AWK

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- Introduced in 1977 by A. V. **A**ho, P. J. **W**einberger, B. W. **K**ernighan
  - Process text files
    - Extracting data
    - Performing statistics
    - Text processing
    - etc.

Introduction

- Main features
  - automatically separates text input into records and records into fields
  - Records and fields are automatically numbered
- It allows
  - Using regular expressions
  - C syntax: variables, operators, constructs, etc.
- An AWK command can be run through
  - the command line
  - script files

Command line execution

```
awk [options] 'command' [file1] … [filen]
```

-v var=val
Defines variable var and set its value

single command

[file$_i$]: command is applied to one or more files

```
awk [options] –f command_file [file1] … [filen]
```

File including sequences of commands

Executes the commands in command file on each file [file$_i$]

Script execution

```
#!/usr/bin/awk

...
commands
...
```

Definition of script (**script.awk**)

Or : **/bin/awk** (**which awk**)

```
script.awk [file1] … [filen]
```

Script execution
(file must have x permission

Records and Fields

- AWK automatically splits the input file into records
  - The record separator is defined by the contents of the built-in variable **RS**
  - The default record separator **`'\n'`**
  - **RS** value can be replaced by another string
  - Records are processed one at a time

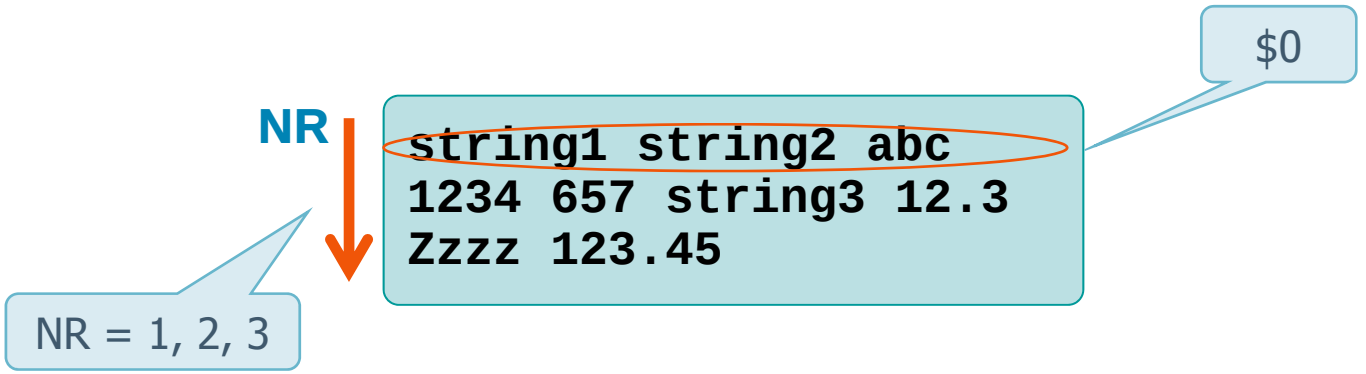
Records and Fields

- AWK automatically splits each record into fields
  - The Field Separator is defined by the contents of the built-in variable **FS**
  - The default field separator is any sequence of the space characters
  - **FS** value can be replaced by another string

Records and Fields

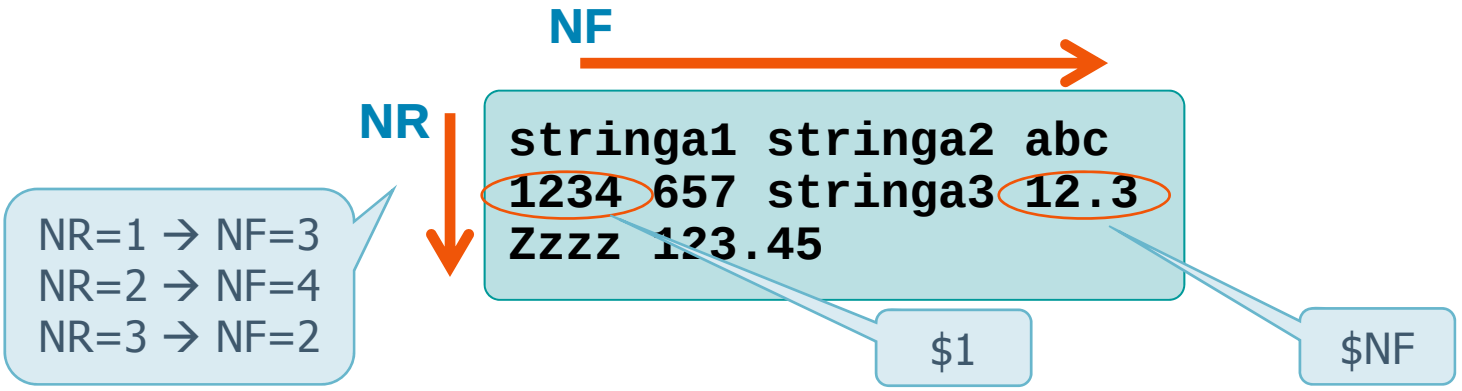
- Some other predefined variables allow the manipulation of records and fields
- **NR**
  - Number of read records
- **$0**
  - Indicates the entire record

