Tekstmanipulatie, week 10

Two mottos for the day:

In Unix everything is file.
In Unix everything is text.

1. A few more commands, just for fun (?)

`echo`: diplays a line of text.

Example:

```
> echo hello word
hello word
> echo I love you!
I love you!
```

`expr`: evaluates a mathematical expression. Arguments of the expression to be evaluated, as well as the operations are given as separate arguments of `expr`. Check `man expr` for further details (logical operations, modulo, etc.) . Example:

```
> expr 3 + 5 / 2
```

2. Wild cards

UNIX gives you the possibility of using wildcards. These are:

- The character * denotes any sequence of zero or more characters
- The character ? denotes a single character
- The construct [cset] denotes any single character in the cset.

A few examples:

x*	any name beginning with 'x' (e.g. x, xold, xerxes)
x	any name containing an 'x' (e.g. x, xold, fox, maxi, xx)
x?	any two-character-long name beginning with 'x' (e.g. xx, xy, x2)
[ptks]	either the character 'p' or 't' or 'k' or 's'.
x[aeiou]	any two-character-long name beginning with an 'x' followed by a vowel (e.g. xa, xu)
x[aeiou]*	any name beginning with an 'x' followed by a vowel (e.g. xa, xaver, xerxes)
x[aeiou]*[abc]*x	any name beginning with an 'x' followed by a vowel, then any characters (or none) , then 'a' or 'b' or 'c', then any characters (none or one or more) , and ending with an 'x' (e.g. xabx, xanax, xenmnaqwx) .
.	any name containing a period
[A-Z]*	any name beginning with a capital letter.
[1-9]	any non-zero number
????	any four-character-long name
????*	any at least four-character-long name
???*[0-9.x]	any at least four-character-long name ending with a numeral, a period or an 'x'
[!T]*	any name not beginning with a capital T.

Remark: Wildcards usually don't match the '/' characters referring to subdirectories, neither the initial period of some special file names.

How to use them?

`> rm *x*` : removes all files matching the expression (e.g. 'xerxes', 'maxi', 'fox')
`> cp x[aeiou] My_Directory` : copies files 'xa', 'xe',..., 'xu' to

'My_Directory'.
`> ls ????` : lists all files whose name contains exactly four characters.
`> ls apple/???*` : lists all files in the directory 'apple' whose name contains at least three characters.
`> mv /[0-9]* .` : moves all files in the root directory whose name starts with a number into the actual directory.
`> ls ~/*[0-9]*` : lists all files in the home directory whose name contains a number.
`> ls ../[A-Z][A-Z][A-Z]` : lists all files in the parent directory whose name contains exactly three capital letters.
`> ls ./*` : same as simply `ls`. Why?

!! Never do anything like 'rm a_*' ! (where '_' stands for a space...)

How does this work? Actually an expression containing a wildcard describes a set of characters. For instance: `x[aeiou] = {xa, xe, xi, xo, xu}`. A file name matches a wildcard expression iff it is element of the set.

When the shell encounters an expression with a wildcard in the command line, it automatically replaces it with the list of the matching file names (in the current directory), except if there is no matching file name.

The command `rm` (or `ls`) can receive multiple arguments and would remove / list all of them. Therefore in the case of a command line like `rm a*` (or `ls a*`) what happens, is the following. First, the pre-processor of the shell replaces the expression with whatever files names matching it, so you get something like `rm apple ananas amerika a1445`. Then this command is executed, deleting (listing) all of them.

The command `mv` and `cp` have two possibilities. If they receive exactly two arguments which are file names, then they move or copy the first one into the second one. But if they receive an (unbounded) list of file names followed by the name of a directory, then they move / copy all files in the list into the directory, retaining their names. This is how e.g. `cp x[aeiou] My_Directory` works. But if the last name is not a directory, this does not work. Why? Therefore it is not possible e.g. to rewrite the first character of all file names from 'a' to 'A' by giving the command `mv a* A*` .

Similarly, in order to concatenate all your files with a name beginning with 'a', you can simply use:

```
cat a*
```

This preprocessing by the shell takes place before running the actual command (commands, if more than one in a command line), independently of the command used. Therefore the shell does this for all commands, even if the command does not expect any file names as arguments.

