Bash Crash course + **`bc`** + **`sed`** + **`awk`**[*]

Andrey Lukyanenko, CSE, Aalto University

Fall, 2011

There are many Unix shell programs: `bash`, `sh`, `csh`, `tcsh`, `ksh`, etc. The comparison of those can be found on-line [1]. We will primary focus on the capabilities of `bash v.4` shell[2].

1. Each bash script can be considered as a text file which starts with **`#!/bin/bash`**. It informs the editor or interpretor which tries to open the file, what to do with the file and how should it be treated. The special character set in the beginning **`#!`** is a magic number; check **`man magic`** and `/usr/share/file/magic` on existing magic numbers if interested.

2. Each script (assume you created “**`scriptname.sh`** file) can be invoked by command **`<dir>/scriptname.sh`** in console, where **`<dir>`** is absolute or relative path to the script directory, e.g., **`./scriptname.sh`** for current directory. If it has **`#!`** as the first line it will be invoked by this command, otherwise it can be called by command **`bash <dir>/scriptname.sh`**.

   Notice: to call script as **`./scriptname.sh`** it has to be executable, i.e., call command **`chmod 555 scriptname.sh`** beforehand.

3. Variable in **`bash`** can be treated as integers or strings, depending on their value. There are set of operations and rules available for them. For example:

```
#!/bin/bash
var1=123           # Assigns value 123 to var1
echo var1          # Prints ’var1’ to output
echo $var1         # Prints ’123’ to output

var2 =321          # Error (var2: command not found)
var2= 321          # Error (321: command not found)
var2=321           # Correct

var3=$var2         # Assigns value 321 from var2 to var3

echo $var3         # Prints ’321’ to output
echo ${var3}       # Prints ’321’ to output
echo "$var3"       # Prints ’321’ to output
echo "${var3}      # Prints ’321’ to output

foo="A B  C   D"
echo $foo          # Prints ’A B C D’ to output
echo "$foo"        # Prints ’A B  C   D’ to output
echo ’$foo’        # Prints ’$foo’ to output

foo=               # Setting $foo to NULL value
echo $foo          # Prints empty line
unset var1         # Sets var1 to NULL (unsets var1)

var1=world!         # Assigns string ’world!’ to var1
echo "Hello, $var1" # Prints ’Hello, world!’ to output

read var1           # Reads string from input and assigns it to var1

a=$(ls -l)          # Assigns to ’a’ command ’ls -l’ invocation result
echo $a             # Unquoted, removes whitespaces.
echo
echo "$a"           # The quoted variable preserves whitespace.

a=‘ls -l‘           # Older version, the same as ’a=$(ls -l)’

let b=5+7           # Sets arithmetic operation on integers
echo $b             # Prints ’12’ to output
b=5+7               # String assignments
```

[*] This is squeezed document, compiles partly and aggregates information from open documents. For more information read those documents.

[1] For example, in `http://www.faqs.org/faqs/unix-faq/shell/shell-differences/` or `http://www.hep.phy.cam.ac.uk/lhcb/LHCbSoftTraining/documents/ShellChoice.pdf`

[2] A detailed information on bash scripting can be found at `http://tldp.org/LDP/abs/html/` for example

```
echo $b             # Prints '5+7' to output
let c=b-2           # Arithmetic operation
echo $c             # Prints '10' to output
let d=B+5           # String in arithmetic interpreted as 0
echo $d             # Prints '5' to output
```

Variables can be global, local, environmental, constants, arrays (for details...).

4. Test operators.

- **test operator**

  test opts args or,

  equivalently:
  [opts args] or,

  even preferably:
  [[opts args]]

