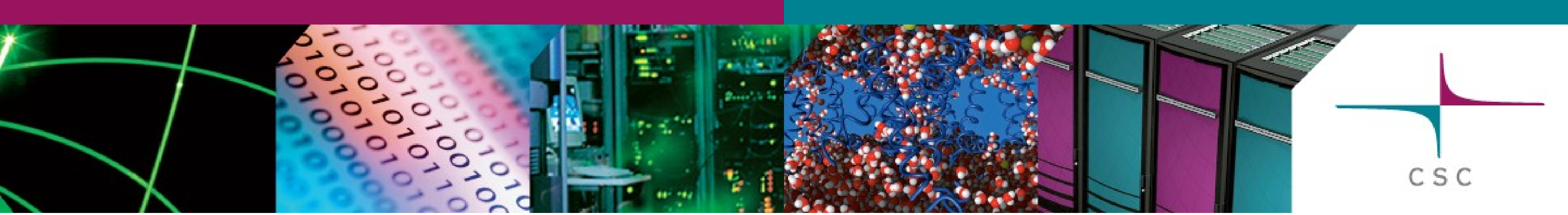


Scripting Techniques : awk & perl basics

Sami Saarinen

CSC

25-Oct-2010

CSC – Tieteen tietotekniikan keskus Oy
CSC – IT Center for Science Ltd.

Contents

- Searching keywords from text files
- Printing only the necessary information
- Replacing text
- Covering features from Unix-commands:
  - awk
  - perl
- No Python in this presentation ☹

Searching from text files

- A very common task is to search particular words from text output files, and perhaps modify and print those
- Typically Unix-command **grep** is used to perform text search
  - **grep** was covered by the earlier presentation
- Here we will explain basic usage of Unix-tools **awk** and **perl** in data filtering context

awk

- by Aho, Weinberger, Kernighan (1977)
- A versatile text processing language which resembles C (hmm... by Kernighan & Ritchie)
- Powerful with spread-sheet / tabulated data
- Typical usage perhaps in one-liners with matching/reordering/formatting/calculating fields from the existing tables of data
- awk command scripting is also available

awk pattern – action chain

CSC

- awk commands essentially match a pattern from a text and apply an action to it :

```
/ pattern /   { action }
```

- For example

```
BEGIN  { x = 100 }
{ print  $1 }
/Janet/ { x = x + $1 ; print }
END  { print  ”x = ”,x }
```

NF, NR, args $0, $1, $2, ...

- **NF** is the number of fields on each line

```
awk '{for (i=1; i<=NF; i++) print $i}'
```

- **NR** is the number of input records (lines)

```
awk 'END {print NR}'  file.txt
```

- Much simpler still : `wc -l file.txt`
- awk fields accessed through variables
  $1 , $2, …, $(NF-1), $(NF)
  - As mentioned the **$0** refers to the whole input

BEGIN and END

- **BEGIN { }** and **END { }** statements are optional in awk and if present, they execute code before and after reading the input
- They are *not* tested against the input
- **BEGIN** is often used to initialize variables before the first input line has been read in
- **END** is usually used to print some summary information after input has been finished

Some awk one-liners

CSC

- Print the 2nd field from each input line :

```
cat  file.txt | awk ’{print $2}’
```

- Take only the lines containing ”ZZZ” :

```
awk ’/ZZZ/ {print $0}’   file.txt
```

The same as : `egrep  ZZZ  file.txt`

- Calculate the sum of the field $3

```
awk ’BEGIN {s=0} {s += $3} END {print s}’
```

Some awk one-liners …

- Normally the field delimiter is a white space, but with option –F you can change it
  - Internal variable FS can also be used
- The following prints fields 1,2,3 from a file, where fields are separated by a colon :

```
awk –F:  '{print $1,$2,$3}'  /etc/passwd
```

- Print whole line only if number of fields > 2

```
awk '{ (NF > 2) {print }'   file.txt
```

Print statement in awk

CSC

- Instead of using generic **print** in awk, it is possible to use C-language like **printf**
- This gives you a full spectrum of C-like formatting capabilities, e.g.

```
awk '{printf("Time = %2d:%2.2d\n",$1,$2)}'
```