For example:

`> echo What does the * symbol stands for ?` : what would this command line do?
`> echo *x*` : lists all file names matching the expression (e.g. 'xerxes', 'maxi', 'fox')
`> expr 3 * 4` : this returns you an error message. Why?

3. Metacharacters, escapes, quotes

Let's try to calculate how much is (3 + 4) * 7? We would like to have something like
`> expr ( 3 + 4 ) * 7`

But this won't work. How can we overcome this problem?

The problem is that some characters have special meanings, those characters are called **metacharacters**. We have seen so far the special meanings of the characters '*', '?', '[' and ']'. The *space* has also a special meaning: it is the delimiter between two arguments of a command, therefore a file name containing a space will also cause problems under all versions of Unix. How to **escape** from the special meanings of the metacharacters?

The way is to introduce further metacharacters which will neutralize the effect of the metacharacters. There are two types of neutralizing characters. The **escape character** `\` (backslash) neutralizes the effect of the following character. Whereas the two types of **quotes** (**`'...'`** and **`"..."`**) neutralize all the metacharacters within them.

Note that the escape characters are also metacharacters, since they also have special meanings. Thus, you have to escape the escape characters, whenever you want to use them in their original sense (as a simple character) . Summing up, we have so far at least the following metacharacters: *, ?, [,] (used in wild cards) , space (separating the arguments of a command) , ", ', \ (escape symbols) .

Remember also that at the end of last week's lecture note we had a list of characters to be preferably avoided in file names: in fact, most of them are metacharacters, bearing some special meaning, and this is why it is easier to avoid them. Nonetheless, you may use them in file names (see also the assignments this week) , once you escape them.

Now, a few examples: what is the difference between the following two commands?

```
echo What does the * symbol mean?
echo What does the \* symbol mean\?
```

Now these should work:

```
> expr 3 \* 7
21
> expr 3 '*' 7
21
> expr "3 * 7"
3 * 7
```

The problem with the latter is that the command has only one argument, because you also escape the two spaces which should differentiate between the three arguments (3, * and 7) . This does not work either:

```
> expr ( 3 + 4 ) * 7
```

becauses the parantheses are also metacharacters. Further metacharacters are: (,) , & , < , > , | , etc. The solution therefore is something like

```
> expr \( 3 + 4 \) "*" 7
```

Remark: the difference between the apostrophe ('hard quotation') and the double quotation mark ("soft quotation") will become clear when we will deal with variables. In fact if you put a variable between "-s, the variable will be replaced with its actual value, unlike in the case of hard quotations. Another advantage of having two types of quotation marks is that you can quote even the quotation marks:

```
> echo "the ' symbol is an apostrophe"
the ' symbol is an apostrophe
> echo 'The word "apple" is English'
```

```
The word "apple" is English
```

Try also these:

```
echo "Here you have 'single quotation marks' between
double ones"
echo 'And here you have "double quotation marks"
between single ones'

echo There is plenty of room
between these words
echo There is plenty of room "                 "
between these words
```

An other way to calculate something is putting the expression between '$[' and ']', like:

```
echo $[3+5] $[11/5]
```

Question: the backslash character (\) is a metacharacter itself. How to neutralize it? By typing it twice.

Note: The parameters in Unix are seperated by a space. So space is also a metacharacter, as you could see it in the above examples ("echo there is plenty of room...") , and you may want to neutralize it, too.

You could use most metacharacters in file names. For example spaces (as in Windows >95) , but you should always think of neutralizing them. Therefore it is better to avoid them. For instance, `rm file name` would remove two files, one named "file" and one named "name", and not the file whose name is "file name", if you happen to have all of them. To remove this third file, use `rm 'file name'` or `rm file\ name`, by escaping the space.

4. From characters to files, and further

Remember the two mottos of the day:

- In Unix, everything is a file
- In Unix, everything is a text

What does this mean? First, last week we learned that Unix is an operational system, and the task of operational systems is to create a bridge between the user and applications on the one hand, and the computer's hardware and periferies on the other.

