LAB WORK 08.

UNIX REGULAR EXPRESSIONS AND FILTERS.

1. PURPOSE OF WORK

- Get the basic concept of regular expressions.
- Learn to use regular expressions in egrep command.
- Acquire skills of working with filter-programs.

2. TASKS FOR WORK

2.1. Syntax of Regular Expressions.

2.2. The practice of Regular Expressions. (Fill in table 1 and table 2)

2.3. Learning the work of Filter-Command.

3. REPORT

Make a report about this work and send it to the teacher's email (use a docx Report Blank).

REPORT FOR LAB WORK 08: UNIX REGULAR EXPRESSIONS AND FILTERS.

Student Name Surname	Student ID (nV)	Date

3.0. Generate Your Variant Nr.
3.1. Insert Completing Table 1. Regular Expressions understanding.
3.2. Insert Completing Table 2. Regular Expressions creation.

4. GUIDELINES

4.0. GENERATE YOU VARIANT NR.

a) Write your Surname in the letters of the English alphabet. Must be at least 7 letters, if not enough, then add the required number of letters from the Name (if not enough, then repeat Surname and Name).

For example, for Li Yurijs there will be LIYURIJS.

b) Replace the first 7 letters with their ordinal numbers in the alphabet.

For example, 12 09 25 21 18 09 10.

c) Consistently add these 7 numbers.

For example, (12 + 09 + 25 + 21 +18 + 09 + 10) = 104

d) The resulting will be your variant Nr.

For example, Variant Nr = 104

4.1. Syntax Regular Expressions.

4.1.1. Regular Expressions Description.

Regular Expressions (REs) definition.

To find some text, sometimes it is necessary to formulate complex queries according to the pattern. Many of the utilities with editing capabilities use the standard set of special characters when searching for a pattern. A pattern containing such special characters is called a regular expression (RE - Regular Expression).

The concept arose in the 1950s when the American mathematician Stephen Cole Kleene formalized the description of a regular language. RE allow you to search for the form: find all four-letter words starting with d; or find all strings containing real numbers; or find all lines starting with the correct IP address and etc.

Utilities and programs supporting REs:

- editors: emacs, vi, vim, ed, sed, ex, emacs;
- filters: grep, egrep, more, less;
- command processors: Korn Shell;
- special tools: expr, less, flex, Expect;
- scripting languages: Perl, PHP, Java Script, TCL, Java, Python, awk;
- programming environment: Delphi, MS Visual C ++.

Components of REs

- escape sequences (meta-sequence);
- single characters;
- character classes;
- quantifiers (or factor, or multiplayer);
- fixations or statements symbols;
- alternative match patterns;
- back references;
- additional constructions (re-extensions).

4.1.2. Regular Expressions Syntax.

Meta-sequence - several consecutive characters that together form a specially interpreted meaning. For example, “\k” or “\.” or “\\”.

Atom - RE element that has a nonzero width: symbol, symbol class, symbol group, meta-sequence forming a symbol. For example, the element "[a-zA-Z0-9]" or “.” is an atom, and “^” is not an atom.

Basic Regular Expression (BRE) POSIX standard, consists of RE components found in any utility / program that works with RE.

Extended Regular Expression (ERE), special components that are not present in every utility / program that uses the RE mechanism.

Basic Regular Expressions Elements.

Element	Description
`c`	Character. Simple, not special, symbol. Corresponds to oneself.
`c1c2c3`	Character Sequence. A sequence of consecutive characters that does not form a meta-sequence. Corresponds to itself.
`.`	Dot. Any character. Matches any single character.
`$`	Dollar End of line. If it is at the end of RE or sub-RE, then it corresponds to the position “end of line”.
`^`	Caret. Start of line or inverse of class. If it is at the beginning of RE or sub-RE, then it corresponds to the position “beginning of line”. If it stands first in the description of a character class, it means the inversion of this character class. Otherwise, it corresponds to itself.
*	Star Multiplier> = 0. Corresponds to no or more instances of the atom standing directly in front of it.
\	Backslash. Very powerful symbol. It can cancel the value of any other metacharacter or, conversely, form a metaserial together with a suitable character.
`[c1c2c3]`	Character class. The character class specified by the enumeration. Inside the class, the action of any metacharacters, except for “^”, “-“, “[”, “:” and “\”, is canceled. Matches one of the characters listed.
`[c1-c2]`	Character class. A character class defined by a character range from c1 to c2. Matches a single character belonging to a given range.
`[^c1c2c3]`	Character class. Inverse character class specified by enumeration. Matches a single character that does not belong to the class [c1c2c3].
`[^c1-c2]`	Character class. Inverse character class specified by a range of characters. Matches a single character that does not belong to the class [c1-c2].
`[c1c2-c3c4]`	Character class. A character class defined in a mixed way.