For multiple input files **NR** is the total number of read records, whereas **FNR** variable resets itself for each file of the command line



Records and Fields

- **NF**
  - Number of current record fields
- **$1, $2, …, $NF**
  - Indicate the sequence of fields
  - If last field is $5, $6 is a null string



Commands

- An AWK script consists of a sequence of commands
- Each command has the following format

  condition {action}

  - In AWK the condition is usually called **pattern**
- **For each record** of the input file
  - If the pattern (condition) matches, the corresponding action is executed

Commands

- Normally a condition is associated with an action
  - A condition with **no action prints** each record that matches the condition
    - In other words, the default action is the command **print**
  - If the **condition is missing** the corresponding action, it is considered TRUE, thus the action is **executed on every record** of the file

Conditions

- Null -> action is executed for each record
- /regExp/
  - Regular expression(similar to find, grep, sed, …, but different)
  - True if the record matches the regExpr
- exp
  - Logical expression, true if not zero or not null
- exp_1, exp_2
  - Pair of comma separated expressions
  - Specify a range of records
    - The condition is true from the record matching the first condition to the record matching the second condition (included)

Numeric values

Strings

Conditions

- BEGIN
  - **Special condition**
    - Commands executed **before** reading the input file(s)
- END
  - **Special condition**
    - Commands executed **after all the records** of the input file have been processed

Operators and expressions

- Operators
  - Arithmetic
    - +, -, *, /, %, ^ (or **)
  - String concatenation
    - str_1 str_2 str_3 ...
  - Comparison
    - ==, !=, <, >, <=, >=
  - Logical
    - &&, ||, ! (negation)
  - Comparison with regular expressions
    - ~, !~

Cannot use == or != with a regExp

Conditions examples

```
# The record must contain "foo"
/foo/

Field #2, is exactly "foo"
$2 == "foo"

# Condition true from the record with $1
# equal to "rm" to the record with $1 equal "ls"
$1 == "rm", $1 = "ls"

# $1 must contain character 'J'
$1 ~ /J/

# The record must contain "2400" and "bar"
/2400/ && /bar/

# The record must not contain "xxx"
!~ /xxx/
```

Actions

- An action may include
  - All the major C language operators
- Several typical AWK operators
  - Among them
    - Functions for input and output
    - Operators and expressions
    - Scalar variables
    - Control and iteration constructs
    - Vector variables (**associative**, i.e., hash-table)
    - functions

Input

- AWK takes its input stream from
  - Files
    - The files are listed in the command line, and sequentially processed
    - The built-in variable **`FILENAME`** indicates the currently processes file
  - Standard input
    - In this case **`FILENAME`** is equal to "**`-`**"
- Additional files can be processed by using the **getline** command (described later)

Input

- The arguments on the command line are stored in the predefined variables **ARGC** and **ARGV**
  - **ARGV[0]**
    - Stores the name of the script
  - **ARGV[1] ... ARGV[ARGC-1]**
    - Store the arguments
- They can be modified at run-time
  - Decrementing **ARGC** or storing **""** in **ARGV[i]** deletes a file from the input stream

```
for (i=0; i<ARGC; i++)
  print ARGV[i]
```

Displays the command line

Output

- The output commands are quite standard
  - print [p1] ... [pn] [> file]
    - If the arguments p_i are not separated by commas
      - they are printed without the separator
    - Print is completed by a newline
    - Output can be redirected to a file
  - printf (format, ...) [> file]
    - Print formatted with C-like syntax
    - Output can be redirected to a file

Format string inside "" or given as a variable

Variables

- Definition
  - Have not associated type
  - Are considered strings or floating-point numbers depending on the context
  - Numeric variables are automatically initialized to **0**
  - String variables are automatically initialized to **""**
- Usage
  - C-like
    - Character **$** is **not needed** as imposed by bash