Second, remember that Unix was developed in the late 1960s, in a period when computer memories were very-very limited. Therefore, it was of vital importance to store all data not immediately used in the background peripheries (magnetic bands, etc.) in files. Hence the corner stone of the *Unix philosophy*: data preferably enter each elementary programs as file, and the results are also stored as files. Furthermore, a typical Unix program reads only a smaller unit of its input file at once, it processes it, and writes the result as an output, before reading the next unit of the input file. Consequently, it does not matters how big a file is, it could be processed 30 years ago even by the computers having only a very restricted memory. As we shall soon see, the "basic units" of a file are *lines*. Most of the utilities we are going to use process their input file line after line: they stop reading their input once they have reached a *new-line character*, and do not return back to their input file as long as they have not finished processing the previous line.

Third, there are many-many files, possibly thousands of them, that are available to the operational systems. Some of them are on a winchester, others are on a floppy drive or another, remote, machine. Subsequently, the idea in Unix is to organize all files into *one* hierarchical tree, as we learnt it last week. Do not forget that this is just a virtual tree: by mounting a floppy drive or the winchester of another computer, you can even incorporate files into that tree that does not have anything to do phyisically with your hard disk, or even with your system. However, once you have so many files, why not making *everything* a file, at least virtually? This is why, for instance, different terminals and konsoles have a corresponding file: you can write on a terminal by writing into that file.

So far about the motto "everything is a file". Nevertheless, what is a file? A file is nothing but a list of numbers, each number being represented on 8 bits, so each number has to be between 0 and 255. And yet, a file can be much more than just a list of numbers. The key idea is to realize that this list of numbers can be interpreted in several different ways. If the numbers, for instance, represent a sequence of commands that may be understood by the machine (i.e., the numbers can be interpreted as a program in the machine code), then the file is said to be an *executable file*, namely, a program that you can run.

The second typical way to interpret this list of numbers is to see it as a text. The second motto, viz. "in Unix, everything is a text", expresses the central role of this second way of interpretation. With the exception of executable files, you almost always use texts. For instance, imagine that you run a simulation the result of which is a list of 100 numbers between 10 and 99. Theoretically, you have two possibilities. Either you write a file of 100 bytes, and the value of each byte is a binary number. Or, you write a much longer file, three times longer, in which you represent an output on two bytes (one byte for each digit, so if "56" is the output of your program, first you write '5', and then '6'), followed by a space or a new-line-character. Which one would you choose? The first solution creates a much shorter file, and yet, most people go for the second one: because the second solution produces a *human readable* format. In addition, the second format is much more flexible, if you happen to have outputs higher than 255.

Summarizing, it would be nice to be able to interpret a file as a text. Once you have that, you can use this way of encoding for any file that should be human readable: a sequence of numbers, a plain text, an email, a program code, etc. A compiler may then turn the program code written by you into a non human readable executable file, and similarly, a LaTeX code written by you into a non human readable graphical format.

In the following sections, we will go through the consequences of this idea. First, you have to introduce some standard way to encode a text. What should be the way of interpreting a sequence of numbers as a text? Each number between 0 and 255 should refer to a character - but how? And, vice-versa, if you want to encode a text, you have to find out a standard way to encode the character "a", the character "b", etc. For instance, if your first byte contains the value 65, and your second one contains the value 48, you can visualize it as A0, once you have decided to represent the character 0 with the value 48, and A with 65. The other way around, if you want to encode the text A0, and you have chosen the standard just described, you can simply write 65-48.

The original standard was called ASCII, and aimed at coding the English alphabet (i.e. no diacritical marks, such as á, ö, ü, ú). Furthermore, it used only 7 bits, for the 8th bit was used as a security (parity) bit to detect possible errors: thus, the ASCII standard assigns a value between 0 and 127 to all types of characters, upper and lower case letters, numbers, space, comma, full stop, $, #, @, etc. But there are also special characters in ASCII, such as the end-of-file character (^d), end-of-line character, etc. (The end-of-line character will become very important to us: it breaks the file into smaller units, namely, into lines, and these are the smaller units on which most of our commands will operate, as hinted to earlier.) Later on, new standards were developed enriching the ASCII standard (using the range between 128 and 255) in order to be able to encode other languages: languages that include diacritical marks (á, é, ü, š, œ etc.) , or even non-Latin characters. However, this proliferation of standards has created new problems. Recently, Unicode seems to be the solution: instead of one byte, a character is coded on two bytes, and 65536 possibilities seem to be enough to encode most languages of the world. Due to the growth in the size of available computer memory and drives, using two bytes for a character instead of one does not create problems any more.

