EDAF35 Lecture 4

March 27, 2020

1 EDAF35: Lecture 4

Contents: - UNIX Shell Programming - UNIX Commands

1.1 Why Shell Programming ?

- A program written for a shell is called a shell script.
- Shell scripts are (almost always) interpreted
  - *(there is a company in the USA which sold shell-compilers but they now focus on selling C++ compilers instead)*
  - *see also the Shell Script Compiler tool*
- Shell programs have some advantages over C programs:
  - More convenient to write when dealing with files and text processing.
  - The building blocks of the shell are of course all the usual UNIX commands.
  - More portable.
- However, the shell is slower than compiled languages.

1.2 Different Shells

- There are a number of shells.
- **Bourne shell** is the original but lacked many features *(e.g. name completion)*.
- The **csh** and **tcsh** have different syntax but were more advanced.
- The **Korn shell** was written at Bell Labs as a superset of Bourne shell but with modern features.
- The GNU program **Bourne Again Shell**, or bash, is similar to Korn shell.
- We will focus on bash.

1.3 Bash as Login Shell

- Every user has a path to the login shell in the password file.
- When you login, and have bash as login shell, bash will process the following files:
  - `/etc/profile`
  - First found (in `$HOME`) of `.bash_profile`, `.bash_login`, `.profile`.
- When the login shell terminates, it will read the file `.bash_logout`.

[1]:
```
cat /etc/profile
```

```
# System-wide .profile for sh(1)
```

```
if [ -x /usr/libexec/path_helper ]; then
        eval `/usr/libexec/path_helper -s`
fi

if [ "${BASH-no}" != "no" ]; then
        [ -r /etc/bashrc ] && . /etc/bashrc
fi
```

1.4 Interactive Non-Login Shell

- An *interactive shell* is, of course, one which one types commands to.
- A *non-interactive shell* is one which is executing a shell script.
- An interactive shell which is not the login shell executes the file `.bashrc`.
- There is a file `/etc/bashrc`, but it is not automatically read.
- To read it automatically, insert `source /etc/bashrc` in your `.bashrc`.

1.5 Non-Interactive Shell

- Non-interactive shells do not start with reading a specific file.
- If the environment variable `$BASH_ENV` (or `$ENV` if the bash was started as `/bin/sh`) contains a file name, then that file is read.
- The first argument to bash itself, contains the program name, so `echo $0` usually prints `bash`.

[2]:
```
echo $BASH_ENV
echo $ENV
echo $0
```

```
/bin/bash
```

1.6 `Source` Builtin Command

- To ask the current shell to read some commands use the `source filename` command.
- You can use `.` instead of `source`.

1.7 Aliases and Noclobber

- UNIX commands perform their tasks without asking the user whether he/she really means what he/she just typed. This is very convenient (most of the time).
- For instance the rm command has an option -i to ask for confirmation before a file is removed.
  - Sometimes people put the command `alias rm='rm -i'` in a bash start file.
- A similar feature is to use the command: `set -o noclobber` which avoids deleting an existing file with I/O redirection (e.g. `ls > x`).
- But remember, generally *UNIX is not a safe place*

1.8 I/O Redirection

- `< file` Use file as stdin.
- `> file` Use file as stdout.

- `>> file` Append output to file.
- `2> file` Use file as stderr.
- `2>&1` Close stderr and dup stdout to stderr.
- `cmd1 | cmd2` Use the stdout from `cmd1` as stdin for `cmd2` (aka *pipe*)

[4]:
```
#echo 'Hello' > f1
echo ' world!' >> f1
cat < f1
```

```
Hello
 world!
 world!
 world!
 world!
 world!
```

[5]:
```
ls -al f1
f1
```

```
-rw-r--r--  1 flagr  staff  46 Jan 20 13:58 f1
bash: f1: command not found
```

[6]:
```
chmod a-x f1
```

[7]:
```
ls -al f1
./f1
```

```
-rw-r--r--  1 flagr  staff  46 Jan 20 13:58 f1
bash: ./f1: Permission denied
```

1.9 Shell Script Basics

- The first line should contain the line `#!/bin/bash`
- To make the script executable, use `chmod a+x file`.
- A line comment is started with #.
- Commands are separated with newline or semicolon `;`.
- Backslash `\` continues a command on the next line.
- Parenthesis `()` group commands and lets a new shell execute the group.

