Unix utilities

grep, sed, and awk

Harold Pritchett
The University of Georgia

Session 561[67]

Abstract

- Before there was perl there was awk. The basic UNIX utilities are grep, sed and awk. Most simple tasks can be done with a combination of these three programs. All UNIX systems have them. A basic knowledge of these three utilities and the text editor vi should be in the tool kit of all UNIX systems administrators. This is a two hour session which will provide a very basic overview of these three utilities.

The Speaker

- **Harold Pritchett**
- The University of Georgia
- (706) 542-0190
- harold@uga.edu

Disclaimer

- Everybody has lawyers:
- The ideas and concepts set forth in this presentation are solely those of the respective authors, and not of the companies and or vendors referenced within and these organizations do not endorse, guarantee, or otherwise certify any such ideas or concepts in application or usage. This material should be verified for applicability and correctness in each user environment. No warranty of any kind available.

Introduction

- Who am I?
- What makes me qualified to talk about this subject?
  - 25 Years working with computers
  - 10 Years experience with Unix
  - Unix Security Administrator
  - Security Incident Handling Team for UGA

Outline of the course

- A quick review of regular expressions
- grep and egrep
- sed
- General structure of awk scripts
- Elementary awk programming
- Advanced awk programming

Regular Expressions

- Three types
  - Shell – used by most unix shells
  - Basic – used by grep and sed
  - Extended – used by egrep and awk

String Matching

- A Regular Expression is always a string matching mechanism.
- The match proceeds left-to-right in simple comparison steps across both:
  - The string being matched and
  - The pattern defining the match

Characteristics

- Matching proceeds left to right
- The leftmost longest match is always made at each step
- Matching is iterative. If a comparison step fails, the match backtracks if possible.

Common Features

- Any character which is not a meta-character matches itself /abc/
- This is UNIX -- all these operations are case sensitive /abc/ vs /ABC/

Common Features

- Classes are described using meta-characters; the most common meta-characters are \ and [and]
- Any character preceded by \ matches itself whether or not it is a meta-character /\[/

Common Features

- Any string of characters in square brackets matches exactly one of the enclosed characters; commonly called a character class /[abcde]/
- ^ as the first character within [] means the complement of the set of characters, not including \n /[^abcde]/

Common Features

- - within (i.e. not first) [] means a contiguous range of characters; this may not work if the processor is not an ASCII-native machine /[a-e]/

Shell Regular Expressions

- Sometime referred to as “wild card” matching
- Automatically anchored to both the beginning and end of the line
- The first regular expression in the pattern must match the first character in the file name and the last regular expression in the pattern must match the last character in the file name

Shell Regular Expressions

- The Bourne, Korn and C shells recognize a common set of metacharacters

? [ ] ! - *

- Escaping in shell REs is rarely necessary, because file names rarely include Metacharacters.

Shell Regular Expressions

- ? - Matches any single character
- * - Matches zero or more characters
- [] - define a character class
- ! - Only within []
- - - Only within []

Basic RE Meta-characters

- ^ at the beginning of the RE, means match only at the beginning of the line
- $ at the end of the RE, means match only at the end of the line
- * means repeat the previous item an indefinite number of times, from 0 up
- . means any character except \n

- Note the difference in behavior of "*" from the shell regular expression

Differences between Shell and General Regular Expressions

Shell	General	Meaning
?	.	Wildcard
*****	**.***	Zero or more wildcards
[ ]	**[ ]**	Character class
[!]	**[^]**	Reverse Character class
re	**^re$**	Fully anchored matching

Extended RE Meta-characters

- **|** alternate choices at this location in the string **/ab|cd/**
- **()** group REs for processing **/(ab|cd)ef/**
- **+** matches one or more of the preceding item **/[0-9]+/**
- **?** matches 0 or 1 of the preceding item **/-?/**

Important things
which will bite you

- Make sure that your RE will not match the null string (or any arbitrary string) Be careful with the use of the asterisk modifier (*)