In the second step, we will discuss the different ways for visualizing and editing a file seen as a text. The third point on our agenda, from next week on, is to get acquainted with different commands that *manipulate texts* ("tekstmanipulatie") . Once we have understood that almost all files in Unix are seen as a text, it is obvious that "manipulating a text" is useful not only in humanities computing ("alfa informatica") , that is, when you have real texts (literary, legal or historical documents, web sites, manuals of electronic tools, dialogues,...) in front of you. In fact, (almost) everything you do in Unix is *text manipulation*: even if you are searching some pattern in a set of numerical data or solving some task as a system administrator.

5. Coding different alphabets.

"In UNIX everything is text": what does this mean?

Originally computers were built and used by English speaking people. English is a special language, because it (practically) doesn't use diacritical marks. What about other languages, that have all kind of strange characters (e.g. á, é , í, ó, ú, ö, ü,...) ? What about languages using not latin alphabets (Arabic, Greek, Hebrew, languages using Cyrillic alphabet, Asian languages,...) ?

You have to differentiate between the following "levels":

1. **Keyboard layout**: Which key on the keyboard is understood as what character? Different languages have different standards: e.g. English: QWERTY..., German: QWERTZ..., French: AZERTY, etc. Pressing a key sends a signal to the computer (to the central unit of the machine, and not to the computer used as a terminal!) , that is understood as a number according to

the layout set up. Pressing the left-most key in the upper raw of letters, is understood as Q if your keyboard setting is Dutch, English or German, but pressing the same button is understood as A if your keyboard setting is French. (It is needless to say that the keyboard setting is defined within your operating system, and does not necessarily corresponds to what black letter is printed physically on the keyboard.)
2. **Code page**: a number (either a signal received from the keyboard, i.e. the standard input, or a byte appearing in a file) is associated with a character (e.g. 32 is a space, 48 is the character '0', 65 is the character 'A' in the ASCII standard) . This associating process needs a coding system. On modern computers (from the late 80's) there are several coding systems, different standards for associating numbers with characters. These standards are called code pages.
3. **Font**: the exact graphical image of different characters (e.g. Times, Times New Roman, Arial, Courier, Helvetica,...) . Originally fonts were defined using a matrix of points (either points of the screen or points of a plotter / printer) . (E.g. Commodore 64 used a matrix of 8 × 8 points.) Nowdays people use "vector-graphic" fonts, that allow enlarging without loosing the quality of the text. "Bold", "italic", "underlined", etc. multiplies the possibility of fonts.

For instance, what happens when you type the left most key in the second row? An electric signals goes from the keyboard to the operating system. That signals tells which key has been typed. Then, the operating system interprets this signal, based on several factors. First, the operating system has to decide whether it expects you to type any character at all? Suppose that you are not running any other program, but shell is waiting for you to type in some commands. Then, the system has to determine which character you aimed at. If your keyboard layout has been set to English, then the operating system will interpret you typing that key as typing the character 'q' (not 'Q', since the shift key was not pressed simultaneously) . So far so good. The ASCII code of 'q' is 113, so a byte containing the value 113 appears as the standard input of the shell, and hopefully shell will be able to interpret it somehow. On the other hand, the operating system decides that that character has to appear also on your screen. Then a byte containing the value 113 is written to the file which corresponds to your screen. The content of this file is interpreted according to the ASCII standard, so 113 means the character 'q'. But this information is not enough to make the character 'q' appear on your screen: the exact shape of the character (q or q or **q** or something else) depends on the setting of your terminal.

Standard code pages are:

ASCII: American Standard Code for Information Interchange, using 7 bits. (The 8th bit used to be a parity or security bit, to check whether something went wrong.)