1.10 More About Parentheses

- A subshell has its own shell variables such as current directory.
- The builtin `cd` does not read from stdin, so we can pipe as follows: `(cd ; ls) | (cd ~/Desktop; cat > ls-in-home)`

[8]:
```
(cd ; ls) | (cd ~/Desktop; cat > ls-in-home)
cat ls-in-home
```

```
bash: cd: ~/Desktop: No such file or directory
Applications
Box Sync
Desktop
Documents
Downloads
Dropbox
Library
Movies
Music
Pictures
Privat
Public
Qt
SimplicityStudio
Sites
Terminal Saved Output
Zotero
bin
exjobb2017_v2.csv
gcviewer.properties
git
go
moss
node_modules
package-lock.json
target
temp
```

1.11 Shell Variables

- Shell variables do not have to be declared — just assign to them:

```
$ a=unix
$ echo $a
$ b=wrong rm can have unexpected results
$ c="wrong rm can have unexpected results"
```

- The difference between the last two assignments is significant (see prepend variables definition to command).
- A shell variable is by default local to the shell but can be exported to child processes using:

```
$ export a
```

- C/C++ programs get the value using `char* value = getenv("VAR");`

[9]:
```
a=unix
echo $a
b=wrong rm can have unexpected results
echo $b
c="wrong rm can have unexpected results"
echo $c
```

```
unix
rm: can: No such file or directory
rm: have: No such file or directory
rm: unexpected: No such file or directory
rm: results: No such file or directory

wrong rm can have unexpected results
```

[10]:
```
echo $b
```

[11]:
```
x="once upon" y="a time" bash -c 'echo $x $y'
```

```
once upon a time
```

[12]:
```
echo $x
```

1.12 Using Shell Variables

- Use a dollar sign before the name to get the value: `$HOME`.
- If you wish to concatenate a shell variable and a string, use `${VAR}suffix`
  - without `{}` you get wrong identifier

[13]:
```
b=bumble
echo $b
echo ${b}bee
echo $bbee
```

```
bumble
bumblebee
```

1.13 More about Using Shell Variables

- The value of `${var-thing}` is `$var` if var is defined, otherwise `thing` were thing is not expanded. Value of var is unchanged.
- The value of `${var=thing}` is `$var` if var is defined, otherwise `thing` and var is set to thing.
- The value of `${var+thing}` is thing if `var` is defined, otherwise nothing.

- The value of `${var?message}` is `$var` if var is defined, otherwise a message is printed and the shell exits.

[14]:
```
echo ${a-something}
echo ${d-nothing}
echo $d
echo ${e=everything}
echo $e
echo ${d?Variable d not defined}
```

```
unix
nothing

everything
everything
bash: d: Variable d not defined
```

1.14 PS1 and PS2

- The prompts, `$` and `>` are called the primary and secondary prompts. These were the original values of these and they are stored in PS1 and PS2.
- For the root user, the prompt is `#`
- It is possible to get a more informative prompt by using the escapes: e.g. `PS1="\w "`
  - `\$` # if root, otherwise dollar.
  - `\!` Current history number (see below).
  - `\w` Pathname of working directory.
  - `\W` Basename of working directory.
  - `\h` Hostname.
  - `\H` Hostname including domain.
  - `\u` User.
  - `\t` 24-hour time.
  - `\d` Date.