Control and iteration statements

- AWK defines all main statement occurring in C language
  - Conditional
    - if
  - Iterative
    - while, do-while, for
  - Flow control
    - break, continue
    - exit
      - Goes to the end of the file, and executes pattern **END**, if it exists.
    - next
      - Skips next record of the file

"one-line" examples

Prints the first and last field of the lines including string "foo"

Prints all lines whose first field includes string "foo"

```
> awk '/foo/ {print $1, $NF}' file.txt

> awk '$1 ~ /foo/ {print $0}' file.txt
> awk '{if ($1 ~ /foo/) print $0}' file.txt

> awk '{print $NR}' myFile.txt

> awk '{if (NF > 0) print $0}' in.txt

> awk 'length($0) > 80' in.txt

> awk '{$2 = $2 - 10; print $0}' \
    file.txt
```

Prints the i-th field of the i-th line

Prints the lines with at least a field

Prints the lines with length greater than 80 characters

Modifies field 2, and prints its new value

"one-line" examples

Computes the sum, and prints the result

Output redirection on file out.txt

Output on file `f.txt`

Prints the maximum number of fields found in all records of a file

Print the number of lines that include string "foo"

```
> awk '{v=($5+$4+$3+$2) ; \
  print v}' file.txt

> awk '{$6=($5+$4+$3+$2) ; \
  print $6; print $0 >> "out.txt"}' file.txt

> awk -v var=f.txt '{$6=($5+$4+$3+$2); \
  print $6; print $0 > var}' file.txt

> awk '{if (NF > max) max = NF}
    END {print max }' file.txt

> awk 'BEGIN {print "Analysis of foo"}
  /foo/ {++n}
  END {print "foo appears " n " times."}'
     file.txt
```

Example

```
#!/usr/bin/awk -f

BEGIN {
  SIZE = 80
}

{
  l = length($0);
  for (i=0; i<(SIZE-l)/2; i++)
    printf " ";
  printf "%s", $0;
  for (i=0; i<(SIZE-l)/2; i++)
    printf " ";
  printf "\n";
}
```

Prints, centered, the lines of a file

Line width **SIZE** (characters)

Arrays

- In AWK **arrays are associative**
  - In practice, they are implemented as a hash table
- The **index is a string** (even if it is a number)
  - It is the key of the hash table
  - The value of the element can be of any type (integer, string, etc.)
- There is no need to specify the size of an array
- An assignment to a new element adds that item to the array (a new association <key-value> is added to the hash table)

Arrays

- Operations
  - Assigning an element
    - **`arrayName [index] = value`**
  - Reference an element
    - **`arrayName [index]`**
    - If the element does not exist is the **`0`** or the **`null string`** is returned (depending on context)
  - Deleting an element or an entire array
    - **`delete arrayName [index]`**
    - **`delete arrayName`**

Arrays

- Operator **in**
  - **`index in arrayName`**
    - allows verifying the existence of a specific array index (key)
    - if **`(index in arrayName)`** ...

Condition **TRUE** if **index exists** in arrayName

  - **`for (var in array)`**
    - Variable **`var`** takes the value of each element of the array (i.e., of each key of the hash table, the order depends on the hash table implementation)

Multi-dimensional arrays

- It is possible to simulate multi-dimensional arrays
  - An element is identified by a sequence of indices, which are concatenated into one string using a separator character
  - symbol `'@'` is the default separator character, defined in the predefined variable **SUBSEP**
- **`pixel[x,y]`** is converted into **`pixel["x@y"]`**
  - vet[a,b,c], vet ["a","b @ c"], vet ["a @ b @ c"] are indistinguishable, because their key is "a@b@c"
- Operator
  - **`(index`$_1$`, index`$_2$`, ...) in arrayName`**

Example

```
# Notice: indices of arrays are strings
# An index not initialized corresponds to
# string "", not to 0

vet[index]=5

# Notice: the void string exists in the array:
# it is key ""

vet[4] = ""
if (4 in vet)
  print "element exists"
delete vet[4]
if (4 in vet)
  print " element exists "  # is not printed
```

Example

```
#!/usr/bin/awk -f

BEGIN {
  n = 1
}

{
  array[n] = $0
  n++
}

END {
  for (i=n-1; i>0; i--)
    print array[i]
}
```

Prints in reverse order the lines of a file: last line becomes the first, and vice-versa

Example

```
#!/usr/bin/awk –f

{
  for (i=1; i<=NF; i++)
    freq[$i]++
}

END {
  for (word in freq) {
    printf "%s\t%d\n", word, freq[word]
    if (length(word ) > 10) {
      ++num_long_words
    }
  }
  print "NumLongWord:" num_long_words
}
```

