Awk = Pattern Scanning and Processing Language

- Alfred Aho, Peter Weinberger, Brian Kernighan
- General purpose programming language for processing text-based data
- 1978 – nice little language, that was being used for more significant tasks that intended
- 1985...87 – major developments & improvements
- 1989 – accordance to POSIX standard
- Available in a standard Unix environment
- Key features: string data type, associative arrays, regular expressions and dense notation

"Hello world"

```
echo 'Hello world' > file

a) awk '{print "Hello world"}' file
b) awk '{print }' file
c) awk 'BEGIN {print "Hello world"}'
```

- Awk programs are sequences of instructions (similar to other programming languages, rather than patter-matching processing)

- The program execution is triggered by lines in provided files

If at least one instruction exists, awk will wait for data from stream or file

- The whole program is applied to each line of file

Some program sections may be executed one time only: in the beginning and/or in end of the processing.

Awk Scripts

- Awk scripts in separate files may be invoked:

```
awk -f script_file data_file
```

- Comments begin with '#' and ends at newline
- Awk allows comments anywhere in the script

Operation Model

instructions executed once,
before any input

main input loop –
executed for each input line

instructions executed once
after all input lines are processed

```
BEGIN
{                    }

{                    }

...

{                    }

END
{                    }
```

Pattern Matching

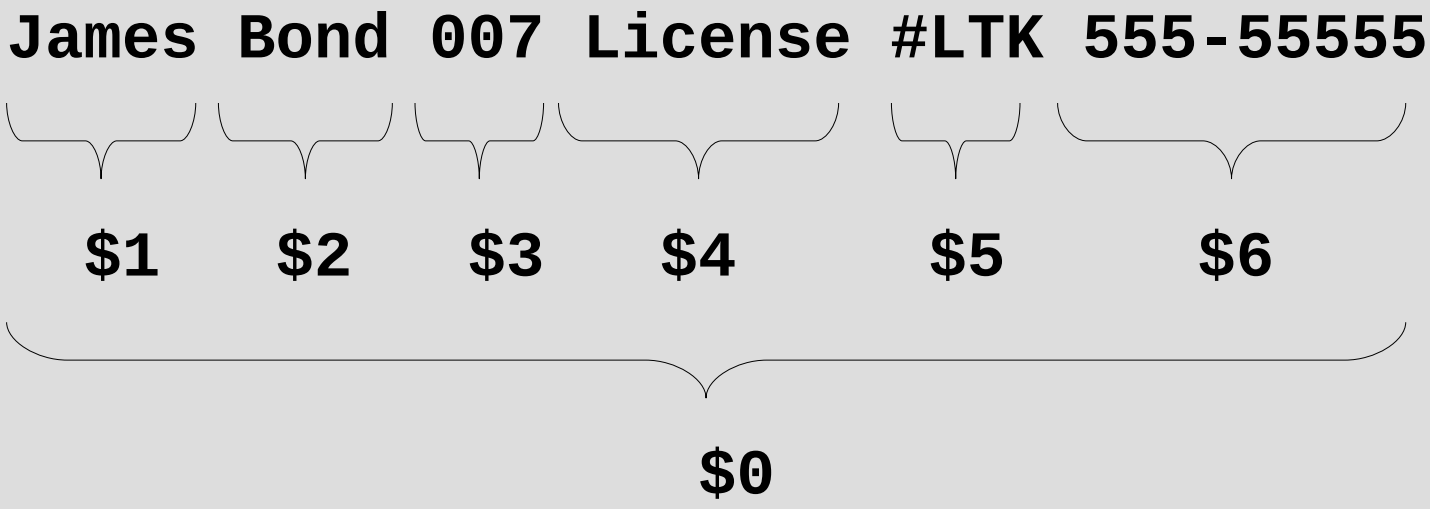
- Awk actions may have associated (reg.exp.) patterns
- Only lines matching the pattern are processed
- A line can match more than one patterns → more actions
- If no patterns is specified for action, all lines are procesed
- If no action is specified for pattern, matching lines are printed

```
/ pattern / { action }

e.g.
   # simple token identification script
   /[0-9]+/ {print “integer”}
   /[a-zA-Z]+/ {print “string”}
   /^$/ {print “empty”}
   {print “ token”}
   /^[^0-9a-zA-Z]+$/
```

Records and Fields

- Data lines treated like records:
  - words (fields) separated with white chars (delimiters)
- Each field may be referenced as **$n**