1.15 Reexecuting Commands with a Builtin Editor

- To reexecute a command, use either the builtin editor (vi or emacs) as specified in your `.inputrc` file.
- `.inputrc` can contain e.g. `set editing-mode vi`
- Using the editor is very convenient since you can change the command if it didn't work as expected. Simply hit ESC (for vi).
- This is a convenient way to experiment with new commands.

1.16 Reexecuting Commands with an Exclamation

Commands available include: - `!!` Reexecute most recent command. - `!n` Reexecute command number n. - `!-n` Reexecute the nth preceding command. - `!string` Redo the most recent command starting with string. - `!?string` Redo the most recent command containing string. - The last word on the previous command can be refered to as `!$`

Check also `history`

[15]:
```
ls
ls f1
ls -al f1
!!
!-2
```

```
EDAF35 Lecture 4.ipynb  ls-in-home              svib
a.c                     svi
f1                      svia
f1
-rw-r--r--  1 flagr  staff  46 Jan 20 13:58 f1
ls -al f1
-rw-r--r--  1 flagr  staff  46 Jan 20 13:58 f1
ls -al f1
-rw-r--r--  1 flagr  staff  46 Jan 20 13:58 f1
```

[16]:
```
ls -al ls-in-home
cat !$
```

```
-rw-r--r--  1 flagr  staff  254 Jan 20 13:59 ls-in-home
cat ls-in-home
Applications
Box Sync
Desktop
Documents
Downloads
Dropbox
Library
Movies
Music
Pictures
Privat
Public
Qt
SimplicityStudio
Sites
Terminal Saved Output
Zotero
bin
exjobb2017_v2.csv
gcviewer.properties
git
go
moss
node_modules
package-lock.json
```

```
target
temp
```

[17]:
```
history
```

```
  63  ./a.out ls
  64  ./a.out ls ls
  65  ./a.out ls .
  66  ./a.out "ls ."
  67  ./a.out "ls ."
  68  pico fastest.c
  69  gcc fastest.c
  70  ./a.out ls
  71  ./a.out ls .
  72  ./a.out "ls ."
  73  ./a.out "ls .."
  74  ./a.out "ls .. ."
  75  ./a.out "ls .." "ls ."
  76  ./a.out "ls .." "ls ." "echo Haha"
  77  ./a.out "ls .." "ls ." "echo Haha"
  78  ./a.out "ls .." "ls ." "echo Haha"
  79  ./a.out "ls .." "ls ." "echo Haha" echo echo
  80  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  81  pico fastest.c
  82  gcc fastest.c
  83  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  84  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  85  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  86  pic fastest.c
  87  pico fastest.c
  88  gcc fastest.c
  89  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  90  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  91  pico fastest.c
  92  gcc fastest.c
  93  pico fastest.c
  94  gcc fastest.c
  95  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
  96  ./a.out "sleep 10" "sleep 2"
  97  ./a.out "sleep 10" "sleep 1"
  98  man sleep
  99  pico fastest.c
 100  gcc fastest.c
 101  ./a.out "sleep 10" "sleep 1"
 102  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
 103  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
 104  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
 105  ./a.out "ls .." "ls ." "echo Haha" echo echo "sleep 5"
```

```
  534  echo $?
  535  echo $b
  536  echo $?
  537  x="once upon" y="a time" bash -c 'echo $x $y'
  538  echo $?
  539  echo $x
  540  echo $?
  541  b=bumble
  542  echo $b
  543  echo ${b}bee
  544  echo $bbee
  545  echo $?
  546  echo ${a-something}
  547  echo ${d-nothing}
  548  echo $d
  549  echo ${e=everything}
  550  echo $e
  551  echo ${d?Variable d not defined}
  552  echo $?
  553  ls
  554  ls f1
  555  ls -al f1
  556  ls -al f1
  557  ls -al f1
  558  echo $?
  559  ls -al ls-in-home
  560  cat ls-in-home
  561  echo $?
  562  history
```

[20]:
```
!542
```

```
echo $b
bumble
```

1.17 Quotation Marks

- There are three kinds of quotation marks:
  - in a string enclosed by ": variables are expanded.
  - in a string enclosed by ’: variables are not expanded.
  - the value of ‘`string`‘ is the stdout from executing string as a command and removing each trailing newline character:

  `$ rm ‘du -ks * | sort -n | awk ’ { print $2 } ’‘ # remove big file/directory`

  Note: the last form (back single quote) is equivalent to `$(command)`.