Extended Regular Expressions Elements.

Element	Description
\< and \>	Word Boundaries. Corresponding positions: beginning of the word “\ <”; end of the word “\>”; the whole word is "\ <word \>". By "word" here is meant a sequence of non-whitened atoms.
\b	Word Boundaries. Corresponds to the position between whitespace and non-whitespace, as well as the position at the beginning or end of a line.
\B	Non-word Boundaries.
(and)	Buffer grouping.
\(and \)	The same as the previous one for BRE mode.
|	Bar. A disjunction operator (or operation) that allows you to combine any two or more regular subexpressions so that the resulting regular expression matches any string that matches any of the subexpressions.
\\|	The same as the previous one for BRE mode.
+	Plus Multiplier> 0. Corresponds to one or more instances of the atom standing directly in front of it.
\+	The same as the previous one for BRE mode.
?	Question Multiplier. Availability factor. Corresponds to no or one instance of the atom standing directly in front of it.
\?	The same as the previous one for BRE mode.
{n,m}	Universal Multiplier. Matches from n to m instances of the atom standing directly in front of it. There are restrictions on the value of n and m, for example, in perl their value does not exceed 65535. Use cases: {n} - strictly n repetitions of an atom; {n,} - n or more repetitions of an atom; {0,} - is equivalent to the factor *; {1,} - is equivalent to the factor +; {0,1} - is equivalent to the factor ?.
\{n,m\}	The same as the previous one for BRE mode.
\k	Backreference. The operator allows you to access the substring previously stored in the buffer, which coincided with the subpattern. k is the number of the buffer. In perl k <= 65535, for other programs <= 9.
[[:class:]]	Character class. A predefined character class. Matches a single character from a named character class. Supports localization. For example: [[: alpha:]] - any alphabetic character, ie letter; [^ [: xdigit:]] - any character that is not a hex-digit. [[:blank:]] – space and tab. [^[:lower:]ABC[0-9]] – none lowercase letters and none ABC and none 0-9

4.1.3. REs examples.

Pattern (RE)	Interpretation
`example`	example anywhere on the line
`^example`	example at the beginning of the line
`example$`	example at the end of the line
`^example$`	example as a separate line
`\<example\>`	example as a single word
`example.$`	at the end of the line there is example and another character
`example\.$`	at the end of the line there is example and another point
`$example`	character sequence $ and example
`example^`	sequence of example and character ^
`example*`	example or exampl or exampleeee
`[eE]xample`	example or Example
`example[0-9]`	example followed by one digit
`example[^0-9]`	example followed by one non-numeric character
`example[a-zA-Z]`	example followed by one latin letter
`example[[:alpha:]]`	example follows one letter according to l10n
`example1.*example2`	example1, then 0 or more characters, then example2
`^example1.*example2$`	the line starts with example1 and ends with example2
`example\.\.\.$`	at the end of the line example and ellipsis
`^$`	empty string, because starts and ends right there
`.`	non-empty string, i.e. having at least 1 character
`.*`	any line: empty and nonempty
`^`	any line, because any line “begins”
`$`	any line, because any line “ends”
`^\$`	line starts with dollar
`$$`	the line ends with a dollar
`X*`	any line, since 0 repetitions of X are enough
`XX*`	a line with at least one X
`\\`	line in which there is \
`[0-9]`	line in which there is a digit
`[0-9][0-9]*`	corresponds to the maximum series of digits
`\<\...\.\>`	a 4-character word starts and ends with “.”
`\<.*([a-z])\1.*\>`	double word

4.1.4. REs Interactive Tutorial.

Read, understand and do 15 simple RE exercises and 8 tasks at site https://regexone.com

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interactive Tutorial References & More

Lesson 1: An Introduction, and the ABCs

Regular expressions are extremely useful in extracting information from text such as code, log files, spreadsheets, or even documents. And while there is a lot of theory behind formal languages, the following lessons and examples will explore the more practical uses of regular expressions so that you can use them as quickly as possible.