Field Separator

- Change separator from command line:
  - option -F

```
awk -F ”\t” '{print $1}'
```

- Using build-in variable FS:
  - definition of FS in BEGIN section

```
BEGIN { FS=”,” }
```

- Output Field Separator (OFS)
  - space by default
  - used in print statements with comas:

```
print $1,$2,$3
```

Field Separator

- If FS is single space (default), then separation is any number of white characters
- FS is expected to be a full regular expression !!!

```
echo "Hello" | awk -F 'l+' '{print $1,$2}'
```

→ He o

- If FS is a single character, fields are separated by that character

```
echo "Hello" | awk -F 'l' '{print $1,$2}'
```

→ He

- If FS is the null string, then each individual character becomes a separate field

```
echo "Hello" | awk -F 'l' '{print $1,$2}'
```

→ H e

Record Separator

- Built-in variable RS – newline by default
- Built-in variable ORS – newline by default

- RS is expected to be a full regular expression !!!

- If RS is a single character, records are separated by that character

- If RS is set to the null string, then records are separated by blank lines

Fields Matching

- Fields ($n) may be used to match lines:
  - field is tested against pattern (with ~ or !~)
  - “~” is pattern matching operator

```
$n ~ /pattern/ {action}

$1 ~  /Hello/ {print}
$1 !~ /Hello/ {print}
```

Variables and Expressions

- Two types of variables (case-sensitive names):
  - string (must be quoted)
  - numeric (arithmetically evaluated – C-like operators)

    x = 2
    y = “Hello”
    z = 200*5 + 4%2
- Strings may be evaluated numerically (non-numerical strings have numerical value of 0)

    z = 100 + y
- Variables are referenced with their names:

    z = x + y

Variables

- “Declared” by assignment
- May not be initialized (default 0)

- Strings may be concatenated with space

```
x = “Hello” “World”
```

- Exist during the whole script processing

```
# count empty lines
/^ *$/ { x++ }
END {print x}
```

Built-in Variables – CONVFMT & OFMT

- CONVFMT – controls number-to-string conv.
  - C-like format string (“%.6g” by default)

```
x = 1/3 “ PLN”; print x

0.333333 PLN        (%.6g)
0.33                (%.2g)
0 PLN               (%d)
3.333333e-01 PLN      (%e)
```

- OFMT – controls number conv. in print

```
x = 1/3; print x,”PLN”

0.333333 PLN        (%.6g)
  0 PLN             (%3d)
3.333333e-01 PLN      (%e)
```

Formatted Printing

- printf statement borrowed from C-languge
  - **`printf (format-string,arguments)`**
  - printf does not automatically supply a newline
    `printf (“Hello world\n”)`
    `printf (“%d\t%f\t%s”,$1,$3,$5)`
    `.`
  - fomat can be specified dynamically
    `printf (“%*.*g”,7,2,$5)`
    (7 becomes width and 2 becomes precision)

Built-in Variables – NF & NR

- NF – number of fields in the current record
  - set by awk dynamically for each input line
  - used as a limit for loop processing of fields
  - $NF is the reference to the last field
  - $(NF-1) is the reference to the last but one field, etc.
  -
- NR – number of the current record
  - set by awk dynamically for each input line

- Direct modification of NF, NR is not recommended
  - change of FS affects the next record

Multiline Records - Example

- Processing of paragraphs separated by blank line and preserving the same output format

```
BEGIN { FS=”\n” RS=”” }

{
... each input line is a field ...
}

END { OFS=”\n”; ORS=”\n\n”}
```

Conditional Processing

- Expressions can be used in place of patterns, controlling the execution of actions
- C-like relation operators are allowed
  - process particular record:
    ```
    NR=10 { action }
    ```
  - process particular range of records:
    ```
    NR>1 { action }
    NR>1 && NR<10 { action }
    ```
  - process records of particular size:
    ```
    NF>3 { action }
    ```
  - mixed conditions:
    ```
    NR>2 && NF=3 || $1 ~ /[XYZ]/ { action }
    !(NR=1 || NF>10) { action }
    ```

