Operating System Unix

Course VI
Grep, Sed & AWK

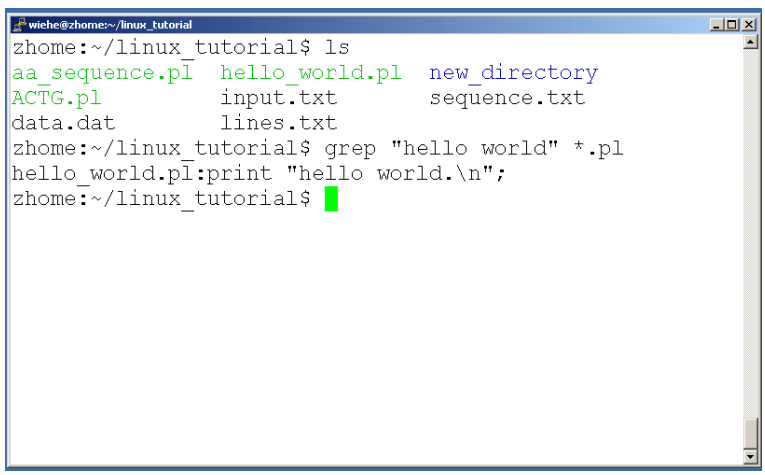
Osman SALEM
Maître de conférences - HDR
osman.salem@u-paris.fr

Université de Paris

1

grep

- To search files in a directory for a specific string use "grep"



2

grep

- `grep`, in its simplest form, is a program that displays lines of text from its input that contain a certain pattern
- Put another way, it removes – or *filters out* – lines of text from its input that do *not* contain the pattern (`grep` is thus a filter in every sense of the word)
- Its usage is as follows:

```
grep pattern [files]
```

- For example:

```
grep sincerely *.txt
```

3

grep

- Global Regular Expression Print
- Finds patterns within a file
- Syntax: grep [-cilnv] pattern file…
  - -c returns count of lines matching pattern
  - -i ignores case
  - -l only prints names of files with matches
  - -n precedes each matching line with its line #
  - -v prints all lines except those matching pattern
  - -r recursive research in subdirectories
  - -A x: displays X lines after
  - -B x: displays X lines before
  - -w word: search for a complete word

4

Regular Expressions

- $ grep alias .bashrc
- $grep -i alias .bashrc
- $grep -n alias .bashrc
- $grep -v alias .bashrc
- $grep -r "alias" code/ (recursive search)

- Example :
  - $grep -n sal /etc/passwd ---> search for substring **sal**, and show the line nubmer (option -n)
  - $grep -nw sal /etc/passwd ---> option -w search for word entier, and not a substring

5

grep

Table 6-2

Search Pattern	Description
?	Any single character (except /)
*	Any string length, including zero characters (except . at the beginning of a file name and /)
[a-z]	Any of the characters enclosed (here: lowercase letters from a to z)
[a-ek-s]	Any character from the ranges a–e and k–s
[abcdefg]	Any of these characters
[!abc]	None of these characters

6

Regular Expressions

- `grep -E` stands for "global regular expression parser"
- A *regular expression* is a term used to describe a set of special text-matching patterns, for example:
  - `^abc` matches any line of text that *begins* with `abc`
  - `abc$` matches any line of text that *ends* with `abc`
  - `^$` matches a blank line
  - `a*` matches any sequence of zero or more `a`'s
  - `a+` matches any sequence of one or more `a`'s
  - `c[aou]t` matches `cat`, `cot` or `cut`
  - `c.t` matches a `c`, followed by *any one character*, followed by a `t`
  - `X[a-zA-Z0-9]*X` matches any sequence (even zero-length) of letters or digits surrounded by a pair of `X`'s
  - and many more – check the `man` page for `grep` or `regexp`