The first thing to recognize when using regular expressions is that **everything is essentially a character**, and we are writing patterns to match a specific sequence of characters (also known as a string). Most patterns use normal ASCII, which includes letters, digits, punctuation and other symbols on your keyboard like %#$@!, but unicode characters

Lesson Notes

abc...	*Letters*
123...	*Digits*
\d	*Any Digit*
\D	*Any Non-digit character*
.	*Any Character*
\.	*Period*
[abc]	*Only a, b, or c*
[^abc]	*Not a, b, nor c*

All Lessons

- Lesson 1: An Introduction, and the ABCs
- Lesson 1½: The 123s
- Lesson 2: The Dot
- Lesson 3: Matching specific characters
- Lesson 4: Excluding specific characters
- Lesson 5: Character ranges
- Lesson 6: Catching some zzz's
- Lesson 7: Mr. Kleene, Mr. Kleene
- Lesson 8: Characters optional
- Lesson 9: All this whitespace
- Lesson 10: Starting and ending
- Lesson 11: Match groups
- Lesson 12: Nested groups
- Lesson 13: More group work
- Lesson 14: It's all conditional
- Lesson 15: Other special characters
- Lesson X: Infinity and beyond!

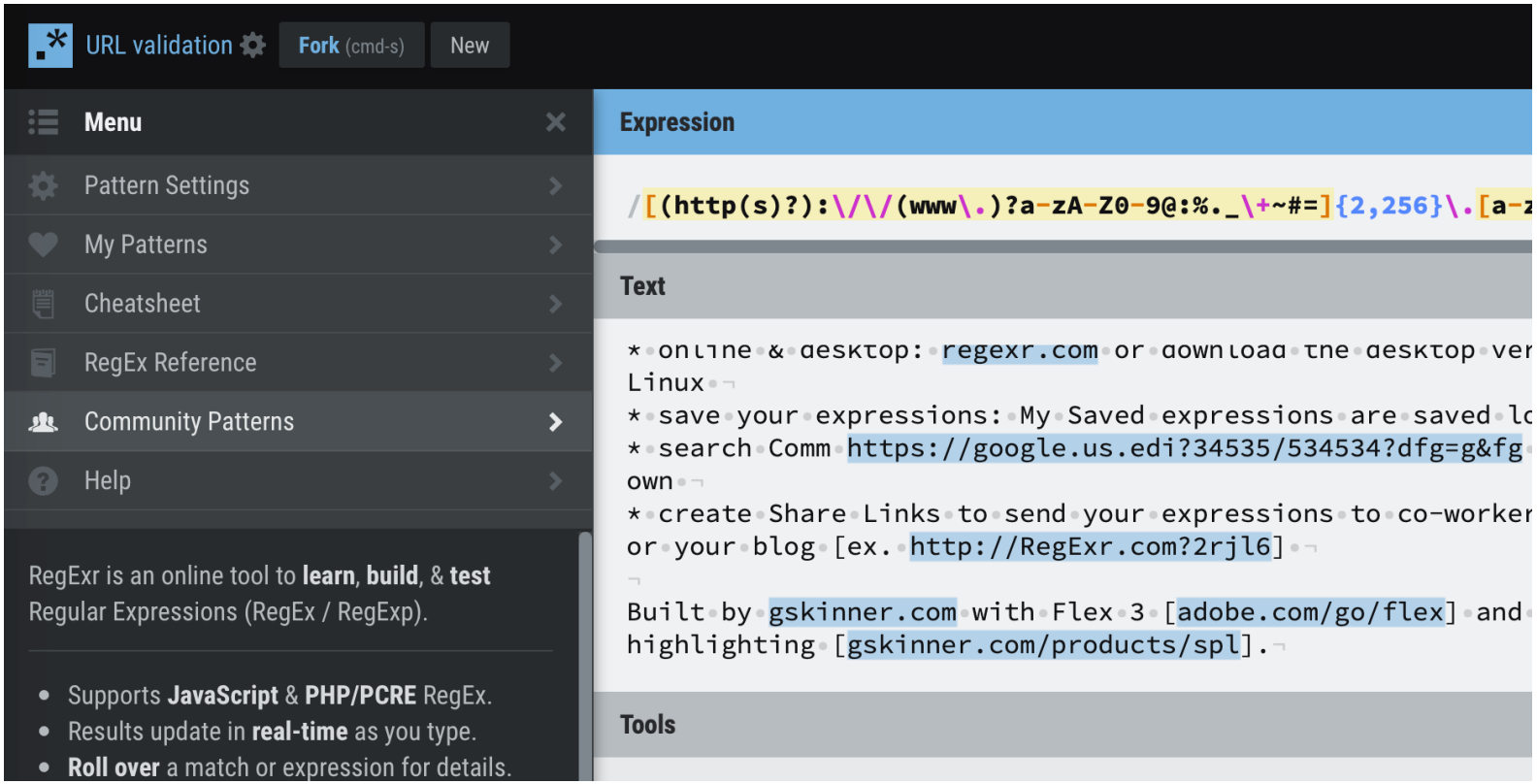
You've finished the tutorial!