  Example:

  ```
  # File tests
  test -e file          # Checks: file exists
  test -s file          # Checks: file non-zero size
  test -d file          # Checks: is directory file
  test -f file          # Checks: file is regular (not device file)
  test -b file          # Checks: file is block device
                        # ... same 'c' character device, 'p' - pipe,
                        # 'h' or 'L' - symbolic link, 'S' -socket
  test -r test.txt      # Checks: file has read permission
                        # 'w' for write, 'x' for execute
  test -O test.txt      # Checks: are you owner of the file?
  test -N text.txt      # Checks: File was modified since last read
  test file1 -nt file2  # Checks: file1 is newer than file2
  test file1 -ot file2  # Checks: file1 is older than file2

  # String tests
  test -z str1          # Checks: is the length of string str1 empty
  test str1             # Checks: is string str1 non-zero length
  test str1 = str2      # Checks: is string str1 equals str2
  test str1 != str2     # Checks: is string str1 non-equal str2

  # Numerical tests (numeric numbers or string length e.g. '-l str1')
  test num1 -eq num2    # Checks: num1=num2
  test num1 -ne num2    # Checks: num1!=num2
  test num1 -lt num2    # Checks: num1<num2
  test num1 -le num2    # Checks: num1<=num2
  test num1 -gt num2    # Checks: num1>num2
  test num1 -ge num2    # Checks: num1>=num2
  ```

- **if/then**,

  ```
  if [ condition-true ]
  then
    ... # true
  else
    ... # false
  fi
  ```

  Example:

  ```
  if [ -z "$1" ] # Identical to 'test -z "$1"'
  then
    echo "No command-line arguments."
  else
    echo "First command-line argument is $1."
  fi
  ```

- **if/then/elif**

  ```
  if [ condition1 ]
  then
    ...  # condition1 is true
  elif[ condition2 ] # Same as else if
  then
    ...  # condition2 is true,
         # but condition1 was false
  else
    ...  # both condition1 and
         # condition2 are false
  fi
  ```

  Example:

  ```
  if [ -z "$1" ] # Identical to 'test -z "$1"'
  then
    echo "No command-line arguments."
  elif [ -e "$1" ] # Whether file with arg name exists
  then
    echo "File $1 exists locally."
    echo "First command-line argument is $1."
  else
    echo "File $1 does not exists locally."
    echo "First command-line argument is $1."
  fi
  ```

- **case (in) / esac**

```
case "$variable" in
  "$condition1" )
    ...
  ;;

  "$condition2" )
    ...
  ;;
esac
```

Example:

```
echo; echo "Hit a key, then hit return."
read Keypress # read from console a key

case "$Keypress" in  # for this key
  [[:lower:]]   ) echo "Lowercase letter";; # same as [a-z]
  [[:upper:]]   ) echo "Uppercase letter";; # same as [A-Z]
  [0-9]         ) echo "Digit";;
  *             ) echo "Punctuation, whitespace, or other";;
esac      # :lower:, :upper: works with different locales
```

- **`select`**

```
select variable [in list]
do
  ...
  break
done
```

Example:

```
PS3='Choose your favorite vegetable: ' # Sets the prompt string.
                                       # Otherwise it defaults to #? .

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break  # What happens if there is no 'break' here?
done
```

5. Loop operators

- **`for`** loops

```
for arg in [list]
do
  ...
done
```

Example:

```
MFILES=. # current directory

for file in $MFILES/*  # for all files in directory
                       # '*' is wild card as well as '?'
do
  echo -n "$file "    # prints files, '-n' for no newline
                       # everything goes in one line
done
echo                   # final end line
```

- **`while`** loops

```
while [condition]
do
  ...
done
```

Example:

```
var=0
N=10

while [ "$var" -lt "$N" ] # [ $var -lt $N ] also works
                          # [ $var < $N ] does not work
do
  echo -n "$var "         # print all in a line

  var=$((var+1))          # Arithmetic increment
done
echo
```

- **`until`** loops

```
until [not condition]
do
  ...
done
```

Example:

```
END_COND=end # type 'end' to exit
i=1

until [ "$var1" = "$END_COND" ]  # until end typed
do
  echo "$i. Input variable #1 "
  echo "$i. (Type '$END_COND' to stop)"
  read var1
  echo "$i. variable #1 = $var1"
  i=$((i+1))
  echo
done
```

6. Escaping characters.

`\n`	End of line symbol (newline). E.g., `echo "hello\n\n"`
`\r`	Return carriage (beginning of the current line).
`\t`	Horizontal tabular (whitespace horizontally).
`\v`	Vertical tabular (whitespace vertically).
`\b`	Backspace.
`\0xx`	Translates to the octal ASCII equivalent of 0*nn*, where *nn* is a string of digits. E.g., `quote=$'\042'`

More examples:

Example:

```
echo "This will print
      as two lines."

echo "This will print \
      as one line."

echo "\v\v\v\v"     # Prints '\v\v\v\v'
echo -e "\v\v\v\v"  # Makes vertical spacing

echo "Hello"        # Hello
echo "\"Hello\"     # "Hello"

echo "\$var1"           # $var1
echo "The book cost \$7.98."  # The book cost $7.98.
```

7. Internal variables[3]

- Builtin Variables.