7

Regular Expressions

- grep -E Alias .bashrc
- grep -E ^alias .bashrc
- grep -E [Aa]lias .bashrc
- grep -E [0-4] .bashrc
- grep -E [a-zA-Z] .bashrc
- grep "^.o" fichier
- grep "^t" /etc/passwd
- grep -v "^t" /etc/passwd
- grep "T.t." /etc/passwd
- less /etc/group | grep "^[a-j]"
- ls -l /etc | grep "^d"

8

sed

- `sed` (short for "stream editor") is a program for performing basic editing tasks on the output of another program or on a file (similarly to `sort`, the file itself is not changed)
- The most basic form of `sed` is as follows:

```
sed action [files]
```

- `sed` can perform several actions at a time, as follows:
  - `sed -e action1 -e action2 [files]`
  - `sed -f scriptfile [files]`
- Actions specified on the command line are almost always enclosed in single quotes to prevent shell interpretation of special characters

9

sed

```
y ~> cat -n example
 1  This is the first line of an example text.
 2  It is a text with erors.
 3  Lots of erors.
 4  So much erors, all these erors are making me sick.
 5  This is a line not containing any errors.
 6  This is the last line.
```

```
     ~> sed  '/erors/p' example
This is the first line of an example text.
It is a text with erors.
It is a text with erors.
Lots of erors.
Lots of erors.
So much erors, all these erors are making me sick.
So much erors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.
```

```
sed  -n '/^[2-4]/p' supplies
```

```
     ~> sed -n '/erors/p' example
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.
```

10

10

sed

```
~> sed -n '/^This.*errors.$/p' example
This is a line not containing any errors.
```

```
~> sed '/erors/d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
```

```
~> sed '2,4d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
```

```
~> sed '3,$d' example
This is the first line of an example text.
It is a text with erors.
```

11

11

sed

- The most common "action" is text:
  - `s/foo/bar/` – change the first occurrence of `foo` on each line to `bar`
  - `s/foo/bar/g` – change *all* occurrences of `foo` to `bar`
- A range of line numbers can be specified to restrict these actions:
  - `1,10s/foo/bar/pgI` or `40,$s/foo/bar/`
- Note that `$` refers to the last line in the file
- Another common action is *deleting* lines:
  - `11,20d` – delete the second 10 lines of the input
  - `/hopscotch/d` – delete all lines with `hopscotch` in them

12

sed

- `sed -n '2p' file.txt (n:silent, p:print)`
- `sed -n '2,4p' file.txt`
- `sed -n '2,4!p' file.txt`
- `sed -n '/dog/p' file.txt`
- `sed -n '/dog/Ip' file.txt`
- `sed -n '/[0-9]/p' file.txt`
- `sed -n '/cat/,/dog/p' file.txt`
- `sed -n '/cat/,+2p' file.txt`
- `sed -n '/^cat/p' file.txt`
- `sed -n '/cat$/p' file.txt`
- `sed -n '/^cat$/p' file.txt`
- `sed -n '/^c…$/Ip' file.txt`
- `sed -n '/^c.\+$/Ip' file.txt`
- `sed -n '/^c.\+[0-9]$/Ip' file.txt`

13

13

sed

- `sed -n '/^$/d' file.txt`
- `sed '10,$d' somefile`
- `sed '1,10s/foo/bar/'` ou `sed '40,$s/foo/bar/'`
- `11,20d`: suppression de la ligne 11 à 20
- `/yahoo/d`: suppression de lignes contenant le mot "yahoo"
- `3,$!d`: ne supprime pas les lignes de 3 jusqu'à la fin
- `/google/!d`: ne supprime pas les lignes contenant "google"
- `sed -n '/osman.*/p' /etc/passwd`
- `sed -n '/start/,/stop/p' /etc/passwd`

- ! : negation
  - `sed '/^#/!d' somefile`
  - `.`: any character
  - `?`: 0 or 1
  - `+`: 1 or many
  - `*`: 0 or many