Example

```
ls -l
drwx------ 4 maranda users   4096 2007-10-03 22:04 Desktop
drwx------ 2 maranda users   4096 2007-08-09 22:58 Documents
-rw-r--r-- 1 maranda users     22 2007-10-04 02:30 file
-rw-r--r-- 1 maranda users 196427 2007-08-09 23:11 g2_test_clip.eps
-rw-r--r-- 1 maranda users 196382 2007-08-09 23:11 g2_test.eps
```

```
BEGIN {
   print “Bytes”,”\t”,”Files”
}
NF==8 && /^-/ {
   sum+=$5; ++filenum
   print $5,”\t”,$8
}
NF==8 && /^d/ {
   print “<dir>”,”\t”,$8
}
END {
print “Total:”,sum,” bytes in “,filenum,” files”
}
```

Passing Parameters to Script

- Bash-like positional parameters ($1...) have other meaning in awk and do not represent parameters
- Awk accepts variables from command line:

```
awk -f script var1=value1 var2=value2 ... file

awk -f script start=1 msg=”Hello” file
awk -f script x=100 y=200 file
awk -f script FS=”,” file
```

- Command-line paramets are not available in BEGIN section, but only after the first line is read

Conditional Statement

Similar to C-language:

```
if (expression) command

if (expression) command1 else command2

if (expression) {
command1
command2
}

if (expression1) command1
else if (expression2) command2
else if (expression3) command3
else command4
```

Conditional Operator ”?”

- Similar to C-language

```
expression ? command1 : command2

x = ($1 == 0) ? “yes” : “no”
```

Loops

- Similar to C-language

```
while (condition) command

do command while (condition)

for (set; test; increment) command

   for (i=1; i<=NF, i++) ...
```

  – **`break & continue`** work as expected

Example

- Calculating factorials (awk -f script):

```
BEGIN { printf("Enter number: ") }
$1 ~ /^[0-9]+$/ {
        number = $1
        if (number == 0)
                fact = 1
        else
                fact = number
        for (x=number-1; x>1; x--) fact*=x
        printf("%d! = %d\n",number,fact)
        exit
}
```

Main Input Loop – next, exit

- Both affect the main loop of awk processing
- **`exit`** – exits the main loop and pass control to END section, if there is one
- **`exit n`** – returns an exit status of the script
- **`next`** – next line of input to be read and processed by the main loop of script

exit

- **`exit`** can occur in BEGIN, main-loop and END sections
- if exit status was specified, the last exit without a parameter will keep the status value

```
{
   ...
   exit 1
   ...
}
END { ...
   exit
}
```

next

- next breaks the actions on current line, reads the next one and starts the main-loop
- some parts of the script may be avoided

```
{  action before }
{
        action
        next
}
{ action after }
```

next – Example

- FILENAME – built-in variable containing the name of the current input file

```
FILENAME == "myfile" {
        action
        next
}
{other actions}
```

  - processing of input from a selected file

Arrays

- Variable to store a set of values
- Elements are added by assignments and accessed by numerical index

```
array[index] = value
variable = array[index]
```

- Size is not declared

```
{ A[NR] = $1 }
END {
    for (i=1, i<=NR; i++)
        total += A[i]
    average = total /NR
}
```

Associative Arrays

- Index can be an arbitrary string
  - e.g. index "007" is not a numerical value 7
- Flexible description of items
  - e.g. **`array["word"]="definition"`**
- Dictionary-like data structures
  - e.g. **`array["item"]="present"`**
- The order of elements is likely to depend on awk implementation
- Special looping syntax

Loops for Associative Arrays

- Loops iterate over a list of indexes

```
for ( variable in array )
    action
```

  - **`variable`** is a temporary variable, with the same role as a counter in a conventional loop
  - **`var in array`** syntax can be applied also to numerical indexes, but order may by random

```
for (item in dictionary) {
    print item
    print dictionary[item]
}
```

Testing for Membership

- Keyword in can be used in conditional expressions to test the existence of item
- Expression **`item in array`** returns:
  - 1 if **`array[item]`** exists
  - 0 otherwise
- Array items can be deleted:

```
delete array[index]
```

Example – Acronym Lookup

```
FIFO    First-in Last-out
LIFO    Last-in First-out
GIGO    Garbage-in Garbage-out
```

```
awk -f x acronyms -
```

```
BEGIN { FS="\t"; print "Enter acronym: " }

FILENAME == "acronyms" {
        dict[$1]=$2
        next
}
$0 ~ /^([Qq]uit|[Ee]xit|[Bb]ye)$/ {exit}
$0 != "" {
   if ($0 in dict) print dict[$0]
   else print "not found" }
{ printf("Enter acronym: ") }
```

Creating Arrays with split

- **`split`** function parse a string into an array using a defined separator

```
n = split( string, array, separator )
```