`$EUID`	"Effective" user ID number.
`$UID`	User ID number (type "id" in console).
`$GROUPS`	Groups current user belongs to. This is a listing (array) of the group id numbers for current user, as recorded in `/etc/passwd` and `/etc/group`.
`$HOME`	Home directory of the user, usually `/home/username`.
`$IFS`	Internal field separator (by default whitespaces). Shows how to separate words.
`$LINENO`	Current line number of the script. Type `echo $LINENO` in console several times.
`$OLDPWD`	Old (previous) working directory.
`$PWD`	Current working directory.
`$PPID`	`Process id` of a parental process. For current process use `$$`.
`$SECONDS`	The number of seconds the script has been running.
`$REPLY`	The default value when a variable is not supplied to read. E.g., '`read`' is called instead of '`read key`'.
`$SHELL`	Which shell is currently running.
`$TERM`	Which terminal is currently working.
`$LOGNAME`	Name of logged in user.

- Positional Parameters.

`$1, $2,...`	Positional parameters, passed from command line to script. E.g., `script.sh par1 par2`, `$1=par1`, `$1=par2`,.
`$#`	Number of command-line arguments. E.g., `$#=2.` for call `script.sh par1 par2`.
`$*`	All of the positional parameters, seen as a single word. E.g., `$*=par1 par2`.
`$@`	Same as `$*`, but each parameter is a quoted string.

- Other Parameters.

`$-`	Flags passed to script.
`$!`	`PID` (process `ID`) of last job run in background.
`$_`	Special variable set to final argument of previous command executed.
`$?`	Exit status of a command, function, or the script itself.
`$$`	Process `ID` (`PID`) of the script itself.

[3] For almost full list see `http://tldp.org/LDP/abs/html/internalvariables.html`.

8. Parameter Substitution[4].

Operations	**Description**	**Examples**
`${var}`	Same as $var, substitutes the variable name ($var) with its value.	`echo "${0}"`
`${var:-def}`	The same as **`${var}`** except that if $var is unset use def value given as second parameter.	`t=${none:-123}`

`${var:=def}`	The same as **`${var}`** except that if $var is unset assign it to second parameter.	`${none:=123}`
		`echo none`

`${var:+alt_value}`	If **`${var}`** set then returns alt value.	`var1=321`
		`echo ${var1:+123}`

`${var:?err_msg}`	If **`${var}`** set then returns it, otherwise err_msg returned.	`echo ${none:?err}`
`${#var}`	String length (number of characters in `$var`).	`echo "${#0}"`
`${var#pattern}`	Removes from $var the shortest part of $pattern that matches the front end of $var.	`echo "${0#*/}"`
`${var##pattern}`	Removes from $var the longest part of $pattern that matches the front end of $var.	`echo "${0##*/}"`
`${var%pattern}`	Removes from $var the shortest part of $pattern that matches the back end of $var.	`echo "${0%b*}"`
`${var%%pattern}`	Removes from $var the longest part of $pattern that matches the back end of $var.	`echo "${0%%b*}"`
`${#*}` and **`${#@}`**	Gives the number of positional parameters. For an array, `${#array[*]}` and `${#array[@]}` give the number of elements in the array.	
`${var:pos}`	Returns substring of $var starting from `pos`	`echo "${0:3}"`
`${var:pos:len}`	Returns substring of $var starting from `pos` of length `len`	`echo "${0:3:4}"`
`${var/pttrn/subst}`	Substitute *first* occur of `pttrn` in `$var` with `subst`.	`echo "${0/b/123}"`
`${var//pttrn/subst}`	Substitute *all* occur of `pttrn` in `$var` with `subst`.	`echo "${0//b/123}"`
`${var/#pttrn/subst}`	If `$var` prefix matches `pttrn` replace it with `subst`.	`echo "${0/#\/b/123}"`
`${var/%pttrn/subst}`	If `$var` sufix matches `pttrn` replace it with `subst`.	`echo "${0/%h/123}"`
`${!prefix*}`	Returns all declared variables which starts with `prefix`	`echo "${!D*}"`
`${!prefix@}`	Returns all declared variables which starts with `prefix`	`echo "${!D@}"`