14

14

sed

- `3,$!d`
- `/ducks/!d`
- `sed -ne '/hello/s/dog/cat/gp' file.txt`
- `sed -ne '/^!Hello/Is/dog/cat/Igp' file.txt`
- `sed -ne '/.*[0-9]\{1,3\}/gp' file.txt`

15

15

Understand the Stream Editor sed

- sed: stream editor that transforms text line-by-line
- Editing commands are single-character: **d** (delete), **s** (substitute), **p** (output line), **a** (append after)
  - Sed '5 a\ word' < file.txt

```
~> sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with erors.
> Lots of erors.
> So much erors, all these erors are making me sick.
> This is a line not containing any errors.
> This is the last line.
```

```
~> sed 's/$/EOL/' example
This is the first line of an example text.EOL
It is a text with erors.EOL
Lots of erors.EOL
So much erors, all these erors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL
```

16

Understand the Stream Editor sed

- Examples:
  - sed 's/:/ /' /etc/passwd
  - sed 's/:/ /g' /etc/passwd

```
sandy ~> sed -e 's/erors/errors/g' -e 's/last/final/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.
```

```
s/vi/emacs/g
/[Ww]indows/d
```

17

Understand the Stream Editor sed

- To delete everything from line 10 to the end and print the first 9 lines of the file somefile
  - sed '10,$d' somefile
- Regular expressions are enclosed in / slashes
  - sed -n '/Murphy.*/p' somefile
- To perform several editing commands for the same address
  - sed '1,10{*command1* ; *command2*}'
- A leading ! negates editing commands
  - sed '/^#/!d' somefile

18

Understand the Stream Editor sed

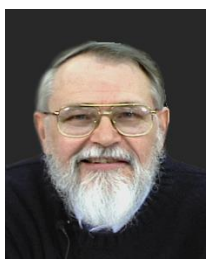
Table 6-8

Command	Example	Editing action
d	sed '10,$d' *file*	Delete line.
a	sed 'a\ *text*\ *text*' *file*	Append text after the specified line, with line breaks and backslashes included as shown in the example.
i	sed 'i\ *text*\ *text*' *file*	Insert *text* before the specified line.
c	sed '2000,$c*text*'*file*	Replace specified lines with the *text*.
s	sed s/*x*/*y*/*option*	Search and replace—the search pattern *x*' is replaced with pattern *y*'. The search and the replacement patterns are regular expressions in most cases, and the search and replace behavior can be influenced through various options.
y	sed y/*abc*/*yxz*/	(yank) Replace every character from the set of source characters with the character that has the same position in the set of destination characters.

19

Why is it called AWK?





Aho *Weinberger* *Kernighan*

AWK

Alfred V. Aho, Peter J. Weinberg, Brian W. Kernighan

...a powerful programming language disguised as a utility

20

Understand the Text Manipulator awk

- **awk** got its name from its developers Alfred V. **A**ho, Peter J. **W**einberger, and Brian W. **K**ernighan
- An awk procedure is an indefinite loop
- Exiting the loop is possible in two cases:
  - There are no more data at input
  - You exit the loop deliberately
- awk is very similar to a stream editor, but you can also define such things as variables, functions, and loops

21

awk programming model

- **Input**: awk views an input stream as a collection of **records**, each of which can be further subdivided into **fields.**
  - Normally, a record is a line, and a field is a word of one or more nonwhite space characters.
  - For each pattern that matches input, action is executed; all patterns are examined for every input record

    pattern { action } *##*Run action if pattern matches
  - Either part of a pattern/action pair may be omitted.
    - If pattern is omitted, action is applied to every input record

      { action } *##*Run action for every record
    - If action is omitted, default action is to print matching record on standard output

      pattern *##*Print record if pattern matches

22

awk

- `Awk` is a text-processing tool and programming language. It is capable of virtually any conceivable text processing
- As with `sed`, its simplest usage is as follows:

```
awk action [files]
```