- Don't forget to protect your scripts from the shell; almost all of the meta-characters for grep|egrep|sed scripts are also shell meta-characters and the shell **will** attempt to interpret them rather than pass them to the utility if you forget to quote them

grep - (g/re/p(rint))

- Search for contents in a file and return matching records

Command line switches

- -c Print only the count of the matching lines rather than the lines themselves
- -v Print only the non-matching lines
- -I Ignore case in matching process
- -h Don't display file names in output
- -l Display only file names
- -n Display the lines numbers with the lines

- Others, less often used; see man grep

Selection, processing, and files

- Every line (un)selected is displayed with possible adornments chosen by the command line switches
- Any further processing must be done by a following program or script
- Multiple files are allowed
- The file name is displayed with each selected line unless suppressed by -h or in one file name case
- Wild card expansion is common *.c, *.h

Examples and uses

- **grep read *.c** -- list all the lines with the word 'read' in them in the files ending .c in the current directory
- **grep -l read *.c** -- list all files whose name ends with '.c' and which contain the word 'read'
- **grep -i pascal *.txt** -- list all the lines from the '.txt' files which contain the word 'pascal' in any capitalization.

egrep (e(xtended) grep)

- Search for contents in a file using the extended form of the regular expression and return contents

Command line switches

- -c Print only the count of the matching lines rather than the lines themselves
- -v Print only the non-matching lines
- -i Ignore case in matching process
- -h Don't display file names in output
- -l Display only file names
- -n Display the lines numbers with the lines
- -f Find the selection expression in the given file name

- Others, less often used; see man egrep

Selection, processing, and files

- Every line (un)selected is displayed with possible adornments chosen by the command line switches
- Any further processing must be done by a following program or script
- Multiple files are allowed
- The file name is displayed with each selected line unless suppressed by -h or if there is only one file name
- Wild card expansion is common ***.c, *.h**

Examples and uses

- **egrep 'SysName|SysID|Ucast' a.1** -- find all lines with any one of the three string in it for later processing
- **egrep '(Patty|Bill) (Smith|Jones|Brown)' phone.book** -- find any/all of the 6 people named in the file phone.book
- **egrep '[0-9][0-9]+|".*"' *.c *.h** -- find any decimal numeric constants or string constants in any of the '.c' or '.h' files in the current directory

Examples and uses

- **egrep ‘[2-9][0-9][0-9]-[0-9][0-9][0-9][0-9]’ notes**
- find a phone number anywhere in the lines of the file notes

sed (stream editor)

- Find and replace based upon content

Command line switches

- -f Take the sed script from the named file
- -n Do not automatically output each input line

General sed command syntax

- Each sed command has zero, one, or two addresses; in the latter case they are separated by a comma.
- Each command may have parameters

General sed command syntax

- These rules lead to the following formats for sed commands:
  - command parameters
  - address command parameters
  - address,address command parameters
- There may be multiple commands in a sed script, separated by \n or a semi-colon

Line selection

- The address(es) may be a line number, $ meaning the current line, or an RE enclosed in slashes.
- Commands without an address select all lines
- Commands with one address select either the line whose number is the address or all lines which match the RE which is the address

Line selection

- Commands with two addresses select: the first line which matches the 1st address, the next line which matches the 2nd address and all the lines between; if the addresses are numbers and the 2nd is less than the 1st, only the 1st is selected
- The command is applied to the selected lines
- The address selection for later commands is performed after processing earlier commands

Common editing commands

- **s/old/new/g** -- substitute the new text for the old text in the current line
- **d** -- delete the current line
- **p** -- print the current line
- **N** -- append the next line to the current line

Common editing commands

- **y/from/to/** -- translate the characters in from to the corresponding characters in to; the strings from and to must have the same number of characters
- **atext** -- insert text into the output before the next line is processed
- **ctext** -- replace the (last) selected line with text
- **n** -- write the buffer and go to the next line