9. Random numbers.

   It is possible to use the following techniques:

   - `$RANDOM`

```
# Password generator
ALPHABET="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
while [ "${n:=1}" -le "8" ]
do
   PASS="$PASS${ALPHABET:$(($RANDOM%${#ALPHABET})):1}"
   let n+=1
done
echo "$PASS"
```

     Problem: Every run in the same bash environment returns the same number.

     How to explicitly reseed? External tool is required or give seeding number directly: `RANDOM=10` reseeds.

   - Use external program such as **`awk`**.

```
AWKSCRIPT=' { srand(); print rand() } ' # awk script
echo | awk "$AWKSCRIPT"  # echo gives to awk empty file on input using pipes
```

   - Use linux random number generator device **`/dev/urandom`** (the best option, the most "pseudorandom").

```
head -c4 /dev/urandom| od -An -tu4 # takes first 4 bytes from urandom device
                                   # and parse it (od) as numbers (option u4), can be symbols with "-ta"
```

[4] http://tldp.org/LDP/abs/html/parameter-substitution.html

10. Functions

```
# 1st definition
function function_name {
  ...
}

# 2nd definition
function_name () {
  ...
}
```

Example:

```
fun1 () { echo "This is a function"; echo; }  # simple function

echo $1                                 # prints positional parameter
fun2 () { echo "Function takes $1" }    # prints functional arguments
```

11. Regular expression [5].

Operation	Definition	Example
char	A single ordinary character matches itself.	
string	A string of chars matches itself.	
*****	Matches for the preceding regular expression repeated 0 or more times.	
+	As *, but matches one or more.	
?	As *, but only matches zero or one.	
{i}	As *, but matches exactly i times.	
{i,j}	As *, but matches between i and j, inclusive, times.	
{i,}	As *, but matches more than or equal to i sequences.	
(regexp)	Groups the inner regexp as a whole.	
.	Matches any character, including newline.	
^	Looks for matching only at the beginning of strings.	
$	It is the same as ^, but refers to end of pattern space.	
[list]	Matches any single character in list.	`x[12]` matches `x1` or `x2`.
-	A list may include sequences like char1-char2, which matches any character between char1 and char2.	
[^list]	Reverses the meaning of list, so that it matches any single character not in list.	
exp1|exp2	Matches either exp1 or exp2.	
exp1exp2	Matches the concatenation of exp1 and exp2.	
[:digit:]	Digits. (abbr: **\d**, reverse **\D**)	
[:alnum:]	Any alphanumeric character (abbr, **\w**, reverse **\W**).	
[:alpha:]	Any alpha character A to Z or a to z.	
[:blank:]	Space and TAB characters only.	
[:xdigit:]	Hexadecimal notation 0-9, A-F, a-f.	
[:punct:]	Punctuation symbols.	
[:print:]	Any printable character.	
[:space:]	Any whitespace characters (space, tab, NL, FF, VT, CR) (abbr **\s**, reverse **\S**).	
[:graph:]	Same as [:print:], excluding whitespaces (SPACE, TAB).	
[:upper:]	Any alpha character A to Z.	
[:lower:]	Any alpha character a to z.	
[:cntrl:]	Control Characters.	
\b	Boundary of a word.	
\B	Non-boundary of a word.	

