a given file?

- Number of the occurences of "the" is given as the output of the following command line: `tr ' ' '\012' < file | tr -d ".,;:" | grep '^the$' | wc -w`

  Remark: if you put just '`grep the`', then you would match words like "therefore", too. The second `tr` will delete characters that might follow our word and are not separeted by a space: without this our command line wouldn't recognize them as tokens of the word that we are looking for, and `grep` would filter them out.)
- The number of words occuring in the text is given by: `wc- w < file`.

  Remark: if you wrote just `wc- w file` then the filename is also mentioned in the output, and this would lead to syntax error in the last step. (Try it out! It took me pretty long to find out what the problem was,,,)
- Since dividing is understood by `expr` as dividing within integers, therefore let's multiply by 10,000, so that we receive the results in 0.01%.
- As we need the input file twice, we need to write it to a temporary file. So the command line will be:

```
cat > file; expr ` tr ' ' '\012' < file | tr -d ".,;:"
| grep '^the$' | wc -w ` \* 10000 \/ ` wc- w < file `
```

2. Variables

Unix can and does handle a high number of variables. You can get the list of these with the command called '`set '`. In fact a useful way of using it is by pipelining it with `grep`, like:

```
set | grep a=
set | grep PATH=
```


The system itself has a high number of variables. They have always upper case names. Here are some of them:

SHELL : gives the path of the running shell
PATH : a set of paths that are checked (in this order) when you give a command (i.e. the name of a program), and the shell looks for it in the file system
HOME : the path of the home directory of the actual user (you)
MAIL: the path where your mails are located
PWD : the actual working directory
OLDPWD : the previous working directory (before the last cd command)
LOGNAME : your login name
HISFILE : the file where your 'history' is (the list of your previous commands, max. HISTSIZE / HISTFILESIZE number of them, and you can read them with the '`history`' command)
PS1, PS2: the settings of your primary and secondary promp
TERM: the type of your terminal

You can check their settings on your account.

The way you can give them a new value is the following:

```
PWD=Federalist
```

N.B.: no space before and after the = symbol. (Try out what happens if you put one.)

Changing the PWD variable results in changing your prompt, but in fact does not change your directory. Change the other system variables only if you are sure of yourself, or there is a system administrator standing just behind you... (Not in practicum time, please...)

You can define new variables yourself, just by giving them values. It is important to remember that all variables in UNIX are strings. (Remember: methacharacters, quotes, escapes,...)

Refering to a variable (let it be a system variable or a variable you have just defined) is done by putting the $ symbol before the name of the variable: in this case the shell replaces the string `$<var_name>` by the value of the variable, in the shell's pre-processing phase. This happens within the double quotation marks (".."), but not within the simple quotation marks ('...').

Examples:

```
birot@hagen:~> pear=apple
```

```
birot@hagen:~> set | grep pear=
pear=apple
birot@hagen:~> echo $pear
apple
birot@hagen:~> echo "$pear"tree
appletree
birot@hagen:~> echo '$pear'tree
$peartree
birot@hagen:~> echo $TERM
xterm
birot@hagen:~> echo '$TERM'
$TERM
```

Now it is logical that if you want to give the value of one variable to another variable, the way to do it is:

```
birot@hagen:~> banana=$pear
birot@hagen:~> echo $banana
apple
```

Remark: If you want to use a variable in one shell that you have defined in another one (like in a running program), then you have to *export* it. Consult any Unix book or '`man export`' on how to do that.

3. Type-token ratio

In a text, you will find several words, some of them occur more than once. For instance, if a text was composed of the previous sentence and this one, then the word "of" would appear three times.

If I ask "how many words are there in this text?", you can give two different answers. If each case when the word "of" occurs is a different word, then you speak about the number of **tokens**. Each occurence of the same word counts as a different token. But you can also ask what is the number of **types**, that is, how many *different* words you have in your text. If a word occurs more times, then these are different tokens of the same type.

Imagine that you have a text, in which word *A* occurs 5 times, word *B* occurs 3 times,

word *C* occurs once, and word *D* occurs only ones. Then you have 10 tokens (5+3+1+1=10), and 4 types (*A*, *B*, *C* and *D*).

If you are given a text, then you can calculate different statistics. You can calculate the number of tokens, which is the length of the text. You can calculate the number of types, which gives you how rich the vocabulary of the text actually is. Another useful statistics is the **type-token ratio**: the ratio of the number of types and the number of tokens (you divide the number of types with the number of tokens). In the above example, it is 4 / 10 = 0.4.

Type-token ratio is used for very different purposes. It can be used to measure somehow the richness of the vocabulary, for instance in child speech development. It has been claimed that the type-token ratio is typical to authors, different authors have different type-token ratios, so some researchers have tried to determine the authors of writings with debated authorship, based on type-token ratios.

Here is are the results of a very primitive way to calculate type-token ratios for the Federalist papers:

Some papers by Alexander Hamilton:

fed11.txt: 0.335
fed12.txt: 0.368
fed13.txt: 0.404
fed15.txt: 0.345
fed17.txt: 0.393
fed21.txt: 0.358
fed29.txt: 0.344

Some papers by James Madison:

mad37.txt: 0.336
mad38.txt: 0.310
mad39.txt: 0.250
mad40.txt: 0.277

Some papers by John Jay:

jay2.txt: 0.377
jay3.txt: 0.349

jay4.txt: 0.358
jay5.txt: 0.392

The type-token ratios of James Madison are much lower than the type-token ratios of the two other others. Unlike Hamilton, John Jay never has a type-token ratio above 0.400.

4. Shell scripts

After having solved a number of assignments, you might want to save some of them so that you won't need the reinvent them each time you need them. You can save them in a file, and just check that file each time before retyping the long chain of commands. But why not let the computer itself read this file and execute it? To make the long story short, can we write programs using UNIX?