4.2. The practice of Regular Expressions.

4.2.1. REs Online Constructor.

Task 1. Fill in the Table 1.

Use REs Online Constructor on site **https://regexr.com** to create and test the REs exercises below Table 1.



Remark 1.

In all of the below, the question is, does the regular expression match the full string.
Slash (/) is the delimiter character showing where the regular expression begins and ends.
Strings to be matched start and end with non-blank characters: there are no leading or trailing blanks.

Remark 2. Select odd questions Nr for odd Variant Nr; even questions Nr for even Variant Nr

Table 1. REs understanding.

Nr	Task Description	Your Answer
1. a) b) c) d) e)	Which of the following matches regexp /a(ab)*a/ abababa aaba aabbaa aba aabababa	
2. a) b) c) d)	Which of the following matches regexp /ab+c?/ abc ac abbb bbc	
3. a) b) c) d) e) f)	Which of the following matches regexp /a.[bc]+/ abc abbbbbbbb azc abcbcbcbc ac asccbbbbcbcccc	
4. a) b) c)	Which of the following matches regexp /abc|xyz/ abc xyz abc|xyz	
5. a) b) c) d) e) f) g)	Which of the following matches regexp /[a-z]+[\.\?!]/ battle! Hot green swamping. jump up. undulate? is.?	

6.	Which of the following matches regexp /[a-zA-Z]*[^,]=/	
a)	Butt=	
b)	BotHEr,=	
c)	Ample	
d)	FIdDlE7h=	
e)	Brittle =	
f)	Other.=	
7.	7 Which of the following matches regexp /[a-z][\.\?!]\s+[A-Z]/ (\s matches any space character)	
a)	A. B	
b)	c! d	
c)	e f	
d)	g. H	
e)	i? J	
f)	k L	
8.	Which of the following matches regexp /(very)+(fat)?(tall|ugly) man/	
a)	very fat man	
b)	fat tall man	
c)	very very fat ugly man	
d)	very very very tall man	
9.	Which of the following matches regexp /<[^>]+>/	
a)	<an xml tag>	
b)	<opentag> <closetag>	
c)	</closetag>	
d)	<>	
e)	<with attribute=”77”>	
10.	Which of the following matches regexp /\bb[ou]y\b/	
a)	bbouy man	
b)	bouy man	
c)	very fat boy man	
d)	very tall buy	
e)	tail buoy	

4.2.2. REs egrep Creation Practice.

4.2.2.0. Examples grep (egrep) commands options usage.

The most common use of **grep (egrep)** is to filter lines of text containing (or not containing) a certain string. Command egrep – extended grep.

```
$ cat tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
$ grep Williams tennis.txt
Serena Williams, usa
Venus Williams, USA
```

One of the most useful options of grep is **grep -i** which filters in a case (ignore registry) insensitive way.

```
$ grep Bel tennis.txt
Justine Henin, Bel
$ grep -i Bel tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
```

Another very useful option is **grep -v** which outputs lines not matching the string.

```
$ grep -v Fra tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
```

And of course, both options can be combined to filter all lines not containing a case insensitive string.

```
$ grep -vi usa tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
```

With **grep -A1** one line **after** the result is also displayed.

```
$ grep -A1 Henin tennis.txt
Justine Henin, Bel
Serena Williams, usa
```

With **grep -B1** one line **before** the result is also displayed.

```
$ grep -B1 Henin tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
```

With **grep -C1** (context) one line **before** and one **after** are also displayed. All three options (A,B, and C) can display any number of lines (using e.g. A2, B4 or C20).

```
paul@debian5:~/pipes$ grep -C1 Henin tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
```

Example of using REs in grep (egrep).

NOTE. Start Your **UbuntuMini** Virtual Machine on your VirtualBox.