- Please do not forget to supply the newline "\n" in **printf** ! The generic **print** already adds that for you – automatically

awk variables

CSC

- awk has predefined variables, user defined variables and arrays
- Predefined contain fields columns (`$1,$2,...`), the whole line (`$0`) or internal variables (in capital letters) like `NF, NR, FS, RS`
- User defined variables are usually given in a lowercase to avoid mix-up, e.g. `a, b, tmp`

Field separator – FS

- Field separator (FS), the same as –F option, can be used to indicate character(s) used to separate consecutive fields
- If you don't want to use the –F option, give

```
BEGIN { FS="[:,]" }
```

- Your FS is either colon or comma, try f.ex.

```
echo “”1:2,3  4” | awk  -F”[:,]”   ’{ print NF }’
```

Record separator – RS

- Similar to FS, the record separator (RS) can be used to turn any character(s) into line breakers
- No command line option for RS
- The following prints out not 1, but 3 lines

```
echo "AA-BB:CC DD" | awk \
        'BEGIN { RS = "[:-\n]" } { print }'
```

awk variable arrays

- awk arrays are in fact *associative arrays*
- This means the *index into an array need not to be an integer number*
- It can be anything from numerical values (even floating point) to character strings :

```
tmp1 [ 80 ] = 1;
tmp2 [15.5] = ”Gazette”;
tmp3 [”Saab”] = 2.0;
```

awk variable arrays …

- Looping through an associative array :

```
for (i in tmp) { print i,tmp[i] }
```

- Note: the order in which the array is scanned through is more or less arbitrary

Functions in awk

- Some numerical functions
  - **int, exp, log, sin, cos, sqrt, …**
- Some string handling functions
  - **substr, match, sprintf, tolower, toupper, …**
- Bit manipulation functions
  - **and, or, xor, lshift, …**
- User defined functions through cmd scripts

Control statements

CSC

- awk contains **if-then-else** statements for conditional computation :

```
awk '{if ($1 > 10) { print "Over 10" }  \
   else { print "Less or equal to 10" ; } }'
```

- Backslash ("\") above just allows to continue awk statements to the next line
- Semicolon (";") is only needed if you have several awk clauses after each other

Loops in awk

- awk contains **for**, **while** and **do-while** loops
- Messy within one-liners – useful in scripts

```
awk  '{for (i=1; i<=NF; i++)  print $i }'
echo "288"   |   awk \
     '{ n=$1; s=int(sqrt(n)); \
       while (s>1 && n%s!=0) --s; \
       print  s":"int(n/s) }'
```

- We explain this mess later on

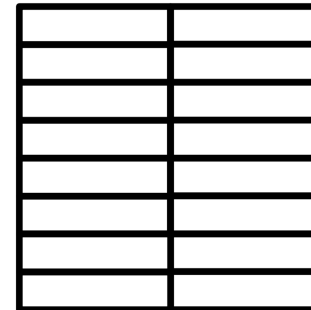
awk command scripts

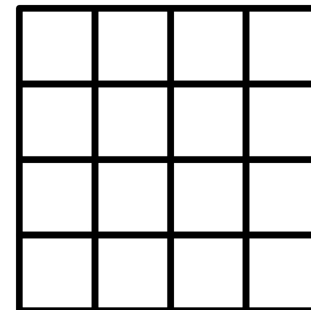
- Sometimes one-liners are becoming a burden due to their complexity or length
- Instead, it is possible to create a command script, where awk statements are put
- Consider awk command script ”**cmd.awk**”

```
awk –f cmd.awk < input > output
awk –f cmd.awk input > output
cat input | awk –f cmd.awk > output
```

awk command script example

- 2D-decomposition of a finite difference mesh
  - Find the most square-like decomposition
  - Usually leads to the best parallel performance
- Given NPROC
  - Find the optimal NPROCX & NPROCY





awk command script example

CSC

- An example script (2d_decomp.awk) :

```
#!/bin/awk -f
/[0-9]+/  {  # Match for a number only
   n = $1     # Total no. of processors
   s  = int(sqrt(n))
   while (s > 1  &&  n%s != 0) --s
   print "NPROCX="s", NPROCY="int(n/s)
}
```

awk command script example

- Running awk command script examples:

  echo ”144” | 2d_decomp.awk

- Returns NPROCX=12, NPROCY=12

  echo ”288” | 2d_decomp.awk

- Returns the most square-like decomposition NPROCX=16, NPROCY=18
  – This leads often better performance in a HPC-code than use of : NPROCX=12, NPROCY=24