[21]:
```
du -ks * | sort -n | awk '{ print $2 }'
```

```
a.c
```

```
f1
ls-in-home
svi
svia
svib
EDAF35
```

[22]:
```
echo $(du -ks * | sort -n | awk '{ print $2 }')
```

```
a.c f1 ls-in-home svi svia svib EDAF35
```

[23]:
```
echo `du -ks * | sort -n | awk '{ print $2 }'`
```

```
a.c f1 ls-in-home svi svia svib EDAF35
```

1.18 Here Documents

- Sometimes it can be useful to provide the input to a script in the script file. The input is right "here".

```
$ cat phone
grep "$*" <<End
Office 046 222 9484
Mobile 0767 888 124
$X
End
```

- Above script contains both the command and the input.
- The variable X is expanded; suppress this behaviour by preceding End with a backslash on first line.

[24]:
```
variable=$(cat <<SETVAR
This variable
runs over multiple lines.
SETVAR
)

echo "$variable"
```

```
This variable
runs over multiple lines.
```

1.18.1 broadcast: Sends message to everyone logged in

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
E-mail your noontime orders for pizza to the system administrator.
    (Add an extra dollar for anchovy or mushroom topping.)
# Additional message text goes here.
```

```
# Note: 'wall' prints comment lines.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
#         wall <message-file
#  However, embedding the message template in a script
#+ is a quick-and-dirty one-off solution.

exit
```

more about here documents

1.19 Functions

```
function fun()
{
echo $1 # echo first argument
echo $2 # echo second argument
}
```

- The keyword `function` is optional.
- A function must be declared before it can be used.
- A function can be used as if it was any other UNIX command, i.e. no parentheses when the function is called (not even for passing arguments).

[25]:
```
function fun()
{
echo $1 # echo first argument
echo $2 # echo second argument
echo $0
}

fun ha hi
fun he ho hu
fun hiii
```

```
ha
hi
/bin/bash
he
ho
/bin/bash
hiii

/bin/bash
```

1.20 Simple Shell Syntax

- `a && b` executes `b` only if `a` succeeds (ie returns 0).
- `a || b` executes `b` only if `a` fails (ie returns nonzero).

The following commands can cause unhappiness if you run out of disk space during tar:

```
$ tar cf dir.tar dir; rm -rf dir; bzip2 -9v dir.tar
```

This is better:

```
$ tar cf dir.tar dir && rm -rf dir && bzip2 -9v dir.tar
```

Edit-compile-run without leaving the keyboard:

```
vi a.c && gcc a.c && a.out
```

But it is better to remap e.g. v, V, or t in `vi` to run `make`

1.21 For Loops

Iterate through certain files in your the current directory:

```
for x in *.c
do
    lpr $x # prints them
done
```

or through all argumets passed to a script:

```
for x in $*
do
    lpr $x
done
```

[26]:
```
for x in *
do
    echo $x
done
```

```
EDAF35 Lecture 4.ipynb
a.c
f1
ls-in-home
svi
svia
svib
```

You can also iterate through a string:

[27]:
```
a="x y z v"
for s in $a
do
    echo $s
done
```

```
x
y
```

```
z
v
```

Or simply a list:

[28]:
```
for s in a b c b
do
    echo $s
done
```

```
a
b
c
b
```

1.22 While and Until

```
while command
do
    body # do body while command returns true
done

until command
do
    body # do body while command returns false
done
```

1.23 If-Then-Else-Fi

```
if command
then
    then-commands
[else
    else-commands]
fi

if ! command
then
    then-commands
[else
    else-commands]
fi
```

1.24 Case

```
case word in
pattern1) commands;;
pattern2) commands;;
*) commands;;
```

```
esac
```

- Nothing happens if no pattern matches: putting `*)` last makes a default.

1.24.1 Longer example:

This is an excerpt of the script that starts Anacron, a daemon that runs commands periodically with a frequency specified in days.