The shell script below shows lines of the file under test that contain syntactically valid IPv4 addresses from 0.0.0.0 to 255.255.255.255, with possible leading zeros in each octet. The test file is set as a script parameter $1.

Example of test-file content:

```
Right string. Abcdef 192.168.1.1 dfghj
Bad string. 10.10.10.10asdfgh
Bad string. 192.168.1.256 dfghj
```

Script execution command:

```
$ ./egrep-script test-file
```

Egrep-script file content:

```
#!/bin/sh
egrep "\<\
([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\
([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\
([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\
([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\
\>" $1
```

Here at the end of each line is the escape character “\” of the line feed character for the shell. This screening works for the shell and allows you to arrange the RE in several lines, which makes it more readable.

4.2.2.1. Time finding.

Select Your sub-task Nr = (Your Variant Nr) mod 4 + 1. Example for Var.Nr=104 → 104 mod 4 + 1 = 0 + 1 = 1.

0. Use egrep command for create and test Your REs, allowing to find the correct time in the format Mm.Ss (00.00 – 59.59).

   Remark. Create pattern with possible leading zeros in each field (for example, 59.01 or 59.1 or 01.00 or 01.0 or 1.0).

1. Use egrep command for create and test Your REs, allowing to find the correct 24 clock time in the format Hh:Mm:Ss (00:00:00 – 23:59:59).

   Remark. Create pattern with required leading zeros in each field (for example, 23:00:07).

2. Use egrep command for create and test Your REs, allowing to find the correct 12 clock time in the format Hh:Mm.Ss (00:00.00 – 11:59.59).

   Remark. Create pattern with required leading zeros in each field (for example, 11:00.07).

3. Use egrep command for create and test Your REs, allowing to find the correct 24 clock time in the format Hh:Mm (00:00 - 23:59).

   Remark. Create pattern with possible leading zeros in each field (for example, 23:01 or 23:1 or 03:00 or 03:0 or 3:0).

4. Use egrep command for create and test Your REs, allowing to find the correct 12 clock time in the format Hh:Mm (00:00 - 11:59).

   Remark. Create pattern with possible leading zeros in each field (for example, 11:01 or 11:1 or 01:00 or 01:0 or 1:0).

4.2.2.2. IP address finding.

Select Your sub-task Nr = (Your Variant Nr) mod 5 + 1. Example for Var.Nr=104 → 104 mod 5 + 1 = 4 + 1 = 5.

0. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches with every Class networks (0.0.0.0-255.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet. (See solution example before page).

1. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches all Class A networks (1.0.0.0-126.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet.

2. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches all Class B networks (128.0.0.0-191.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet.

3. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches all Class C networks (192.0.0.0-223.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet.

4. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches all Class D networks (224.0.0.0-239.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet.

5. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches all Class E networks (240.0.0.0-254.255.255.255).

   Remark. Create pattern with possible leading zeros in each octet.

4.2.2.3. Date finding.

Select Your sub-task Nr = (Your Variant Nr) mod 6 + 1. Example for Var.Nr=104 → 104 mod 6 + 1 = 2 + 1 = 3.

1. Use egrep command for create and test Your REs, allowing to find the correct date in the format Dd/Mm/YYyy (21/03/2019).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

2. Use egrep command for create and test Your REs, allowing to find the correct date in the format YYyy.Mm.Dd (2019.03.21).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

3. Use egrep command for create and test Your REs, allowing to find the correct date in the format Mm.Dd.yy (03.21.19).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

4. Use egrep command for create and test Your REs, allowing to find the correct date in the format yy.Mm.Dd (19.03.21).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

5. Use egrep command for create and test Your REs, allowing to find the correct date in the format YYyyMmDd (20190321).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

6. Use egrep command for create and test Your REs, allowing to find the correct date in the format DdMmyy (210319).

   Remark. Use leap years with 365 days (February is always 28 days). Apply only years between 1000 and 9999. Create pattern with required leading zeros in each field.

4.2.2.4. Credit Card finding.

Select Your sub-task Nr = (Your Variant Nr) mod 7 + 1. Example for Var.Nr=104 → 104 mod 7 + 1 = 6 + 1 = 7.