Multiple file processing

- All of the files are concatenated; i.e. the command
- **sed '...' a b c**
- is equivalent to the commands
- **cat a b c | sed '...'**
-
- Therefore line numbers continue through the different files and are not recommended

File Processing

- The command
- **sed '...' x > x**
- does not do what one might expect; in fact the file x is destroyed without possibility of recovery. Instead one must do the following:
- **sed '...' x > ~x~ && mv ~x~ x**
- which will create the temporary file ~x~ and, if sed succeeds, rename it to the original file name

Examples and uses

- **foreach a (*.ltr)**
- **sed 's/SCM/Smith-Corona-Marchant/g' $a >\ $a.fix && mv $a.fix $a**
- **end**
- in each .ltr file this replaces the abbreviation SCM with its full definition; csh syntax is shown, others are similar

- **sed '/^$/d'** -- deletes all absolutely empty lines

Examples and uses

- **sed '/^[ ]*$/d'** -- deletes all lines which appear to be empty; there is both a TAB and a blank inside the []s

awk

- Select records and process

General awk structure
(Aho, Weinberg, Kernighan)

- awk, oawk, nawk, gawk, mawk
- The original (v1) was called awk
- 2nd edition of book led to nawk
- Unices ship with either oawk=awk or nawk=awk.
- gawk is the FSF version.
- mawk is a speedier rewrite.

Command line switches

- -f Read the awk script from the specified file rather than the command line
- -F Use the given character as the field separator rather than the default "whitespace“
- -v variable=value Initialize the awk variable with the specified value

General structure of commands

- selector - action is print
- {action} - selector is every line
- selector{action} - perform action when selector is true
- An awk script may have multiple commands separated from each other by semicolons or \n
- Each action may have multiple statements separated from each other by semicolons or \n
- Comments start with #, continue to end of line

Line selection

- A selector is either zero, one, or two selection criteria; in the latter case the criteria are separated by commas
- A selection criterion may be either an RE or a boolean expression (BE) which evaluates to true or false
- Commands which have no selection criteria are applied to each line of the input dataset

Line selection

- Commands which have one selection criterion are applied to every line which matches or makes true the criterion depending upon whether the criterion is an RE or a BE
- Commands which have two selection criteria are applied to the first line which matches the first criterion, the next line which matches the second criterion and all the lines between them.
- Unless a prior applied command has a next in it, every selector is tested against every line of the input dataset.

Processing

- The BEGIN block is run (mawk's -v runs first)
- Command line variables are assigned
- For each line in the input dataset
  - It is read, NR, NF, $i, etc. are set
  - For each command, the criteria are evaluated
  - If the criteria is true/matches the command is run
- After the input dataset is empty, the END block is run

Elementary awk programming Constants

- Strings are enclosed in quotes (")
- Numbers are written in the usual decimal way; non-integer values are indicated by including a period (.) in the representation.
- REs are delimited by /

Elementary awk programming Variables

- Can not be declared
- May contain any type of data, their data type may change over the life of the program
- Are named as any token beginning with a letter and continuing with letters, digits and underscores

Elementary awk programming Variables

- As in C, case matters; since all the built-in variables are all uppercase, avoid this form.
- Some of the commonly used built-in variables are:
  - NR The current line's sequential number
  - NF The number of fields in the current line
  - FS The input field separator; defaults to whitespace and is reset by the -F command line parameter

Elementary awk programming
Fields

- Each record is separated into fields named $1, $2, etc
- $0 is the entire record
- NF contains the number of fields in the current line
- FS contains the field separator RE; it defaults to the white space RE, /[TABBlank]*/
- Fields may be accessed either by $n or by $var where var contains a value between 0 and NF

Elementary awk programming print

- **print** prints each of the values of $1 through $NF separated by OFS then prints an ORS onto stdout; the default value of OFS is a blank, ORS a \n
- **print value value ...** prints the value(s) in order (without separators) and then puts out an ORS onto stdout;