```
case "$1" in
        start)
            start
            ;;

        stop)
            stop
            ;;

        status)
            status anacron
            ;;
        restart)
            stop
            start
            ;;
        condrestart)
            if test "x`pidof anacron`" != x; then
                stop
                start
            fi
            ;;

        *)
            echo $"Usage: $0 {start|stop|restart|condrestart|status}"
            exit 1

esac
```

1.25 cmp, diff, and ndiff

- `cmp` reports whether two files are equal.
- `diff` does the same but also shows how they differ.
- `ndiff` is a variant for which one can specify numerical differences which should be ignored.
  - `ndiff` is not standard but easy to find.

1.26 cut

- `cut` cuts out characters from each line of stdin
- `ls -l | cut -c2-10` prints the rwx-flags of the files.
- The first character on a line is c1.

- Multiple ranges can be specified: `ls -l | cut -c2-10 -c51-55` also prints five characters from the file name.

[29]:
```
ls -l | cut -c2-10
ls -l
```

```
otal 624
rw-r--r--
rw-r--r--
rw-r--r--
rw-r--r--
rwxr-xr-x
rwxr-xr-x
rwxr-xr-x
total 624
-rw-r--r--@ 1 flagr  staff  293310 Jan 20 14:03 EDAF35 Lecture 4.ipynb
-rw-r--r--  1 flagr  staff      14 Mar 26  2018 a.c
-rw-r--r--  1 flagr  staff      46 Jan 20 13:58 f1
-rw-r--r--  1 flagr  staff     254 Jan 20 13:59 ls-in-home
-rwxr-xr-x  1 flagr  staff      93 Mar 26  2018 svi
-rwxr-xr-x  1 flagr  staff      81 Mar 26  2018 svia
-rwxr-xr-x  1 flagr  staff      93 Mar 26  2018 svib
```

[30]:
```
ls -l | cut -c2-10 -c51-55
```

```
otal 624
rw-r--r--AF35
rw-r--r--c
rw-r--r--
rw-r--r---in-h
rwxr-xr-xi
rwxr-xr-xia
rwxr-xr-xib
```

1.27 find

Example: `find . -name '*.c'` The output will be a list of files (with full path) with suffix c.

We can feed that list to wc using: `wc ‘find . -name '*.java'‘` The default action is to print the file name.

A number of criteria can be specified, including - `-anewer filename` selects files newer than filename. - `-type type` selects files of type type which is one of b,c,d,f,l, p, or s (with the same meaning as printed by `ls -l`: block special file (eg disk), character special file (eg usb port), directory, ordinary file, symbolic link, name pipe, or socket).

[31]:
```
find . -name '*.ipynb'
find . -name '*.c'
```

```
./EDAF35 Lecture 4.ipynb
./.ipynb_checkpoints/EDAF35 Lecture 3-checkpoint.ipynb
./.ipynb_checkpoints/EDAF35 Lecture 4-checkpoint.ipynb
./a.c
```

1.28 cleanfiles

```
find . -name *.tac.??? -exec rm '{}' \;
find . -name *.pr -exec rm '{}' \;
find . -name cmd.gdb -exec rm '{}' \;
find . -name *.ps -exec rm '{}' \;
find . -name *.dot -exec rm '{}' \;
find . -name *.aux -exec rm '{}' \;
find . -name *.o -exec rm '{}' \;
find . -name out -exec rm '{}' \;
find . -name x -exec rm '{}' \;
find . -name y -exec rm '{}' \;
find . -name a.out -exec rm '{}' \;
find . -name cachegrind.out.* -exec rm '{}' \;
```

Have a look at `man find`

1.29 awk

- Stands for Aho (from the Dragonbook), Weinberger (from hashpjw in the Dragonbook), and Kernighan (K in K&R C).
- Each line of input is separated into fields and are denoted `$1,$2,...` Assume a variable is called `X` and has value `2`. Then `$X` refers to the second field.
- The entire line is `$0`, number of fields on a line is denoted `NF`, and line number is `NR`.
- Each line in an `awk` program has *a pattern* and *an action*. If a line in the input matches the pattern, the action is executed.

1.30 Example awk programs