1. Use egrep command for create and test Your REs, allowing to find the valid New Visa card numbers start with a 4 and have 16 digits. Visa put digits in sets of 4.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

2. Use egrep command for create and test Your REs, allowing to find the valid American Express card numbers start with 34 or 37 and have 15 digits. Amex use groups of 4-6-5 digits.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

3. Use egrep command for create and test Your REs, allowing to find the valid Diners Club card numbers begin with 300 through 305, or 36, or 38. All have 14 digits. Diners Club use groups of 4-6-4 digits.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

4. Use egrep command for create and test Your REs, allowing to find the valid Discover card numbers begin with 6011 or 65. All have 16 digits. Discover put digits in sets of 4.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

5. Use egrep command for create and test Your REs, allowing to find the valid JCB cards beginning with 2131 or 1800 have 15 digits. JCB cards beginning with 35 have 16 digits. JCB put digits in sets of 4.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

6. Use egrep command for create and test Your REs, allowing to find the valid MasterCard numbers either start with the numbers 51 through 55 or with the numbers 2221 through 2720. All have 16 digits. MasterCard put digits in sets of 4.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

7. Use egrep command for create and test Your REs, allowing to find the valid Universal Electronic Card (UEC) numbers either start with the numbers 7. All have 16 digits. UEC put digits in sets of 4.

   Remark. Create pattern with possible spaces (“ “) or dashes (“-“) in card numbers.

Task 2. Fill in the table 2.

Table 2. Egrep REs creation.

Nr	Your Task Variant Nr and Text	Your Answer (RE)

Example
4.2.2.1.0.
	For example.
	0. Use egrep command for create and test Your REs, allowing to find the correct time in the format Mm.Ss (00.00 – 59.59). **Remark.** Create pattern with possible leading zeros in each field (for example, 59.01 or 59.1 or 01.00 or 01.0 or 1.0).
		For example.
		`#!/bin/sh`
		`egrep "\<\`
		`([0-9]|[0-5][0-9])\.\`
		`([0-9]|[0-5][0-9]\`
		`\>" $1`

4.2.2.2.0.	0. Use egrep command for create and test Your REs, allowing to find the correct IPv4 address that matches with every Class networks (0.0.0.0-255.255.255.255). **Remark.** Create pattern with possible leading zeros in each octet. (See solution example before page).	`#!/bin/sh`
		`egrep "\<\`
		`([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\`
		`([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\`
		`([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\.\`
		`([0-9]|[0-9][0-9]|[01][0-9][0-9]|2[0-4][0-9]|25[0-5])\`
		`\>" $1`

4.2.2.1.	Time finding. Select Your sub-task Nr = (Your Variant Nr) mod 4 + 1. Example for Var.Nr =104 → 104 mod 4 + 1 = 0 + 1 = 1. Your Task Variant Text	
4.2.2.2.	IP address finding. Select Your sub-task Nr = (Your Variant Nr) mod 5 + 1. Example for Var.Nr =104 → 104 mod 5 + 1 = 4 + 1 = 5. Your Task Variant Text	
4.2.2.3.	Date finding. Select Your sub-task Nr = (Your Variant Nr) mod 6 + 1. Example for Var.Nr =104 → 104 mod 6 + 1 = 2 + 1 = 3. Your Task Variant Text	
4.2.2.4.	Credit Card finding. Select Your sub-task Nr = (Your Variant Nr) mod 7 + 1. Example for Var.Nr =104 → 104 mod 7 + 1 = 6 + 1 = 7. Your Task Variant Text	

4.3. Learning the work of Filter-Command.

Commands that are created to be used with a **pipe** are often called **filters**. These **filters** are very small programs that do one specific thing very efficiently. They can be used as **building blocks**. The combination of simple commands and filters in a long **pipe** allows you to design elegant solutions.

cat, tac

When between two **pipes**, the **cat** command does nothing (except putting **stdin** on **stdout**). Command tac – revers cat.

```
$ tac count.txt | cat | cat | cat | cat | cat
four
three
two
one
```

tee

Writing long **pipes** in Unix is fun, but sometimes you may want intermediate results. This is where **tee** comes in handy. The **tee** filter puts **stdin** on **stdout** and also into a file. So **tee** is almost the same as **cat**, except that it has two or more identical outputs.

```
$ tac count.txt | tee temp.txt | tac
one
two
three
four
$ cat temp.txt
four
three
two
one
```

cut

The **cut** filter can select columns from files, depending on a delimiter or a count of bytes. The screenshot below uses **cut** to filter for the username and userid in the **/etc/passwd** file. It uses the colon as a delimiter, and selects fields 1 and 3.

```
$ cut -d: -f1,3 /etc/passwd | tail -4
Figo:510
Pfaff:511
Harry:516
Hermione:517
$
```