There are two arguments pointing toward this possibility:

- Most of the Unix commands are in fact programs. Why couldn't we add new programs to them?
- The special program executing other ones is the Shell. The input of the Shell is also a file: if not specified otherwise, the standard input, i.e. what we type on the keyboard. (That was the reason why ^d, meaning end-of-file, results in logging out, i.e. quiting the Shell.) Why could we not run the Shell with files other than the standard input? The expression "shell scripts" comes from this idea.

Is Unix a programming language? It has been designed as an operating system, but it has so many possibilities that you can even write simple programs using it. What is a program?

- It is a file containing a sequence of commands, telling the machine what to do, and therefore it can be run several times.
- You may not want it to run each time exactly in the same way, but you want it to make the run dependent upon some circumstances.
- You may therefore want to give your program some parameters.

- So you need to be able to handle variables, also in order to store intermediate results.
- Using them you would like to write conditional commands (if...then...), as well as cycles.

All of these are possible within UNIX. We shall come back to some of these later.

At the moment what we want is to put a sequence of commands into a file, and then just run it.

How to have a sequence of (complex) commands? If you want to simply combine a sequence of commands, pipes, etc., just write them into new lines, or separate them with a semi-collumn (;).

For instance:

```
cat > a_simple_shell_script
echo Now I will list the subdirectories of the
directories whose name contains exactly 4 characters.
ls -l ???? | grep ^d
echo Thank you for your waiting.
echo What about an alphabetical order of these?
ls -l ???? | grep ^d | sort
echo Here you have it.
^d
```

Now, we have a file named `a_simple_shell_script` that contains six lines. What can we do with this? We want to run it. Let's type the file name after the promt, type enter, and... we get an error message:

```
bash: a_simple_shell_script: command not found
```

What is wrong? Let's type '`./a_simple_shell_script`', in some systems this is the way you can run the programs that are within your own directory. Did it help? No, you get the same error message. Because the machine doesn't know that this file has been written to make it run (and not only a text-file, that can be, e.g. sent to Mariette as the solution of your assignment). What to do? There are two steps:

- First you have to tell the machine how to understand the code, since it is a machine code. The way to do that is by inserting a first line beginning with `#!` (pronounce 'hash-bang'), followed by the path and the file name of the program that is supposed to execute your code (i.e. in our case the path of the shell, such as `/usr/local/gnu/bin/bash` or `/bin/sh`). (In fact in the case of our examples this is not needed, because the running bash shell is the one that should execute our shell scripts. But the standard way is still to

insert this first line.)

- Then you make the file executable by typing: '`chmod +x a_simple_shell_script`'. Now you can run your program, either with typing '`a_simple_shell_script`' or with typing '`./a_simple_shell_script `', depending on your system setup (depending on whether your local directory is given in $PATH or not).

When you have a file that you want to use pretty often, it might be complicated to give always the entire path. Why not to make it into a "real" command? There is a system variable (we will speak about them later) that give you a set of paths: when you type the name of a program to be run, without determining the exact (absolut or relative) path, the Shell will look for the directories given in this variable. You can add additional paths to this variable by typing:

```
PATH=$PATH:$HOME/shellscripts
```

The meaning of this is the following: the new value of the variable `PATH` should be its actual value, followed by a column (separating the differents paths within the variable), and then you can give the new path to be added. Suppose it is a directory called `shellscripts` within your own home directory. You can save typing the exact path of your home directory by refering to this other system variable.

You might want to use arguments in your shell scripts, similarly to the arguments of the standard Unix commands: these arguments influence the task performed by the program. The way to do this is by referring to them within your shell script as $1, $2, etc. These will refer respectively to the first, second, etc. argument given aftern the script's name. The arguments will be separated by a space in the command line, unless the space is neutralized by an escape character or a quote.

Furthermore, the variable $0 in the shell script refers to the zeroeth argument of the script, which is the command name used under which the program has been called. Although this seems to be redundant, it is not. Imagine that you have more file names that are hard links of each other. In that case, the same script can be launched under different command names, and the task to be performed by the script may depend upon which file name has been used. For instance, cp and mv may be the same programs, but if mv has been used, the file is also deleted once it has been copied.

An example: a shell script containing

```
ls -l $1 | grep $2
```

will look for the second argument as a regular expression within the long list of

directory given by the first argument.

$* refers to all arguments (a list including the argument list of the script).

$# means the number of arguments used.

N-gram-based text categorization

See the web site of the previous week about N-grams.

Imagine that you work for a news agency, and that you have many-many documents entering your agency each day. It would be nice to have a program that sorts you those documents, based on language or content. Indeed, in the last 10-15 years there has been intensive research in computational linguistics in order to produce better algorithms classifying documents.

You can, for example, compare the most typical words. If the document contains many tokens of "een", then it must be a Dutch document. If the document contains "ein", then it may be German, and if it contains "une", then it should be French. If it frequently contains the word "computer" then it is about information technology, unlike if the typical word is "inflation" or "recession".

Very often the typical characteristics are not words, but N-gram of words: "stock exchange" is a 2-gram typical for economic texts, while "F. C. Groningen" is a 3-gram typical for sport.

You can also look for N-grams on the character level, especially if you want to sort your documents according to language. For instance, the trigram 'eau' is typical to French, 'sch' to Dutch or German, 'aa' to Dutch, 'sh' to English, etc.

If you are interested in this topic, for more information, please have a look at this web-site from 2002, and to the article mentioned there.

(I can also tell you more about the work that I had done on this field myself.)

Bíró Tamás:
e-mail
English web site
Magyar honlap

Last modified: Thu Jul 3 11:39:17 METDST 2003