- where `action` is a sequence of statements enclosed in `{ }`, each separated by a `;` For example:

```
who | awk '{print $1, "is on terminal", $2}'
```

- `$1`, `$2`, `$3`, etc are the tokens from each line of input. Tokens are separated by spaces and tabs

23

awk

- With the `-F` option, it is possible to specify the character(s) used to separate tokens:

```
awk -F : '{print $1, "home:", $6}' /etc/passwd
```

- It is possible to perform different actions on lines that match certain (regular expression) patterns:

```
awk '/Australia/ {print $1}' database
```

/zhang/ {print $3}
NR<10 {print $0}

- It is possible to perform arithmetic on variables within `awk`, for example:

```
awk '{print $1,($3+$4)/$5}' database
```

24

awk

```
$ awk -F ':' '{ print $1 }' /etc/passwd
$ awk -F ':' '{ if ($3 >= 500) print $1}'
$ awk -F ':' '$3 >= 500 && $3 < 1000 \
          { print $1 }' /etc/passwd
```

- The **–F** option specifies the **input field separator**
- The **–f** option **names** the script **file**
  - awk –f scriptFile.awk input-file
  - awk –F: -f scriptFile.awk input-file

25

Action

- Enclosed by braces
- Statements: separated by newline or ;
  - Assignment statement
    line=1
    sum=sum+value
  - print statement
    print "sum= ", sum
  - if statement, if/else statement
  - while loop, do/while loop, for loop (three parts, and one part)
  - break, continue

26

Awk pattern

- Pattern: a condition that specify what kind of records the associated action should be applied to
  - string and/or numeric expressions: If evaluated to nonzero (true) for current input record, associated action is carried out.
  - Or an regular expression (ERE): to match input record, same as *$0 ~ /regexp/*

NF == 0 Select empty records

NF > 3 Select records with more than 3 fields

NR < 5 Select records 1 through 4

(NR == 3) && (FILENAME ~ /[.][ch]$/) Select record 3 in C source files

$1 ~ /jones/ Select records with "jones" in field 1

/[Xx][Mm][Ll]/ Select records containing "XML", ignoring lettercase

$0 ~ /[Xx][Mm][Ll]/ Same as preceding selection

28

Output (continued)

- **NF**, the Number of Fields
  - Any valid expression can be used after a **$** to indicate the contents of a particular field
  - One built-in expression is **NF**, or Number of Fields
  - `{ print NF, $1, $NF }` will print the number of fields, the first field, and the last field in the current record
  - `{ print $(NF-2) }` prints the third to last field
- Computing and Printing
  - You can also do computations on the field values and include the results in your output
  - `{ print $1, $2 * $3 }`

29

Output (continued)

- Printing Line Numbers
  - The built-in variable NR can be used to print line numbers
  - `{ print NR, $0 }` will print each line prefixed with its line number
- Putting Text in the Output
  - You can also add other text to the output besides what is in the current record
  - `{ print "total pay for", $1, "is", $2 * $3 }`
  - Note that the inserted text needs to be surrounded by double quotes

30

Selection

- Awk patterns are good for selecting specific lines from the input for further processing
  - Selection by Comparison
    - `$2 >= 5 { print $0 }`
  - Selection by Computation
    - `$2 * $3 > 50 { printf("%6.2f for %s\n", $2 * $3, $1) }`
  - Selection by Text Content
    - `$1 == "NYU"` ⇔ `NYU 12 13`
    - `$1 ~ /NYU/` ⇔ `AbNYUbcde 120 15`
  - Combinations of Patterns
    - `$2 >= 4 || $3 >= 20`
  - Selection by Line Number
    - `NR >= 10 && NR <= 20`

31

BEGIN, END pattern

```
BEGIN    {Begin's Actions}

         Pattern   {statement}

         Pattern   {statement1;  statement 2;  statement3}

         Pattern   {statement1
                    statement2
                    statement3
                   }

END      {End's Actions}
```