When using a space as the delimiter for **cut**, you have to quote the space.

```
$ cut -d" " -f1 tennis.txt
Amelie
Kim
Justine
Serena
Venus
$
```

This example uses **cut** to display the second to the seventh character of **/etc/passwd**.

```
$ cut -c2-7 /etc/passwd | tail -4
igo:x:
faff:x
arry:x
ermion
$
```

tr

You can translate characters with **tr**. The screenshot shows the translation of all occurrences of e to E.

```
$ cat tennis.txt | tr 'e' 'E'
AmEliE MaurEsmo, Fra
Kim ClijstErs, BEL
JustinE HEnin, BEl
SErEna Williams, usa
VEnus Williams, USA
```

Here we set all letters to uppercase by defining two ranges.

```
$ cat tennis.txt | tr 'a-z' 'A-Z'
AMELIE MAURESMO, FRA
KIM CLIJSTERS, BEL
JUSTINE HENIN, BEL
SERENA WILLIAMS, USA
VENUS WILLIAMS, USA
```

Here we translate all newlines to spaces.

```
$ cat count.txt
one
two
three
four
five
$ cat count.txt | tr '\n' ' '
one two three four five
$
```

The **tr -s** filter can also be used to squeeze multiple occurrences of a character to one.

```
$ cat spaces.txt
one    two        three
     four   five  six
$ cat spaces.txt | tr -s ' '
one two three
 four five six
$
```

You can also use **tr** to 'encrypt' texts with **rot13**.

```
$ cat count.txt | tr 'a-z' 'nopqrstuvwxyzabcdefghijklm'
or
$ cat count.txt | tr 'a-z' 'n-za-m'
bar
gjb
guerr
sbhe
svir
$
```

This last example uses **tr -d** to delete characters.

```
$ cat tennis.txt | tr -d e
Amli Maursmo, Fra
Kim Clijstrs, BEL
Justin Hnin, Bl
Srna Williams, usa
Vnus Williams, USA
$
```

wc

Counting words, lines and characters is easy with **wc**.

```
$ wc tennis.txt
  5  15 100 tennis.txt
$
$ wc -l tennis.txt
5 tennis.txt
$
$ wc -w tennis.txt
15 tennis.txt
$
$ wc -c tennis.txt
100 tennis.txt
$
```

sort

The **sort** filter will default to an alphabetical sort.

```
$ cat music.txt
Queen
Brel
Led Zeppelin
Abba
$
$ sort music.txt
Abba
Brel
Led Zeppelin
Queen
$
```

But the **sort** filter has many options to tweak its usage. This example shows sorting different columns (column 1 or column 2).

```
$ sort -k1 country.txt
Belgium, Brussels, 10
France, Paris, 60
Germany, Berlin, 100
Iran, Teheran, 70
Italy, Rome, 50
Latvia, Riga, 1
$ sort -k2 country.txt
Germany, Berlin, 100
Belgium, Brussels, 10
France, Paris, 60
Latvia, Riga, 1
Italy, Rome, 50
Iran, Teheran, 70
```

The screenshot below shows the difference between an alphabetical sort and a numerical sort (both on the third column).

```
$ sort -k3 country.txt
Latvia, Riga, 1
Belgium, Brussels, 10
Germany, Berlin, 100
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
$ sort -n -k3 country.txt
Latvia, Riga, 1
Belgium, Brussels, 10
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
Germany, Berlin, 100
```

uniq

With **uniq** you can remove duplicates from a **sorted list**.

```
$ cat music.txt
Queen
Brel
Queen
Abba
$ sort music.txt
Abba
Brel
Queen
Queen
$ sort music.txt |uniq
Abba
Brel
Queen
```

uniq can also count occurrences with the **-c** option.

```
$ sort music.txt |uniq -c
      1 Abba
      1 Brel
      2 Queen
```

comm

Comparing streams (or files) can be done with the **comm**. By default **comm** will output three columns. In this example, Bowie and Sweet are only in the first file, Turner is only in the second, Abba, Cure and Queen are in both lists.

```
$ cat > list1.txt
Abba
Bowie
Cure
Queen
Sweet
$ cat > list2.txt
Abba
Cure
Queen
Turner
$ comm list1.txt list2.txt
                Abba
Bowie
                Cure
                Queen
Sweet
        Turner
```