For more information

CSC

- Please do read awk Unix manual pages :

  man awk

  info awk

- Web contains a plenty of additional info
  - Do google for instance on ”awk tutorial”

Exercises for awk

(1) Create an awk script, which writes ”Hello World!” :

echo ”Hello” | hello.awk

(2) Given an input file (rovaniemi.txt), which contains climate data for Rovaniemi. Calculate yearly average max and min temperatures. What is the total rainfall per year ?

(3) Continue the previous example and calculate the total number of rainy days for those months, where average min temperature is greater than zero.

(4) Examine alternative ways of coding 2d_decomp.awk

perl

CSC

- Practical Extraction and Report Language
  - by Larry Wall
- Combines to the certain extent features provided by awk, sed, C and shell scripts
- Perl syntax : more practical than beautiful !
- Often as one-liners in text replacements
- Quite common as a scripting language
- Object oriented

A quick Perl syntax helper

CSC

- You often search (match) and replace text
- The following will do both (s = substitute):

```
s/Text/NewText/
```

- Replace Text with NewText on each line:

```
s/Text/NewText/g
```

- Substitute within a variable & ignore case

```
$var =~ s/text/newtext/i
```

A quick Perl …

CSC

- Sometimes – in Perl scripts in particular you just want to match, not replace:

```
if (m/Some text/) …
```

- As with substitute (s), you can use case insensitive match (m) as follows:

```
if (m/some text/i ) …
```

- Match text from a variable

```
if ( $var =~ m/some text/i ) …
```

Perl matching expressions

- Perl often uses the following expressions:
  - Numbers (\d+)
  - Words (\w+)
  - White space (\s+ or maybe a white space \s*)
  - Non-white space (\S+)
  - Word boundaries (\b)
  - The beginning (^) or the end ($) of the line

More on regular expressions

CSC

- Extra meta-characters : often used with Perl:

  \s, \s*, \s+, \b, \w+, \d+, \S+

- The following are used also in awk:
  - **?** the same as star (*), but can be matched only once or not at all
  - **+** as *, but has to be matched at least once
  - **( )** is for grouping, and **|** for or'ring, e.g.

    (Jim|Tom|Ben) matches one of the names shown

Perl one liners

CSC

- Typical Perl usage starts with one liners : **-pe**

```
perl –pe ’s/\bAnn\w*/Susan/g’ < in > out
date | perl –pe ’s/.*?(\d+):(\d+):(\d+).*$/$1$2$3/’
```

- File in-place text replacement :

```
perl –i  –pe ’s/name.*\s*=\s*(\d+)/$1/’ inplace.txt
```

- Conditional printing : **-ne** option

```
perl –ne ’print ”$1\n” if (m/name.*\s*=\s*(\d+)/)’
```

Notes on Perl matching

CSC

- Greedy vs. non-greedy string matching
  - Consider string ”Jack the Ripper”
- The following greedy match

```
perl -pe 's/Jack.*e/ZZZ/'
```

  - Matches until the **last** ”e” and returns ”**ZZZr**”
- Add ”?” after ”*” for non-greedy matching

```
perl -pe 's/Jack.*?e/ZZZ/'
```

  - Matches the **first** ”e” and you get ”**ZZZ Ripper**”

Perl command scripting

- Perl becomes much more comfortable (than one-liners) when used through scripts
- You can write your own rudimentary text parser to filter out and post-process only those lines which you want to include (match)
- Nowadays you can find that many system related scripts are in fact written in Perl (or even Python) rather than (say) in bash / ksh