32

BEGIN, END pattern

- BEGIN pattern: is performed just once, *before any command-line* is processed,
  - normally used to handle special initialization tasks
- END pattern: is performed just once, *after all of input data has been processed.*
  - normally used to produce summary reports or to perform cleanup actions

```
BEGIN    {Begin's Actions}

         Pattern   {Action}

         Pattern   {Action}

         Pattern   {Action}

END      {End's Actions}
```

33

Computing with AWK

- Counting is easy to do with Awk

```
$3 > 15 { emp = emp + 1}
END { print emp, "employees worked
      more than 15 hrs"}
```

- Computing Sums and Averages is also simple

```
{ pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay/NR
    }
```

34

awk -f total.awk total.dat

total.awk

```
# Begin Processing
BEGIN {print "Print Totals"}

#Body Processing
{total = $1 + $2 + $3}
{print $1 " + " $2 " + " $3 " = " total}

#End Processing
END {print "End Totals"}
```

total.dat

Input:

```
22 78 44
66 31 70
52 30 44
88 31 66
```

Output:

```
Print Totals
22 + 78 + 44 = 144
66 + 31 + 70 = 167
52 + 30 + 44 = 126
88 + 31 + 66 = 185
End Totals
```

35

awk

- Complex sets of actions may be put in a separate script file and executed thus:

```
awk –f scriptfile inputfile
```

Demo:
$ average.awk avg.data

```
#!/bin/awk –f

BEGIN{
  lines=0;
  total=0;
}
{
  lines++;
  total+=$1;
}
```

```
END{
  if (lines>0)
    print "agerage is ", total/lines;
  else
    print "no records"
}
```

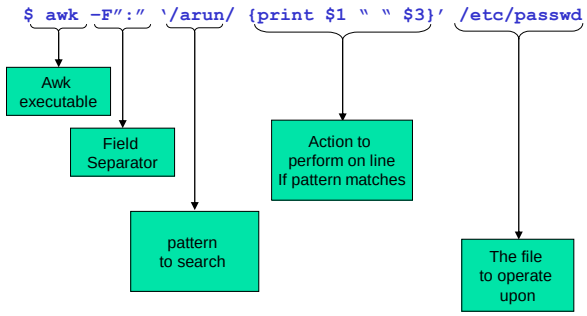
36

Table 9-3. Commonly used built-in scalar variables in awk

Variable	Description
FILENAME	Name of the current input file
FNR	Record number in the current input file
FS	Field separator (regular expression) (default: " ")
NF	Number of fields in current record
NR	Record number in the job
OFS	Output field separator (default: " ")
ORS	Output record separator (default: "\n")
RS	Input record separator (regular expression in gawk and mawk only) (default: "\n")
$0	the current record
$1, $2, … $NF	the first, second, … last field of current record

37

A simple example (cont..)



38

awk: Built-in Functions

- Counting lines, words, and characters using length (a poor man's wc):

```
{
    nc = nc + length($0) + 1
    nw = nw + NF
}
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

- substr(s, m, n) produces the substring of s that begins at position m and is at most n characters long.

39

A simple example (cont..)

- The output of the above command will be

```
[root@tux root]# awk -F":" '/arun/ {print $1 " " $3}' /etc/passwd
 arun 504
[root@tux root]#
```

- Another way to write the command is

```
[root@tux root]# awk 'BEGIN { FS=":" } /arun/ {print $1 " " $3}'
/etc/passwd
 arun 504
[root@tux root]#
```

```
$ awk -f source-file input-file1 input-file2 ...
```

40

Running awk programs (cont.)

- Executable Scripts: Making self-contained awk programs.

(eg) : Write a script named `hello` with the following contents

```
#! /usr/bin/awk -f
# a sample awk program
/foo/ { print $1}
```

Execute the following command