```
$ awk '{ print $1, $5; }' # print first and fifth item.
$ awk '$1 > 10 { print $1, $2; }' # print first two items if $1 is > 10.
$ awk 'NR == 10' # print tenth line.
$ awk 'NF > 4' # print each line with > 4 fields.
$ awk 'NF > 0 ' # print each nonempty line.
$ awk '$NF > 4 ' # print each line with last field > 4.
$ awk '/abc/ ' # print each line containing abc.
$ awk '/abc/ { n = n + 1; }\
  END { print n;}' # print number of lines containing abc.
$ awk 'length($0) > 80' # print each line longer than 80 bytes.
```

The `END` pattern matches at `EOF`. There is also a `BEGIN` pattern which is matched before the first character is read.

[32]:
```
echo a b c d e | awk '{ print $1, $5; }'
```

```
a e
```

1.31 head and tail

- `head` prints the first 10 lines of a file (or stdin).
- `head -100` prints the first 100 lines of a file (or stdin).
- `tail` prints the last 10 lines of a file (or stdin).
- `tail -100` prints the last 100 lines of a file (or stdin).
- `tail -f file` like normal tail but at EOF waits for more data.

1.32 od

- Octal dump
- `od file` dumps the file contents on stdout in as octal numbers.
- `od -c file` prints file as characters.
- `od -x file` prints file as hex numbers.

1.33 sed

- stream editor.
- It can be useful for e.g. changing prefixes in a `Yacc` generated parser:

```
sed 's/yydebug/pp_debug/g' y.tab.c > tmp; mv tmp y.tab.c
```

[33]:
```
echo a b c d aa | sed 's/a/Hahahah/g'
```

```
Hahahah b c d HahahahHahahah
```

1.34 grep

- Grep searches for a pattern in files.
- GNU `grep` has the useful `-r` option which traverses directories.
- In *basic regular expressions* ?, +, braces, parentheses and bar (i.e. |) have no special meaning. Backslash them to get that.
- In *extended regular expressions*, enabled with `-E`, above characters are special. More about that on next slide.

```
$ grep abc # matches line with abc.
$ grep -e '[abc]' # matches line with any of a, b, or c.
$ grep -e '[^abc]' # matches line with none of a, b, or c.
$ grep -e '[^ab-d]' # matches line with none of a, b, c, or d.
$ grep ab*c # matches line with ac, abc, abbbbbc.
```

[34]:
```
grep abc EDAF*
```