Examples:

```
'abcdef'  Matches 'abcdef'.
'a*b'     Matches zero or more 'a's followed by a single 'b'. For example, 'b' or 'aaaaab'.
'a\?b'    Matches 'b' or 'ab'.
'a\+b\+'  Matches one or more 'a's followed by one or more 'b's: 'ab' is the shortest possible match, but other examples are 'aaaab'
'.*'
'.\+'     These two both match all the characters in a string; however, the first matches every string (including the empty string),
'^main.*(.*)' This matches a string starting with 'main', followed by an opening and closing parenthesis. The 'n', '(' and ')' need
'^#'      This matches a string beginning with '#'.
'\\$'     This matches a string ending with a single backslash. The regexp contains two backslashes for escaping.
'\$'      Instead, this matches a string consisting of a single dollar sign, because it is escaped.
'[a-zA-Z0-9]' In the C locale, this matches any ASCII letters or digits.
```

[5] http://www.grymoire.com/Unix/Regular.html, http://www.gnu.org/software/sed/manual/sed.html#Regular-Expressions, http://www.zytrax.com/tech/web/regex.htm and http://tldp.org/LDP/abs/html/x16947.html

```
'[^ tab]\+'   (Here tab stands for a single tab character.) This matches a string of one or more characters, none of which is a spac
'^\(.*\)\n\1$' This matches a string consisting of two equal substrings separated by a newline.
'.\{9\}A$' This matches nine characters followed by an 'A'.
'^.\{15\}A' This matches the start of a string that contains 16 characters, the last of which is an 'A'.
```

Notice: Yet uncovered topics for home reading: **`aliases`**, **`declare`**, **`source`** (dot command), **`shopt`**, **`getopts`**, **`hash`**, **`arrays`**, **`lists`**, **`piping`**, **`eval`**, **`debugging`**, **`set`** (options), **`regex`** (regular expressions), **`local`** variable.

Check out arcanoid written in bash arcanoid.sh is available on course homepage the original is taken from[6].

[6] http://bolknote.ru/2011/09/18/~3410

Additional very useful programs, which extend functionality of bash scripts.

1. `bc`[7]

Basic `bc` operands are variables (`var`) and expressions (`expr`).

`- expr`	Negation of the expression.
`++ var`	Pre-increment by one.
`-- var`	Pre-decrement by one.
`var ++`	Post-increment by one.
`var --`	Post-decrement by one.
`expr + expr`	Summing up expressions.
`expr - expr`	Subtracting expressions.
`expr * expr`	Multiplying expressions.
`expr / expr`	Dividing expressions (affected by scale).
`expr % expr`	Remainder of expressions divisions (affected by scale, only if scale is 0 then integer reminder).
`expr1 êxpr2`	Computing power of `expr1` with exponent `expr2`. `expr2` should be integer.
`( expr )`	Force computation of expression before usage.
`var = expr`	Assignment operation to variable.
`var = expr`	

Comparison and logical operations:

`expr1 < expr2`	Return 1 if `expr1` less than `expr2`, 0 otherwise.
`expr1 <= expr2`	Return 1 if `expr1` less or equal `expr2`, 0 otherwise.
`expr1 > expr2`	Return 1 if `expr1` greater `expr2`, 0 otherwise.
`expr1 >= expr2`	Return 1 if `expr1` greater or equal `expr2`, 0 otherwise.
`expr1 == expr2`	Return 1 if `expr1` equal to `expr2`, 0 otherwise.
`expr1 != expr2`	Return 1 if `expr1` not equal to `expr2`, 0 otherwise.
`!expr`	Return 1 if `expr` is 0, returns 0 otherwise.
`expr && expr`	Rerun 1 if both expressions are non-zero.
`expr || expr`	Return 1 if one of the expressions is non-zero.

Program `bc` also accepts **`if`**, **`for`**, **`read`** operations and so on [8].

Math operations should be used with parameter `-l`:

`sqrt(x)`	Returns square root (may be used without option `-l`).
`s(x)`	Sine of `x` radians.
`c(x)`	Cosine of `x` radians.
`a(x)`	Arctangent of `x`.
`l(x)`	Natural logarithm of `x`.
`e(x)`	Exponential function (e^x) of `x`.
`j(n,x)`	The bessel function of integer order `n` of `x`.