```
$ chmod +x hello
```

To run this script simply type

```
$ ./hello file.txt
```

41

awk (cont.)

- awk *'BEGIN { OFS=";" ; ORS="|n-->|n" }*
  { print $1,$2}' test

```
kelly@octarine ~/test> cat processed.awk
BEGIN { OFS="-" ; ORS="\n--> done\n" }
{ print "Record number " NR ":\t" $1,$2 }
END { print "Number of records processed: " NR }

kelly@octarine ~/test> awk -f processed.awk test
Record number 1:        record1-data1
--> done
Record number 2:        record2-data2
--> done
Number of records processed: 2
--> done
```

```
kelly@octarine ~> cat revenues
20021009        20021013        consultancy     BigComp         2500
20021015        20021020        training        EduComp         2000
20021112        20021123        appdev          SmartComp       10000
20021204        20021215        training        EduComp         5000

kelly@octarine ~> cat total.awk
{ total=total + $5 }
{ print "Send bill for " $5 " dollar to " $4 }
END { print "---------------------------------\nTotal revenue: " total }

kelly@octarine ~> awk -f total.awk test
Send bill for 2500 dollar to BigComp
Send bill for 2000 dollar to EduComp
Send bill for 10000 dollar to SmartComp
Send bill for 5000 dollar to EduComp
---------------------------------
Total revenue: 19500
```

42

awk (cont.)

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
            { print $1, $2 }'  file1.txt
```

- Consider that we have the following input in a file called grades

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

- The following awk script grades.awk will find the average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
$ awk -f grades.awk grades
```

43

Simple one-line awk program

- Using awk to cut
  - awk -F ':' '{print $1,$3;}' /etc/passwd
- To simulate head
  - awk 'NR<10 {print $0}' /etc/passwd
- To count lines:
  - awk 'END {print NR}' /etc/passwd
- What's my UID (numerical user id?)
  - awk –F ':' '/^root/ {print $3}' /etc/passswd

44

Doing something new

- Output the logarithm of numbers in first field
  - echo 10 | awk '{print $0,log($0)}'
- Sum all fields together
  - awk '{sum=0; for (i=1;i<NF;i++) sum+=$i; print sum}' data2.txt
- How about weighted sum?
  - Four fields with weight assignments (0.1, 0.3, 0.4,0.2)
  - awk '{sum= $1*0.1+$2*0.3+$3*0.4+$4*0.2; print sum}' data2.txt

45

Awk variables

- Difference from C/C++ variables
  - Initialized to 0, or empty string
  - No need to declare, variable types are decided based on context
    - All variables are global (even those used in function, except function parameters)
- Difference from shell variables:
  - Reference without $, except for $0,$1,…$NF
- Conversion between numeric value and string value
  - N=123; S="N" ## s is assigned "123"
  - S=123, N=0+S ## N is assigned 123
- Floating point arithmetic operations
  - awk '{print $1 "F=" ($1-32)*5/9 "C"}' data
  - echo 38 | awk '{print $1 "F=" ($1-32)*5/9 "C"}'

46

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

```
% awk '$3 * $4 > 500 {print $0}' file
```

47

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
< =	Less than or equal	x < = y
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
> =	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

48

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