The output of **comm** can be easier to read when outputting only a single column. The digits point out which output columns should not be displayed.

```
$ comm -12 list1.txt list2.txt
Abba
Cure
Queen
$ comm -13 list1.txt list2.txt
Turner
```

od

European humans like to work with ascii characters, but computers store files in bytes. The example below creates a simple file, and then uses **od** to show the contents of the file in hexadecimal bytes

```
$ cat > text.txt
abcdefgh
12345678
$ od -t x1 text.txt
0000000 61 62 63 64 65 66 67 0a 31 32 33 34 35 36 37 0a
0000020
```

The same file can also be displayed in octal bytes.

```
$ od -b text.txt
0000000 141 142 143 144 145 146 147 012 061 062 063 064 065 066 067 012
0000020
```

And here is the file in ascii (or backslashed) characters.

```
$ od -c text.txt
0000000   a   b   c   d   e   f   g  \n   1   2   3   4   5   6   7  \n
0000020
```

sed

The **s**tream **ed**itor **sed** can perform editing functions in the stream, using **regular expressions**.

```
$ echo level5 | sed 's/5/42/'
level42
$ echo level5 | sed 's/level/jump/'
jump5
```

Add **g** for global replacements (all occurrences of the string per line).

```
$ echo level5 level7 | sed 's/level/jump/'
jump5 level7
$ echo level5 level7 | sed 's/level/jump/g'
jump5 jump7
```

With **d** you can remove lines from a stream containing a character.

```
$ cat tennis.txt
Venus Williams, USA
Martina Hingis, SUI
Justine Henin, BE
Serena williams, USA
Kim Clijsters, BE
Yanina Wickmayer, BE
$ cat tennis.txt | sed '/BE/d'
Venus Williams, USA
Martina Hingis, SUI
Serena williams, USA
```

4.4. The practice of Filter-Command.

4.4.1. Put a sorted list of all bash users (**from /etc/passwd file**) in bashusers.txt file.

```
$ grep bash /etc/passwd | cut -d: -f1 | sort > bashusers.txt
```

4.4.2. Put a sorted list of all logged on users (**who**) in onlineusers.txt.

```
$ who | cut -d' ' -f1 | sort > onlineusers.txt
```

4.4.3. Make a list of all filenames in **/etc/** directory that contain the string **conf** in their filename.

```
$ ls /etc | grep conf
```

4.4.4. Make a sorted list of all files in **/etc/** directory that contain the registry case insensitive string **conf** in their filename.

```
$ ls /etc | grep -i conf | sort
```

4.4.5. Look at the output of **/sbin/ifconfig**. Write a line that displays only ip address and the subnet mask.

```
$ /sbin/ifconfig | head -2 | grep 'inet ' | tr -s ' ' | cut -d' ' -f3,5
```

4.4.6. Write a line that removes all non-letters from a stream.

```
$ cat text1
This is, yes really! , a text with ?&* too many str$ange# characters ;-)
```

```
$ cat text1 | tr -d ',!$?.*&^%#@;()-'
This is yes really  a text with  too many strange characters
```

4.4.7. Write a line that receives a text file, and outputs all words on a separate line.

```
$ cat text2
it is very cold today without the sun

$ cat text2 | tr ' ' '\n'
it
is
very
cold
today
without
the
sun
```

4.4.8. Write a spell checker on the command line. (There may be a dictionary in **/usr/share/dict/** .)

```
$ echo "The zun is shining today" > text3

$ cat > DICT
is
shining
sun
the
today

$ cat text3 | tr 'A-Z ' 'a-z\n' | sort | uniq | comm -23 - DICT
zun
```

You could also add the solution from question number 6 to remove non-letters, and **tr -s ' '** to remove redundant spaces.

4.4.9. Here's a way to get a sorted list of the unique file extensions in the current directory, with a count of each type.

```
ls | rev | cut -d'.' -f1 | rev | sort | uniq -c
```

There's a lot going on here.

- *ls*: Lists the files in the directory
- *rev*: Reverses the text in the filenames.
- *cut*: Cuts the string at the first occurrence of the specified delimiter ".". Text after this is discarded.
- *rev*: Reverses the remaining text, which is the filename extension.
- *sort*: Sorts the list alphabetically.
- *uniq*: Counts the number of each unique entry in the list.

The output shows the list of file extensions, sorted alphabetically with a count of each unique type.