Perl command script example

- A Perl script (scr.pl) to read lines & print them:

```
#!/usr/bin/perl
use strict;
use warnings;
my @in = <>;  # slurp the whole input
foreach my $line (@in) { chomp($line);   print ”$line\n”; }
```

- Usage:

```
scr.pl < input > output
```

About Perl variables

- Perl allows to specify various variables *without* strong typing
  - it guesses from the context whether we have a string or a numerical value in concern
- The main "categories" are
  - Scalar variables : $var
  - Arrays : @var
  - Associative arrays : %var

Perl variables …

CSC

- Variables $x, @x and %x refer to different variables and can thus co-exist
- Value assignment goes as follows, e.g.

```
$x1 = 1.2 ; $x2 = ”text”; $x3 = ’text’; $x4 = $x3;
@x1 = ( ); @x2 = (1,5,7);
@x3 = (’an’, ”apple”); @x4 = split (/:/, $string);
%x = ( ); $x{”Honda”} = 2.0;  $x{”Volvo”} = 2.3;
%x = ( ”Honda” => 2.0,   ”Volvo” => 2.3,  );
```

Recommendations

CSC

- Always try to include the following two use-clauses and you are less prone to errors
- If ”use strict” is present, then variables must be declared with ”my” –keyword in front of the variable name
- If ”use warnings” is present, then use of uninitialized variables as well as many other Perl warnings are displayed to stderr

Perl filtering with script

CSC

- We continue to enhance the earlier script
- To print only the desired lines, add string matching (e.g. only the lines with ”carrot” – case insensitive match – are printed) :

```
foreach my $line (@in) {
   chomp($line);   # Get rid of  newline ’\n’
   print ”$line\n”  if ($line =~ m/carrot/i) ;
}
```

Perl filtering …

- It is possible to print only the matched items, automatically placed in $1, $2, …
- Instead of "**my $line**", use (often hidden) $_

```
foreach (@in)  {
 # Each line is now implicitly a $_ , in turn
  if  (m/(\d+):(\d+)/)  { # Match from $_
    print "Time : $1 hours,  $2 minutes\n";
  }}
```

Perl filtering …

CSC

- You can also replace (f.ex.) the matched text before writing it out :

```
foreach (@in) { # $_ is now a complete line
   chomp; # Side-effect: changes $_ AND @in
   if (m/some text/i) {  # Match text
     s/other text/ZZZ/; # Replace $_ AND @in
     printf ("%s\n", $_); # C-style print stmt
} }
```

A practical example

- In the following a complete Perl command script will be developed
- It shows some (non-trivial) new features, which are explained later
- The purpose of the script is to turn a range of non-negative numbers into a sorted list :
  - For example : 8-10,1-4 6,18,7-9
  - Is turned into : 1,2,3,4,6,7,8,9,10,18

Script expand_list.pl (1)

```
#!/usr/bin/perl
#
# Usage  :  expand_list.pl   8-10,1-4  6,18,7-9
# Output :  1,2,3,4,6,7,8,9,10,18
# -- The usual recommended precautions
use strict;
use warnings;
# -- Obtain the number of command line arguments
my $numargs = $#ARGV + 1;
```

Script expand_list.pl (2)

```
# -- Concatenate input into a blank separated string
my $input = "";
foreach my $argnum (0 .. $numargs - 1) {
    my $arg = $ARGV[$argnum];
    $input .= "$arg ";
}
# -- Prune $input -string
$input =~ s/\s+$//;   # Remove trailing white space
$input =~ s/\s+/,/g;  # Each white space into a comma
```

Script expand_list.pl (3)

```
while ($input =~ m/(\d+)-(\d+)/)  {
    my $x1 = $1, my $x2 = $2, my $list = "";
    if  ($x1 <= $x2 ) {
        foreach my $x ( $x1 .. $x2 ) {
            $list  .=  "$x,";  #  Occupy the comma -list
        }
    }
    $input =~ s/(\d+)-(\d+)/$list/; # Replace the range
}
```

Script expand_list.pl (4)

-- Further pruning of $input -string

```
$input =~ s/,+/,/g; # Get rid of any multiple commas
$input =~ s/,$//;   # .. as well as trailing comma
$input =~ s/^,//;   # .. and leading comma
```