Prints the absolute frequency of the words in a file, and then the number of those having more than 10 characters

Example

```
#!/usr/bin/awk -f

{
  if (maxNC < NF)
    maxNC = NF
  maxNR = NR
  for (i=1; i<=NF; i++)
    matrix[NR, i] = $i
}
END {
  for (c=1; c<=maxNC; c++) {
    for (r=1; r<=maxNR; r++)
      printf("%s ", matrix[r, c])
   printf("\n")
  }
}
```

Reads a matrix and computes the dimensions of its rows and colums

Displays the transpose matrix

`getline` command

- The **`getline`** command allows
  - Reading the next record
  - Reading a record from another
    - **`getline [var] [<otherFile]`**
- Reads the next record
  - From the current file (from otherFile)
  - In **`var`** (or in **`$0`**if **`var`** is not indicated)
- Return value
  - 1 if it reads a record
  - 0 at the EOF

otherFile defined by:
- A string
- ARGV [i]
- An external variable (-v var = ...)

getline examples

```
getline
```

Reads in **$0** the next line from the input file, (**$0** of the previous read line is overwritten)

```
getline < "new.txt"
```

Reads in **$0** the next line from file **new.txt**, (**$0** of the previous read line is overwritten)

Reads in **tmp** the next line from the input file. **$0** and **NF** do not change, the fields in **tmp** are not split

```
getline tmp
```

Reads in **tmp** the next line from a file given in the command line. **$0** and **NF** do not change, the fields in **tmp** are not split

```
getline tmp < ARGV[2]
```

Example

```
#!/usr/bin/awk -f

{
  if (getline tmp) {
    print tmp
    print $0
  } else {
    print $0
  }
}
```

Swaps even and odd lines of a the input file

Reads odd lines NR=1,3,5,..

If **getline** returns 0 the file is terminated with an "odd position line"

Block of statements executed for every line of a file

Example

Translates all the words in a file using a vocabulary read from file "**voc.txt**"

```
#!/usr/bin/awk -f

BEGIN {
  while (getline < "voc.txt")
    voc[$1]=$2;
}

{
  for(i=1; i<=NF; i++){
    if ($i in voc) {
      printf ("%s -> %s\n", $i, voc[$i]);
    } else {
      printf ("%s -> ?\n", $i);
    }
  }
}
```

Functions

- It is possible to use predefined, or user functions
  - numerical mathematical functions
    - **`int(x), sqrt(x), exp(x), log(x), sin(x), rand(),`** etc.
- Functions for string manipulation
  - **`length(str)`**
    - Returns the length of the string `str`
    - If **`str`** is a number, returns the length of the number converted to a string
  - **`toupper(str), tolower(str)`**
    - Return **`str`** converted to uppercase or lowercase

Not introduced

Functions

- **`system(command)`**
  - Executes a shell to run the command
  - The input (output) of the shell command is not available to the AWK script
  - The AWK script only receives the termination code of the system command
  - Note that the shell commands can also be entered in the AWK script simply specifying the command **in quotes**
    - **`"command"`**
  - In this case the input and output of the command must be managed directly from AWK script (assigning to a variable, or piping the output to **`getline`**)

Functions

```
system ("ls -laR /home/foo");
…
system ("ls -laR | sort");
```

Output is generated on the shell that executes the command

```
cmd = "ls -laR /home/foo";
while (cmd | getline > 0 ) {
   ...
}
close (cmd);
```

Similar to reading from a file. Output is generated on the shell.

A single pipe at a time
Close the pipe

```
awk '{print $1 | "sort" }' inFile.txt
```

Sorts file **inFile.txt**, displays the sorted output

```
awk '{print $1 | "sort > outFile.txt" }' inFile.txt
```

Sorts file inFile.txt, sorted output goes on outFile.txt

Functions

- **`match(str, regExp)`**
  - Search regular expression **`RegExp`**, in string **`str`**
  - Returns the index of the first character of the first occurrence of the substring of the string **`str`** that matches **`RegExp`**
  - Characters are numbered starting by **`1`**
  - Returns **`0`** if no match

```
index = match ($0, $2);
```

Returns the initial index of the substring in current line, which matches the string in $2, or 0 if no matching