Elementary awk programming printf

- **printf(format,value,value,...)** prints the value(s) using the format supplied onto stdout, just like c. There is no default \n for each printf so multiples can be used to build a line. There must be as many values as there are format items
- Values in print or printf may be constants, variables, or expressions in any order

Elementary awk programming printf formats

- %s — The argument to print is a string
- %[n]d — The argument to print is an integer to be printed in n columns
- %[[n].m]f — The argument to print is floating point to be printed in n columns with m decimals
- %c — The argument to print is an integer to be converted to a single character

Elementary awk programming
Operators

= += -= *= /= %=	The C assignment operators
~ !~	Matches and doesn't match
?:	C conditional value operator
^	Exponentiation
++ --	Variable increment/decrement
	Note the absence of the C bit operators &, |, << and >>

Elementary awk programming
Built-in functions

- **substr(s,p,n)** The substring of s starting at p and continuing for n characters; if n is omitted, the rest of the string
- **index(s1,s2)** The first location of s2 within s1; 0 if not found
- **length(e)** The length of e, converted to character string if necessary, in bytes

Elementary awk programming
Built-in functions

• **sin, cos, tan**	Standard C trig functions
• **atan2(x,y)**	Standard signed arctangent function
• **exp, log**	Standard C exponential functions
• **srand(s), rand()**	Random number seed and access functions

Elementary awk programming
Examples and uses

- `length($0)>72` — print all of the lines whose length exceeds 72 bytes
- `{$2="";print}` — remove the second field from each line
- `{print $2}` — print only the second field of each line

Elementary awk programming
Examples and uses

- **`/Ucast/{print $1 "=" $NF}`**
- -- for each line which contains the string 'Ucast' print the first variable, an equal sign and the last variable (awk code to create awk code; a common technique)

- **`BEGIN{FS="/"};NF<4`**
- -- using '/' as a field separator, print only those records with less than four fields; when applied to the output of du, gives a two level summary

Elementary awk programming
Examples and uses

- **`{n++;t+=$4;print};END{print n "  " t}`**
- -- when applied to the output of an ls -l command provides a count and total size of the listed files; I use it as part of an alias for dir

- **`$0==prv{ct++;next};{printf("%8d %s", \ ct, prv);ct=1;prv=$0}`**
- -- prints each unique record with a count of the number of occurrences of it; presumes input is sorted

Advanced awk programming
Program structure

- **if(boolean) statement1 else statement2**
- -- if the boolean expression evaluates to true execute statement1, otherwise execute statement 2

- **for(v=init;boolean;v change) statement**
- -- Standard C for loop, assigns v the value of init then while the boolean expression is true executes the statement followed by the v change

Advanced awk programming
Program structure

- **for(v in array) statement**
- -- Assigns to v each of the values of the subscripts of array, not in any particular order, then executes statement

- **while(boolean) statement**
- -- While the boolean expression is true, execute the statement

Advanced awk programming
Program structure

- **do statement while(boolean) -- execute statement, evaluate the boolean expression and if true, repeat**
- **statement** may be either a simple statement or a series of statements separated by ; or \n, enclosed in {}, again like C
- **break** -- exit from an enclosing for or while loop

Advanced awk programming
Program structure

- **continue** -- restart the enclosing for or while loop from the top
- **next** -- stop processing the current record, read the next record and begin processing at 1st command
- **exit** -- terminate all input processing and, if present, execute the END command

Advanced awk programming
Arrays

- awk has two types of arrays - standard and general
- Standard arrays take the usual integer subscripts, starting at 0 and going up; multidimensional arrays are allowed and behave as expected
- General arrays take any type of variable(s) as subscripts, but the subscript(s) are treated as one long string expression.

Advanced awk programming
Arrays

- Subscripts are enclosed in [], not ()
- The use of for(a in x) on a generalized array will return all of the valid subscripts in some order, not necessarily the one you wished.