```
    "$ awk '/abc/ ' # print each line containing abc.\n",
    "$ awk '/abc/ { n = n + 1; }\\\n",
    "  END { print n;}' # print number of lines containing abc.\n",
    "$ grep abc # matches line with abc.\n",
    "$ grep -e '[abc]' # matches line with any of a, b, or c.\n",
```

drrPiOPw9bT2egx65qUMSXmqJotpdXNtpa3ssosILiaK22JK618TPE4ipRoYepXq1MPhnVeHoTqTlSoO
vKMqzo023Gm6soxlU5EudxTldo+mhhsPTrV8TToUoYjEqksRWhTjGrXVCMo0VVqL3qipRlKNPmfuKTSt
c36xNgoAKAPMfiP8E/g58Yv7C/4W18KPhx8T/+EXu5r7w3/wALA8E+G/GH9g3dybY3U+kf8JBpuof2dJ
dmzs/tZtBGLr7Ja/aBJ9nh2ehgM2zTK/bf2bmWPy/6xFQr/UsXXwvtox5uWNX2FSHOo80uXmvy80uW3N
I4MdlWV5n7H+0suwOYfV5OVD67hKGK9jKXLzO17aE/ZuXLHm5bc3LHmvZI9MRFjVURVREUIiIAqoqjCq
qjAVVAAAAwAMDHFee3fV6t6tvqd6VtFolokuhyt54D8Eah4x0f4h3/AIQ8M3vj3w9pOoaDoHjS80PTbn
xToeiatIkuq6To+vTWz6ppmn6nJGjahaWVzBBelE+0pKEUV0xxmLhhhauBhisRDBV6sK1bCRrVI4atVpq
1OrVoJqnUqU1pCc1KUfs2OeWEwk8TSxs8NQnjKNOdGjipUacsRSpVHepTpVnF1KcKjXvxhKKl1vax1lc
x0BQAUAFABQAUAFAGBoXhTwx4X18QT+G/D2i6DP4r8QXPivxPNo+mWenS+IfE97Zafpt34h1p7SGFtT1
q50/SdMsZ9TvDPeS2mnWNu8zQ2sCLvWx0IxCoqvXq1lhqMcNh1VqSmqGHh0c40KSk2qdKM6tScacLRUp
zlZuTZjRw+Hw7rSoUKVF4mtLEYh0qcY0viJQhTlXquKTqVZQp040p08nGEI3ailHfrA2CgAoAKACgAoA
KACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKAC
gAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoAKACgAoA/9k=\"
\n"


```
      "    \"$ awk '/abc/ ' # print each line containing abc.\\n\",\n",
      "    \"$ awk '/abc/ { n = n + 1; }\\\\\\n\",\n",
      "    \"  END { print n;}' # print number of lines containing
abc.\\n\",\n",
      "    \"$ grep abc # matches line with abc.\\n\",\n",
      "    \"$ grep -e '[abc]' # matches line with any of a, b, or c.\\n\",\n",
      "    \"$ grep -e '[^abc]' # matches line with none of a, b, or
c.\\n\",\n",
      "    \"$ grep ab*c # matches line with ac, abc, abbbbbc.\\n\",\n",
      "    \"grep abc EDAF*\"\n"
    "grep abc EDAF*"
```

[35]:
```
cat f1
echo ----------------------
grep -e '[^leoH]' f1
```

```
Hello
 world!
 world!
 world!
 world!
 world!
----------------------
 world!
 world!
 world!
 world!
 world!
```

[37]:
```
grep -e '[^leo]' f1
```

```
Hello
 world!
 world!
```

```
 world!
 world!
 world!
```

1.35 grep -E

```
$ grep -E -e 'a|b' # matches line with a or b.
$ grep -E -e 'a|bc' # matches line with a or bc.
$ grep -E -e '(a|b)c' # matches line with a or b, followed by c.
$ grep -E -e '(a|b)?c' # ? denotes optional item.
$ grep -E -e '(a|b)+c' # + denotes at least once.
$ grep -E -e '(a|b)*c' # + denotes zero or more.
$ grep -E -e '(a|b){4}c' # {4} matches pattern four times.
```

- Without -E use backslash before above metacharacters.
- Without ' the shell will try to setup a *pipe* ... |

1.36 sort and uniq

- `sort file` sorts a file alphabetically.
- `sort -n file` sorts a file numerically.
- `uniq` removes duplicates line if found in sequence

[38]:
```
sort f1
```

```
 world!
 world!
 world!
 world!
 world!
Hello
```

[39]:
```
uniq f1
```

```
Hello
 world!
```

[40]:
```
cat svi
```

```
#!/bin/bash
vi -c /$1 `egrep -e $1 *.[ch] */*.[ych] | awk -F: '{ print $1; }' | uniq | sort`
```

1.36.1 What does the above script do?

[ ]:
```

```