Standard usage as following:

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Examples:

```
dev=$(echo "scale=6; sqrt($dev_sqr)"|bc); # taking square root from dev_sqr
                                          # with 6 digit after the decimal separator precision.

pi=$(echo "scale=10; 4*a(1)" | bc -l)     # Computing pi constant, with 10 points


# Function definitions are also available in bc environment
cmmd="
scale=6;
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
};
f(5)" # define factorial function and compute it for 5

echo $(echo "$cmmd" | bc)
```

[7] See also `factor` and `dc`

[8] for details see `http://linux.about.com/od/commands/l/blcmdl1_bc.htm`

As it can be seen `bc` accepts variables inside, and most of arithmetical operations (see `man bc` for details).

2. sed[9]

sed has two data buffers. The first one is pattern buffer (for current match), the second one is hold buffer (for saving purposes). **sed** parses input stream line-by-line and for every line puts it to pattern buffer and performs requested operations based on matching.

The methods to select lines:

Most important and simplest examples:

```
sed s/expr1/expr2/ <infile >outfile # substitutes regular expression expr1 with string expr2
                                     # takes data stream from infile and outputs it outfile

sed s/expr1/expr2/ infile >outfile          # same
cat infile | sed s/expr1/expr2/ >outfile   # same with piping

# 's' - substitute command, '/'" works as delimiter (expr1 and expr2 should not contain "/",
# escaping is allowed '\/' ).

#Delimiter can be changed, e.g.,
sed s|expr1|expr2| <infile >outfile # the same as before, given expr1 and expr2 does not contain |
sed s*expr1*expr2* <infile >outfile # the same as before, given expr1 and expr2 does not contain *


sed 's/abc/(abc)/'      # adding parentheses around
sed 's/[a-z]*/(&)/'     # same for regular expression,
                        # whenever what is found unknown (regular expression) & can be used

echo "abc 123 abc" | sed 's/[0-9]\+/& &/' # returns abc 123 123 abc

sed 's/foo/bar/'              # replaces only 1st instance in a line
sed 's/foo/bar/4'             # replaces only 4th instance in a line
sed 's/foo/bar/g'             # replaces ALL instances in a line

sed '/baz/s/foo/bar/g'        # substitute "foo" with "bar" ONLY for lines which contain "baz"
sed '/baz/!s/foo/bar/g'       # substitute "foo" with "bar" EXCEPT for lines which contain "baz"

sed 10q                 # print first 10 lines of file (emulates behavior of "head")
sed q                   # print first line of file (emulates "head -1")

# two methods to print only lines which match regular expression (emulates "grep")
sed -n '/regexp/p'            # what happens if reverse print (p)? "!p" instead of "p"
sed '/regexp/!d'              # and here "d" instead of "!d"?

sed '/AAA/!d; /BBB/!d; /CCC/!d'  # grep for AAA and BBB and CCC (in any order)
sed '/AAA.*BBB.*CCC/!d'          # grep for AAA and BBB and CCC (in that order)

sed -n '/^.\{65\}/p'  # print only lines of 65 characters or longer. What is "-n" for?
sed -n '/^.\{65\}/!p   # print only lines of less than 65 characters

sed '1,10d' # delete the first 10 lines of a file
sed '$d'  # delete the last line of a file

sed -n '45,50p'             # print line nos. 45-50 of a file
gsed '1~5d'                # delete all 1+5*i lines: 1,6, 11, 16

sed 'y/abcdef/ABCDEF/'

sed -n '$='

# add after
sed '
/WORD/ a\
Add this line after every line with WORD
'

# change
sed '
/WORD/ {
i\
Add this line before
a\
Add this line after
```

[9] Many examples available at `http://www.pement.org/sed/sed1line.txt`, `www.gnu.org/software/sed/manual/sed.html` and detailed description at `http://www.grymoire.com/Unix/Sed.html`.

```
c\
Change the line to this one
}’

# insert before
sed ’
/WORD/ i\
Add this line before every line with WORD
’

# repeating when s is done
sed ’
:again
s/([ ^I]*)//g
t again
’



# just read it and understand
sed -n ’
’/$1/’ !{;H;x;s/^.*\n\(.*\n.*\)$/\1/;x;}
’/$1/’ {;H;n;H;x;p;a\
---
}’

# passing regular expressions
arg=‘echo "$1" | sed ’s:[]\[\^\$\.\*\/]:\\\\&:g’‘
sed ’s/’"$arg"’//g’
```

3. **awk** (based on GAWK version).[10]

- Running AWK scripts.