Advanced awk programming
Arrays

- The subscript separator is called SUBSEP and has a default value of comma (,)
- Elements can be deleted from an array via the **delete(array[subscript])** statement

Advanced awk programming
Built-in variables

- **FILENAME** The name of the file currently being processed
- **OFS** Output Field Separator; default ' '
- **RS** Input Record Separator; default \n
- **ORS** Output Record Separator; default \n
- **FNR** Current line's number with respect to the current file

Advanced awk programming
Built-in variables

- **OFMT** Output format for printed numbers; default %.6g
- **RSTART** The location of the data matched using the match built-in function
- **RLENGTH** The length of the data matched using the match built-in function

Advanced awk programming
Built-in functions

- **gsub(re,sub,str)** replace, in str, each occurrence of the regular expression re with sub; return the number of substitutions performed

- **int(expr)** return the value of expr with all fractional parts removed

Advanced awk programming
Built-in functions

- **match(str,re)** return the location in str where the regular expression re occurs and set RSTART and RLENGTH; if re is not found return 0

- **split(str,array,sep)** split str into pieces using sep as the separator and assign the pieces in order to the elements from 1 up of array; use FS if sep is not given

Advanced awk programming
Built-in functions

- **sprintf(format,expr,...)**
- -- write the expressions as the format indicates into a string and return it

- **sub(re,sub,str)**
- -- replace, in str, the first of the regular expression re with sub; return 1 if successful, 0 otherwise

Advanced awk programming
Built-in functions

- **system(command)**

  -- pass command to the local operating system to execute and return the exit status code returned by the operating system
- **tolower(str)**

  -- return a string similar to str with all capital letters changed to lower case

- Several of these functions/variables are not available in version 1 of awk

Advanced awk programming
Other file I/O

- **print and printf** may have **>** (or **>>**) **filename** or **| command** appended and the output will be sent to the named file or command; once a file is opened, it remains open until explicitly closed
- **getline var < filename** will read the next line from filename into var. Again, once a file is opened, it remains so until it is explicitly closed
- **close(filename)** explicitly closes the file named

Advanced awk programming
Writing your own functions

- A function begins with a function header of the form:

  function name(argument(s), localvar(s)) {

  and ends with the matching **}**

- The value of the function is returned via a statement of the form:

  return value

Advanced awk programming
Writing your own functions

- Functions do not have to return a value and the value returned by a function (either built-in or local) may be ignored by just placing the function and its arguments as a separate statement

- The local variables indicated in the localvars of the heading replace the global variables of the same name until the function completes, at which time the globals are restored

Advanced awk programming
Writing your own functions

- Functions may have side effects such as updating global variables, doing I/O or running other functions with side effects; beware the frumious bandersnatch

Advanced awk programming
Examples and uses


```
{ split($1,t,":")
  $1 = (t[1]*60+t[2])*60+t[3]
  print
  }
```

- Replaces an HH:MM:SS time stamp in the first field with a seconds since midnight value which can be more easily plotted, computed with, etc.

Advanced awk programming
Examples and uses


```
NR=1 {     t0=$1; tp = $1;
           for(i=1;i<=nv;i++) dp[i] = $(i+1);
           next
      }
      { dt=$1-tp; tp = $1
        printf("%d ",$1-t0)
        for(i=1;i<=nv;i++) {
           printf("%d ",($(i+1)-dp[i])/dt)
           dp[i] = $(i+1)
         }
         printf("\n")b
      }
```

Advanced awk programming
Examples and uses

- Take a set of time stamped data and convert the data from absolute time and counts to relative time and average counts. The data is presumed to be all amenable to treatment as integers. If not, formats better than %d must be used. The use of the undefined variable nv allows the same routine to be used with files with different numbers of data per time period, set by an nv=xxx in the awk shell statement.