ANSI-standards: American National Standards Institute

e.g. ANSI-1252: Latin1 for Windows, ANSI-1250: Central- and Eastern-European characters

ISO-8859 series: International Standards Organization
e.g. 8859-1: Latin-1 (Western European languages),
8859-2: Central- and Eastern-European languages,
8859-4: Baltic Languages,
8859-5: Cyrillic languages
8859-6: Arabic
8859-7: Greek
8859-8: Hebrew
8859-11: Thai (planned?)
8859-15: Latin-1, including the Euro-symbol

Unicode (ISO-10646): it uses two bytes for one character (possible by now, due to the increased memory of computers, and increased storing capacity), therefore there are not 128 (c.f. ASCII) or 256 (c.f. ANSI, ISO), but 65,536 possibilities. This makes possible to use more than one code page in the same time. In the past it was very difficult to edit a document containing more languages with different alphabets.

Here is a link to Egyptian hieroglyphs, using Unicode, as well as about coding different "exotic" languages in different ways.

Furthermore, it is useful to differentiate between two types of characters:

- Printable characters: letters, digits, punctuation marks, graphical symbols, white space, tab, etc.
- Non-printable characters: they are sort of 'commands' depending on the system, like ring the bell, delete the next / previous character, end-of-line (new-line, move-home), end-of-file (in some systems), change font, etc. In the ASCII standard these characters have codes between 0 to 31.

Tip: try out 'man ascii', and check its "see also chapter", too!

6. Text editors (vi, pico, emacs, xemacs), e-mailing (pine, xemacs)

How can we visualize and edit the content of a file? The key idea, once again, is to interpret the file as a text. Independently of whether the file is really a text or not, displaying the file as a text is probably the simplest way. To be sure, it is much simpler than displaying the file as a series of numbers.

First, we have to distinguish between four types of programs:

1. `cat`: This commmand does nothing but sends the content of a file to somewhere. This may sound quite vague now, but next week we are going to be more precise. Suffice to say that the syntax `cat filename` sends the content of the file to the screen. Obviously, if the file is too long, you will only see the end of the file on the screen.
2. pg, more, `less`: these programs are so-called "pagers". Unlike `cat`, they help you read the file page after page. The original Unix program was pg. An improved program was more, called as such because it knows more. Then came `less`: this latter program knows even more, and was called thus out of irony.
3. **Text editors 1**: unlike the pagers, the main task of these programs is not simply to display you the content of a file, but also to help you edit it. First came `vi`, and it is still used by real Unix gurus. Then came many other programs. You are expected to get familiarized with `pico` (the text editor of the emailing program `pine`), as well as withemacs and xemacs. Nonetheless, much more text editors exist: for instance, many people like `joe` (it creates a file called 'deadjoe', and the most important combination to remember is ^K-H, which gives you the help). This first set of text editors could be also called as "**file editors**": they display the content of the file by interpreting each byte as a character, following the ASCII standard (or one of its extensions).

4. **Text editors 2**: These are the text editors most of you have been probably the most familiar with, such as MS Word. Under Unix you have programs such as Open Office. These editors, in fact, do not display you the content of a file directly. Rather, they interpret the content of a file according to some rules. Some of the bytes may contain text, but others contain information about the format (character size, font, colours, paragraph identing, margins, location of pictures and the pictures themselves, etc.) . When MS Word opens a .doc-file, you cannot know what is the first byte of the file, what is the second byte, etc., but what you see is already the "interpretation" of the file as understood by the application. Similarly, you can look at the html- source code of a web site, by your browser shows you what the file "means". The different ways of mapping the "meaning" (the information about the edited text) to a file (a series of bytes) is called "format": e.g. Word 2, Word 6, Word 97, WordPerfect, HTML, rtf, etc. Obviously, you can use a text editor of this second type as a file editor (first type text editor) if you save the file in "plain text format".

Intro to vi

Since we will need to create and to edit longer files than the ones we have created with 'cat', we need to learn how to use some text editors.

vi has been long considered to be the standard editor for Unix. Hopefully you will never need to use it, and you will always have an alternative editor available. (But you never know...) . So le't s have 10 minutes of vi, just in order to be able to appreciate any other text editors.