Functions

- **`gsub (regExp, str [, src])`**
  - It replaces in string **`src`** each occurrence (not overlapping) of the regular expression **`regexp`** with the string **`str`**
  - If **`src`** is not present, the replacement is carried out in **`$0`**
  - Character **`'&'`** in **`str`** is replaced with the string that has matches
  - Returns the number of substitutions

```
gsub (/husband/, "wife", str);
```

Replaces "**husband**" with "**wife**" in **str**

```
gsub (/husband/, "& and wife");
```

Replaces "**husband**" with "**husband and wife**" in **$0**

Functions

- **`split (str, vet [, del])`**
  - Splits string **`str`** in substrings according to a delimiter **`del`** , each substring is stored in the **vet** array
  - Returns the number of **`vet`** array elements

Splits "**`this-is-a-spli`**t" in 4 substrings delimited by "**`-`**"

```
split ("this-is-an-example", vet, "-");
```

`vet[1]="this", vet[2]="is", vet[3]="a", vet[4]="example",` returns **`4`**

Functions

- **`substr (str, i [, n])`**
  - Return a maximum of **`n`** characters of string **`str`** starting from its **`i-th`** character
  - If **`n`** is not specified, returns the substring of **`str`** that starts from its **`i-th`** character

```
substr ("washington", 5, 3);
```

Returns "**`ing`**"

```
substr ("washington", 5);
```

Returns "**`ington`**"

Exercise

- A text file does not contains punctuation characters
- Write an script AWK that
  - Takes the name of a file from the command line
  - Displays the histogram of the number of occurrences of all strings of length 5 containing at least two vowels

Exercise

- Example

Content of the file

```
abbey enemy car stores abbey figure table
table enemy table
aaaaa  source three
```

```
three #
table ###
aaaaa #
enemy ##
abbey ##
```

Script output

Solution

```
#!/usr/bin/awk -f
BEGIN {
  vowels="aeiou"
}
```

Prepares the set of characters of interest

Solution

```
{
  for(i=1;i<=NF;i++) {
    if (length($i)==5) {
       split($i,v,"");
       found=0;
       for(j=1;j<=5;j++) {
        if (match(vowels,v[j]))
          found++;
      }
      if (found>=2) {
        count[$i]++;
        if (count[$i] == 1)
          words[$i]=$i;
      }
    }
  }
}
```

Splits each word in characters

Arrays count and words are indexed by the same key

Solution

```
END {
  for(w in words) {
    printf "%s ", words[w];
    for (j=1;j<=count[w];j++)
      printf "#";
    printf "\n";
  }
}
```

For each word `w` in words, get its count

Exercise

- Write an AWK script that
  - Gets from the command line three filenames (amount, price, and output)
  - Displays
    - The number of products that have their amount specified but not their price
    - The number of products that have their price specified but not their amount
    - For products that have both their price and amount specified, a line for each product, specifying its total availability, its average price, and the commercial value of the product (product of amount and average price). Save also this information in the output file

Solution

- Example

Amount file

```
Book 3
Pen 10
Pencil 4
Book 2
Pen 20
Pencil 3
Eraser 3
Book 8
Eraser 1
```

```
Book 50.5
Pen 5.4
Pencil 2.0
Book 20.5
Pen 4.2
Pencil 1.0
Book 18.2
Jotter 12.3
```

Price file

Output file

```
Pen 30 4.8 144
Pencil 7 1.5 10.5
Book 13 29.7333 386.533
Warning: product Eraser has no price!
Warning: product Jotter has no quantity!
```

Solution

```
#!/usr/bin/awk -f
BEGIN {
  fileO = ARGV[3];
  ARGV[3] = "";
  print "" > fileO
# File 1 - Amount
  while ((getline < ARGV[1])) {
    q[$1] = q[$1] + $2;
    n[i]++;
  }
  # File 2 - Price
  while ((getline < ARGV[2])) {
    p[$1] = p[$1] + $2;
  }
}
```

Save the output filename, and clears ARGV[3]

Reset the output file

Compute the total amount, and the occurrences of each product

Compute the sum of the prices of each product

Notice: the input files are read in the BEGIN section

Solution

```
END {
  for (i in q) {
    if (i in p) {
      p[i] = p[i] / q[i];
      print i " " q[i] " " p[i] " " q[i]*p[i];
      print i " " q[i] " " p[i] " " q[i]*p[i] >> fileO
    }
  }
```

Displays, and generates the output file

Solution

```
  for (i in q) {
    if ( !(i in p) ) {
      print "Warning: product " i " has no price!"
    }
  }
  for (i in p) {
    if ( !(i in q) ) {
      print "Warning: product " i " has no amount!"
    }
  }
}
```

Controls missing information