Advanced awk programming
Examples and uses


```
BEGIN{     printf("set term postscript\n\
           set output '|lpr -Php'\n"} > plots
      {    if(system("test -s " $1 ".r") {
                 print "process1 " $1 ".r " $2
                 print "plot '" $1 ".data' \
                       using 2:5 title '" $3 "'" \
                       >> "plots"
           }
      }
END  { print "gnuplot < plots" }
```

Advanced awk programming
Examples and uses

- Write a pair of set lines to a file called plots. For each input line, if a file whose name is the first field on the line with a .r appended exists, write a command to the stdout file containing the file name and the second field from the line; also write a plot statement to a file called plots using the third field from the input line. Af ter the file has been processed, add a gnuplot command to the stdout file. If all of the output is passed to sh or csh through a pipe, the commands will be executed.

Advanced awk programming
Examples and uses


```
BEGIN   { l[1]=25; l[2]=20; l[3]=50 }
/^[ABC]/{ i = index("ABC",substr($0,1,1))
                a=$0 "                          "
                print substr(a,1,l[i])
                next }
          { print }
```

Make lines whose first characters are 'A', 'B', or 'C' have lengths of 25, 20, and 50 bytes respectively, Change no other lines.

Advanced awk programming
Examples and uses


```
/^\+/    { hold=hold "\r" substr($0,2); next}
                  { if( unfirst ) print hold
                    hold =""
                  }
/^1/              { hold = "\f" }
/^0/              { hold = "\n" }
/^-/              { hold = "\n\n" }
                  { unfirst = 1
                           hold = hold + substr($0,2)
                  }
END               { if(unfirst) print hold }
```

Advanced awk programming
Examples and uses

- Convert FORTRAN-type output with leading ANSI carriage control to a file with ASCII printer control

Advanced awk programming
Examples and uses


```
BEGIN   {       b=""; if(ll==0) ll=72 }
NF==0   {       print b; b=""; print ""; next }
                {       if(substr(b,length(b),1)=="-")
                        b=substr(b,1,length(b)-1) $0
                        else
                                b=b " " $0
                        while(length(b)>ll) {
                                i = ll
                                while(substr(b,i,1)!=" ") i--
                                print substr(b,1,i-1)
                                b = substr(b,i+1)
                          }
                }
END     { print b; print "" }
```

Advanced awk programming
Examples and uses

- This will take an arbitrary stream of text (where paragraphs are indicated by consecutive \n) and make all the lines approximately the same length. The default output line length is 72, but it may be set via a parameter on the awk command line. Both long and short lines are taken care of but extra spaces/tabs within the text are not correctly handled.

Advanced awk programming
Examples and uses


```
BEGIN {         FS = "\t"   # tab is field sep
                        printf("%10s %6s %5s   %s\n\n",
                                "COUNTRY", "AREA", "POP",
                                        "CONTINENT")
                }
                {       printf("%10s %6d %5d   %s\n", $1,
                                $2, $3, $4)
                        area = area +$2
                        pop = pop + $3
                }
END             { printf("\n%10s %6d %5d\n", "TOTAL",
                        area, pop) }
```

Advanced awk programming
Examples and uses

- This will take a variable width table of data with four tab separated fields and print it as a fixed length table with headings and totals.

Advanced awk programming
Examples and uses


```
BEGIN     {FS="/“
          q2=sprintf("%c",34)
          print "<html>"
          print "<head>"
          print "<title>Stats</title>"
          print "</head>"
          print "<body bgcolor=#FFFFFF>"}
          {print "<img src=" q2 $3 "/load.gif" q2 ">"}
END       {print "</body>"
          print "</html>"}
```

Advanced awk programming
Examples and uses

- This is an example of how to write a double quote from within an AWK program. The program reads data from a file and produces a web page containing this data.

Important things which will bite you

- $1 inside the awk script is **not** $1 of the shell script
- Actions are within {}, **not** selections
- Every selection **is** applied to each input line **after** the previously selected actions have occurred

Questions?

Session 561[67]