  As **sed**, program **awk** reads input stream (or file if given) line-by-line and performs some operations on per line basic. However, now **awk** has not only two buffers, but a whole set of programming language in order to process calculus on the input stream.

  ```
  awk ’program’ input-file1 input-file2 ...       # direct inline script processing
  awk -f program-file input-file1 input-file2 ..  # script written in file processing
  ```

- Structure of **awk** scripts.

  ```
  BEGIN { init_actions }
  pattern1 { actions1 }
  pattern2 { actions2 }
  ...
  patternN { actionsN }
  END { final_actions }
  ```

  awk has many C style constructions (if-else, for, while, break, continue, etc). However such commands as **next**, which stops current pattern study and force to continue with next, **nextfile**, **exit** are new.

- Built-in variables:

CONVFMT	String controls conversion of numbers to strings. Its default value is "%.6g". Not for converting, but for printing use OFMT.
FS	The input field separator. The default value is " ", a string consisting of a single space and tabs.
OFMT	String controls conversion of numbers to strings for printing with the print statement. Its default value is "%.6g". Same as printf("%.6g", 0.00001) for every print.
OFS	The output field separator. It is output between the fields printed by a print statement. Its default value is " ", a string consisting of a single space.
ORS	The output record separator. It is output at the end of every print statement. Its default value is "\n", the newline character.
RS	the input record separator. Its default value is a string containing a single newline character, which means that an input record consists of a single line of text. Can be regex.
SUBSEP	The multidimensional array separator.
ARGC ARGV	The command-line arguments available to awk programs are stored in an array called ARGV. ARGC is the number of command-line arguments present.
NF	The number of fields in the current input record. NF is set each time a new record is read, when a new field is created or when $0 changes.
NR	The number of input records awk has processed since the beginning of the program’s execution

The **awk** has the following built-in functions[11] we don’t study explicitly, but they are mainly trivial: atan2, cos, exp,

[10] GAWK http://www.gnu.org/s/gawk/manual/gawk.html

[11] http://www.gnu.org/s/gawk/manual/gawk.html#Functions

int, log, rand, sin, sqrt, srand, asort, asorti, gensub, gsub, index, length, match, patsplit, split, strtonum, sub, substr, tolower, toupper, system, close, fflush, systime, strftime, and, compl, lshift, or, rshift, xor.

- Study by examples:

```
# prints Hello, world to standard output
awk ’BEGIN { print "Hello, world!"; }’ # does not need input stream, one operation and exit

# reprints input to standard output
awk ’// { print $0 }’ input   # $0, $1, $2 are positional parameters:
                              # $0 corresponds to the whole line
                              # $1 corresponds to the first space separated word...

# prints lines that has more than 10 symbols
awk ’{ if (length($0) > 10) print $0;}’ input    # applied for every line, internally checks conditions
awk ’length($0) > 10’ input                      # applied to lines where condition holds, internal
                                                 # print is default action

# NF is the number of fields in current record
awk ’NF > 0’ input       # prints lines which has at least one record
awk ’{print $NF}’ input  # prints last word of each line

# awk can get variables from outside
awk ’{ print $n }’ n=4  input1 n=2 input2   # n=4 is outside of the script, n=2 after input1 is processed

# string operations on variables
awk ’BEGIN {two = 2; three = 3; print (two three) + 4}’ # result is 27, why?

# Precision
awk ’BEGIN {printf("%.6g\n", 0.00001);}’ # Prints 1e-05
awk ’BEGIN {printf("%.6f\n", 0.00001);}’ # Prints 0.000010

# Setting separator symbol to be "," instead of whitespaces.
awk -F, ’program’ input-files

# Prints the line number
awk ’{ print FNR }’ input

# Own functions usage
awk ’function foo(num) { print sqrt(num) } { foo($NF) }’ test
awk ’func foo(num) { print sqrt(num) } { foo($NF) }’ test
```

Notice: as in `bash` scripts `sed` and `awk` support “magic numbers” in the beginning of the script. Put `#!/bin/sed -f` or `#!/bin/awk -f` as first line of the script and make `chmod +x scriptname` and the script starts to be “self-executable”, e.g. `./scriptname` for call.