```
% awk '($2 > 5) && ($2 <= 15)
                        {print $0}' file
% awk '$3 == 100 || $4 > 50' file
```

49

Awk functions

Table 9-6. Elementary numeric functions

Function	Description
`atan2(y, x)`	Return the arctangent of y/x as a value in $-\pi$ to $+\pi$.
`cos(x)`	Return the cosine of x (measured in *radians*) as a value in -1 to $+1$.
`exp(x)`	Return the exponential of x, ex.
`int(x)`	Return the integer part of x, truncating toward zero.
`log(x)`	Return the natural logarithm of x.
`rand( )`	Return a uniformly distributed pseudorandom number, r, such that $0 \leq r < 1$.
`sin(x)`	Return the sine of x (measured in *radians*) as a value in -1 to $+1$.
`sqrt(x)`	Return the square root of x.
`srand(x)`	Set the pseudorandom-number generator seed to x, and return the current seed. If x is omitte the current time in seconds, relative to the system epoch. If `srand( )` is not called, awk starts the same default seed on each run; `mawk` does not.

50

Working with strings

- length(a): return the length of a stirng
- substr (a, start, len): returns a copy of sub-string of len, starting at start-th character in a
  - substr("abcde", 2, 3) returns "bcd"
- toupper(a), tolower(a): lettercase conversion
- index(a,find): returns starting position of *find* in *a*
  - *Index("abcde", "cd") returns 3*
- match(a,regexp): matches string a against regular express regexp, return index if matching succeeed, otherwise return 0
  - Similar to (a ~ regexp): return 1 or 0

51

Awk (cont.)

- And of course, it is possible to create standalone `awk` scripts, as follows:

```
#!/bin/awk –f
{print $1, "home:", $6}
```

and then run it as follows:

```
myawkscript /etc/passwd
```

- `awk` offers a more powerful print facility, known as `printf` (similar to that used in C and C++):

```
awk 'printf("%-12s%-20s\n", $1, $6)' database
awk '{printf("%2d %-12s $%9.2f\n", $1, $2, $3)}'
```

- Note the need for the `\n` on the end of the format string

52

sprintf

```
NR == 1 {
        str = sprintf("%2d  %-12s   $%9.2f\n", $1, $2, $3)
        len = length (str)
        print len "  "  str
        }
```

Input:	Output:
1 clothing 3141 2 computers 9161 3 textbook 21312	27 1 clothing $ 3141.00

53

General Structure of an awk Program (continued)

- A semicolon must be at the end of each command
- A pattern can be a regular expression or a comparison of variables
- The commands executed before and after the main program are marked by **BEGIN** and **END**

```
BEGIN   {
        print "Counting lines";
        number=0;
        }

{ number++; }

END   { print "Result: " number; }
```

54

Start an awk Program from cmd Line

- awk '*program*' *file* [*file*]

```
tux@da10> awk 'BEGIN {FS = ":"} {print NR,$1}' /etc/passwd
1 root
2 bin
3 daemon
4 lp
5 mail
...
```

- You can also pipe the output of another command to awk: ***command*** **| awk '*program*'**

```
tux@da10> date | awk '{print "Today is " $2". the "$3".,",$6}'
Today is Jan. the 31., 2007
```

55

Control Flow Statements

- **awk** provides several control flow statements for making decisions and writing loops
- If-Then-Else

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }

END { if (n > 0)
        print n, "employees, total pay is",
              pay, "average pay is", pay/n
      else
        print "no employees are paid more
              than $6/hour"
    }
```

56

if Statement

Syntax:

```
if (conditional expression)
  statement-1
else
  statement-2
```

Example:

```
if ( NR < 3 )
  print $2
else
  print $3
```

57

Loop Control

- While

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year
{  i = 1
  while (i <= $3) {
         printf("\t%.2f\n", $1 * (1 + $2) ^ i)
         i = i + 1
  }
}
```

58

Do-While Loops

- Do While

```
do {
    statement1
    }
while (expression)
```

59

while Loop

Syntax:

```
while (logical expression)
        statement
```

Example:

```
i = 1
while (i <= NF)
{
        print i, $i
        i++
}
```

60

do-while Loop

Syntax:

```
do
     statement
while (condition)
```

- statement is executed at least once, even if condition is false at the beginning

Example:

```
i = 1
do {
  print $0
  i++
} while (i <= 10)
```

61

For statements

- For

```
# interest2 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year

{ for (i = 1; i <= $3; i = i + 1)
   printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

62

for Loop

Syntax:

```
for (initialization; limit-test; update)
        statement
```

Example:

```
for (i = 1; i <= NR; i++)
{
        total += $i
        count++
}
```