Start vi by typing: vi <filename>. Then don't panic!

There are three modes in vi:
- command mode: typing a character is understood as a command, and does not appear on the screen.
- input mode: now you can type in whatever text you wish into your text, to the place where the cursor is.
- status-line mode (last-line mode) : issuing long commands that will appear on the bottom line of your screen.

Changing between these modes:
- when you enter vi, you are in the command mode;
- pressing the 'Esc' button ('escape') brings you back always to command mode;

- from command mode 'a' or 'i' brings you to input mode;
- from command mode ':' brings you to the last-line mode.

In the input mode you can just type in your text, but you will sometimes be surprised that you are not able to delete it. Then go back to the command mode (by using Esc) and bring the cursor onto the character you wish to delete. The just press 'x'. Then 'a' or 'i' will bring you back to input mode.

In fact 'x' deletes the given character by putting it into a buffer ("cut") . 'y' will put the given character into the buffer without deleting it ("copy") , and 'p' will paste it to the actual position of the cursor. If you wish to put more than one character into the buffer (e.g. copying or moving an entire word) , then just type the number of characters before 'x' or 'y'. For instance '5x' will delete five characters, and put them into the buffer. Finally, 'dd' will delete you a line, and put it into the buffer. If you get lost, just don't panic...

By pressing ':', you get to the last-line mode. Pressing 'vi' will start a new file ("new document") , 'vi <file_name>' opens the mentioned file. Typing 'w' will save (write) ou file, while typing 'w <file_name>' saves as (under) the given name.

Leaving this sadistic editor is possible by typing 'q' (quit) , 'q!' (quit, even if not saved) or by 'wq' (save and quit) .

The remaining beauty of vi (like different variations of the mentioned commands, further commands, searching, using the 36 buffers, etc.) are left for those of you who have some masochistic inclinations...

Intro to pico

Therefore let's rather try out another editor. This is 'pico', the text editor of the emailing program called 'pine'. Just run 'pico <file_name>' Alternatively, you can simply run 'pico' if you wish to start a new file: it is enough to give a name to it when you save the file before leaving. And now, enjoy! (Compared to vi...)

(By the way: both in the case of vi and of pico, the file name given when launching the program should not necessarily exist before hand. If it already exists then you can continue editing it, while if it doesn't, then the file will be created as an empty document.)

In pico you always have the list of commands in the last lines of your screen. ^ stands for the CTRL-button. Thus you can ask for help with ^G (CTRL + g), cut a line with ^K (paste it at the same place or to another place with ^U), etc. You can insert the content of another file to the actual position of the cursor with ^R, search for a string of characters (e.g. a word) in your longer file with ^W, etc. With ^Y and ^V you can jump one page backwards or forwards. ^T will even check your English spelling... (I wouldn't trust it too much,...) When you are done, ^O saves your file, and ^X exits. If exiting without having saved it, you are asked if you want to save your file. Finally, ^C is the universal escape-combination.

Is it too much information? Don't worry, the only thing to remember is allways to check the last lines of the screen.

Intro to pine

Once you are familiarized with pico, just type pine, and you have entered an emailing program. The logic is exactly the same, just always check the menu: on the front page you have your main menu (typing 'M' will bring back to this), otherwise you have the menu on the bottom of the screen. (You usually don't need to type the CTRL button, in this case you don't have the ^ symbol in your menu.)

From the main menu just type L to get into your folder list. Then it is up to you to create new folders (with 'A') or to delete them. Then you have your emails within your folders, that you can delete (D), undelete (U), save to another folder (S) or export to a file (E), forward them (F), reply to them (R), etc. Note the option O that will list you another dozen of commands.

The only way to really learn it is just by playing with it!

Emacs and XEmacs

Emacs is a text editor with more options than pico, and easier to use than vi. Although one has to get used to it...

But there is a tutorial (CTRL-H T), and by going through it (with a lot of exercises) you will get the practice...

XEmacs is a more modern version of Emacs.

You can use them also for writing and reading emails.

Run XEmacs by typing xemacs. Then read the manual for more information. Don't forget to ask me for a synopsis of the commands.

Bíró Tamás:
e-mail
English web site
Magyar honlap

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