- Indices start from 1 and go to n

```
n = split($2,array,” ”)
for (i=1; i<=n; i++) print i,array[i]
```

- Default separator is FS
- Separator can be a regular expression
- If separator is empty string, the string is split into separate characters

Multidimensional Arrays

- Awk does not supports true multidimensional arrays
- Multidimensional arrays are implemented as one-dimensional
- Multidimensional indices are treated as string uniquely identifying the element

```
array[2,7] : array["2, 7"]
```

(actually, the indices are concatenated with a nonprintable character \034 – SUBSEP variable - “2\0347”)

More on Parameters Passing

Built-in variables:

- ARGV – array of command line parameters, index starts with 0
- ARGC – number of elements in ARGV

```
awk -f script -a -b name=value datafile -
 0              1  2     3          4    5
```

- ENVIRON – array of system environment variables

Arithmetic Functions

- cos(x), sin(x), atan2(y/x)
- atan2(0,-1) = Pi
- exp(x) – e to the power x, exp(1)=e
- log(x) – natural logarithm
- int(x)
- sqrt(x)
- rand() – random number in range [0,1)
- srand(x) – seed for rand()

String Functions

- gsub(r,s,t) – substitute globally
- sub(r,s,t) – substitute ones
- index(s,t) – position of substring
- length(s) – length of string
- match(s,r) – position of a found match
- sprintf(format,list) – same as in C
- substr(s,p,n) – returns substring from position
- tolower(s), toupper(s)

Functions

- User-defined functions are allowed:

```
function name (parameter-list) {
    function body
}
```

- All global variables are accessible in functions
- Variable defined inside functions are global
- Functions may have local variables (declared in the parameter list)
- Functions may return a results with **`return`**

Function - Example

```
# with global variables: after and below
function insert(string, pos, ins) {
    before = substr(string,1,pos)
    after = substr(string, pos+1)
    return = before ins after
}

-----------

# with local variables: after and below
function insert(string, pos, ins, before, after) {
    before = substr(string,1,pos)
    after = substr(string, pos+1)
    return = before ins after
}
```

Function Library

- General purpose functions can be kept in separate, system-wide, file – awk library
- awk can read multiple source files (-f)

```
awk -f main.awk -f/usr/local/share/awk/lib.awk file
```

Input Reading - getline

- **`getline`** – read another line of current input, other files and pipes
- similar to **`next`**, but does not pass control to the top of the script
- after **`getline`**, $0, NF and NR are set with new values
- the output of **`getline`** can be assigned directly to a variable, $0, NF – not modified, NR - yes

Basic Usage of getline

Returns:

- 1 – if successful
- 0 – if encounter end-of-file
- -1 – if error while reading

```
/pattern/ {
    getline
    print $0
}
---------
/pattern/ {
    while (getline > 0)
        list = list $0
}
```

Reading From Files

- getline can access files with "<" (bash) operator

```
getline < file    # from a variable containing a file name
getline < "file" # "file" is a literal file name
---------
while ( ( getline < "file" )  > 0 )
  print
```

- getline can read from stdin: "-"

```
BEGIN {
  printf("Enter your name: ")
  getline < "-"
  print
}
```

Assigning Input to Variable

- **`getline var`** – $0, NF are not modified

```
BEGIN {
  printf("Enter your name: ")
  getline name < "-"
  print name
}
```

- Do not use **`var = getline`** !

Reading Input from Pipes

- **`getline`** can read lines from a pipe:

```
"command" | getline
```

- **`"command"`** wil be executed as system command with all given options

```
while ("who" | getline )
    who[$1]=$0
--------
# awk -f script /etc/passwd -
BEGIN {"whoami" | getline name; FS = ":"}
name ~ $1 {print $5}
--------
/@date@/ {
  "date +'%a., %h %d, %Y'" | getline today
  gsub(/@date@, today) }
```

Redirecting Output

- Output can be redirected to files and pipes with (bash) operators ">" or ">>"

```
print > "filename"

print | "command"
```

- Redirection inside awk scripts should be avoided in favor of piping output externally

```
awk -f script data | command
```

Function system()

- **`system()`** executes command and returns exit status instead of the output
- **`system()`** halts the script until the command is executed and finished

```
BEGIN {
  if (system("mkdir temp") != 0)
    print "Cannot create directory"
}
------------
{
  if (system("test -r" $1) != 0)
    print "File" $1 " not found"
}
```