Script expand_list.pl (5)

```
# -- Split $input into an array of values (= @input)
my @input = split(/,/, $input);

# -- Truly  incomprehensible ”magic” here ... ☹  !!
#      Prepare for picking up just the unique values
my %uniqhash = map { $_ => 1 } @input;
```

Script expand_list.pl (6)

```
#-- Sort hash keys numerically, the smallest first
my @output = sort {$a <=> $b} keys %uniqhash;

#-- Create $output -string by joining the array values
my $output = join(",",@output);

#-- Finally, print  $output -string
print "$output\n";
```

Some explaining to do !

- split & join functions
  - split creates an array from a string
  - join does reverse of split
- sort function
  - Lexical or numerical sorting of an array
- keys (and values) : %hash handling
- map function : shorthand expression

split & join

CSC

- Consider a string (optional white space in red)

  $string = ”AA\t: BB\t:CC:XDD”

- split(/\s*:\s*/, $string) returns an array

  @array = (’AA’, ’BB’, ’CC’, ’XDD’)

- join(”-”, @array) turns it back to a string :

  $string = ”AA-BB-CC-XDD”

sort

- By default sort will reorder an array lexically, not numerically – *pls make a note of this* !!
  - Consider: @x = (2,3,10); @x = sort @x;
  - print join(",", @x) gives 10,2,3   not 2,3,10
  - As if we had string valued as @x = ('2', '3', '10')
- Fix this by using 2-way comparison operator
  - Correct version: @x = sort { $a <=> $b } @x
  - print join(",", @x) gives now : 2,3,10

Associative (hash) variables

- Associative variable consists of (key,value)-pairs, e.g.

```
my %var = ();
$var { ”Tina” } = 20 ;  $var { ”Jack” } = 35 ;
```

- or simply

```
my %var = (  ”Tina” => 20,
              ”Jack” => 35,
            ) ;
```

keys & values functions

- The keys-function is used to extract the keys of an associative variable into an array
- The values-function in turn return array of hash variable values (= ”left-hand-side”)
- Common usages with sort function:

```
foreach (sort keys %var) { print ”$_\n”; }
foreach (sort { $a <=> $b } values %var)
        { printf (”%d\n”,$_); }
```

map

- map function is “just” a shorthand notation to make script more compact (unreadable?!)
- For example our **expand_list.pl** had :

```
my %uniqhash = map { $_ => 1 } @input ;
```

- This is the same as :

```
my %uniqhash = ( );
foreach (@input) { $uniqhash{$_ } = 1; }
```

map …

- Maybe a simple example of map instead ?
- Consider an array of numbers to print

```
my @array = (10, 20, 30);
foreach (@array) { print ”value = $_\n”; }
```

- The foreach-loop with map :

```
map { print ”value = $_\n”; } @array ;
```

- Which one is more readable/obvious to you?

Conclusions

- Perl is a very popular scripting language
- We have merely touched the surface, f.ex.
  - Ignored f.ex. Perl modules, objects, I/O ☹
- More information available in Web
- But start looking from Unix man pages 1st :

```
man perl
info perl
```

Annex : awk beats perl ☺

- The **expand_string.pl** can roughly be replaced by the following “really compact” (but unreadable) awk one-liner : Pls try !!

```
echo "8-10,1-4 6,18,7-9" | awk \
'BEGIN   {FS="-";RS="[, ]";s=1;e=200}
NF>1     {for(i=$1;i<=$2;i++) t[i]=1}
NF==1  {t[$1]=1}
END {for(j=s;j<=e;j++) {if(t[j])printf("%d,",j)} }'
```

Exercises for perl

CSC

(1) Write a perl script, which prints ”Hello World!”

Would it be easier to code this with a one liner ?

(2) Calculate cumulative working hours using the input file working_hours.txt. Times are found in a format HH:MM. Print the cumulative working hours in minutes.

(3) From the Fortran90 source code (file.f90) find out all assign statements (where a variable changes its value)

A hint: assignment is in a format **VAR = expression**

How many constant numerical expressions you can find ?

(4) Examine expand_string.pl script in order to improve it