63

Arrays

- Array elements are not declared
- Array subscripts can have ***any*** value:
  - Numbers
  - Strings! (***associative arrays***)
- Examples
  - `arr[3]="value"`
  - `grade["Korn"]=40.3`

64

Array Example

```
# reverse - print input in reverse order by line

{ line[NR] = $0 }     # remember each line

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

- Use `for` loop to read associative array
  - `for (v in array) { … }`
  - Assigns to v each subscript of array (unordered)
  - Element is `array[v]`

65

for Loop for arrays

Syntax:

```
for (var in array)
        statement
```

Example:

```
for (x in deptSales)
{
      print x, deptSales[x]
}
```

66

Useful One (or so)-liners

- `END { print NR }`
- `NR == 10`
- `{ print $NF }`
-
```
{ field = $NF }
END { print field }
```
- `NF > 4`
- `$NF > 4`
-
```
{ nf = nf + NF }
END { print nf }
```

67

More One-liners

- `/Jeff/ { nlines = nlines + 1 }`
  `END { print nlines }`
- `$1 > max { max = $1; maxline = $0 }`
  `END { print max, maxline }`
- `NF > 0`
- `length($0) > 80`
- `{ print NF, $0}`
- `{ print $2, $1 }`
- `{ temp = $1; $1 = $2; $2 = temp; print }`
- `{ $2 = ""; print }`

68

Even More One-liners

- ```
  { for (i = NF; i > 0; i = i - 1)
        printf("%s ", $i)
    printf("\n")
  }
  ```
- ```
  { sum = 0
    for (i = 1; i <= NF; i = i + 1)
        sum = sum + $i
    print sum
   }
  ```
- ```
  { for (i = 1; i <= NF; i = i + 1)
        sum = sum $i }
    END { print sum }
  }
  ```

69
  ```

Awk Variables

- $0, $1, $2, $NF
- NR - Number of records processed
- NF - Number of fields in current record
- FILENAME - name of current input file
- FS - Field separator, space or TAB by default
- OFS - Output field separator, space by default
- ARGC/ARGV - Argument Count, Argument Value array
  - Used to get arguments from the command line

70

Operators

- = assignment operator; sets a variable equal to a value or string
- == equality operator; returns TRUE is both sides are equal
- != inverse equality operator
- && logical AND
- || logical OR
- ! logical NOT
- <, >, <=, >= relational operators
- +, –, /, *, %, ^
- String concatenation

71

Builtin functions

```
tolower(string)
toupper(string)
```

72

Arrays in awk

Syntax:

```
arrayName[index] = value
```

Examples:

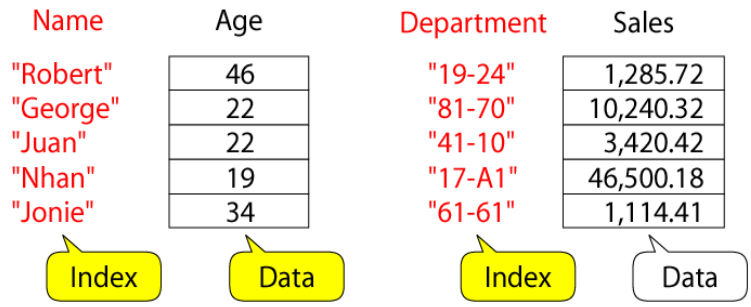
```
list[1] = "one"
list[2] = "three"

list["other"] = "oh my !"
```

73

Illustration: Associative Arrays

- awk arrays can use string as index



74

Awk builtin split function

`split(string, array, fieldsep)`

- divides string into pieces separated by fieldsep, and stores the pieces in array
- if the fieldsep is omitted, the value of FS is used.

Example:

```
split("auto-da-fe", a, "-")
```

- sets the contents of the array a as follows:

```
a[1] = "auto"
a[2] = "da"
a[3] = "fe"
```

75