Awk

- The *awk* Unix utility can manipulate text files that are arranged in columns.
- Like most other Unix utilities, *awk* reads from standard input if no files are specified.
- General forms:

```
awk [-Fc] 'script' [files]
awk [-Fc] -f scriptfile [files]
```

- -F*c* indicates a field separator to be the specified character *c*.
- For instance, one could specify -F: to indicate that ':' is the field separator.
- By default the field separator is blanks and tabs.

Awk Scripts

- Each *awk* script consist of one or more pairs of:
  - *pattern { procedure }*
- *awk* processes each line of a file, one at a time.
- For each pattern that matches, then the corresponding procedure is performed.
- If no pattern is specified, then the procedure is applied to each line.
- If the *{ procedure }* is missing, then the matched lines are written to standard output.

Awk Patterns

- *awk* patterns can be any of the following.
  - */ regular expression /*
  - *relational expression*
  - *pattern-matching expression*
  - BEGIN
  - END

Awk Regular Expressions

- An *awk* pattern that uses a regular expression indicates that somewhere in the line the regular expression matches.
- The regular expression returns true or false depending on if it matches within the current line.
- Using a pattern only (causes each line to be printed that matches) results in similar functionality as grep.

```
/D[Rr]\./    # matches lines containing DR. or Dr.
/^#/         # matches lines begining with '#'
```

Awk Relational Expressions

- An *awk* relational expression can consist of strings, numbers, arithmetic/string operators, relational operators, defined variables, and predefined variables.
  - *$0* means the entire line
  - *$n* means the *n*th field in the current line
  - *length($n)* means the length in characters of the *n*th field in the current line
  - *NF* is a predefined variable indicating the number of fields in the current line
  - *NR* is the number of the current line

Awk Relational Expression Examples

```
$1 == "if"
   # Is first field an "if"?

$1 == $2
   # Are the first two fields the same?

NR > 100
   # Have we already processed 100 lines?

NF > 5
   # Does the current line have more than 5 fields?

NF > 5 && $1 == $2
   # Compound condition.

/if/ && /</
   # Does the line contain "if" and "<"?

/while {/ || /do {/
   # Does the line contain "while {" or "do {"?
```


Awk Pattern-Matching Expressions

- Pattern matching expressions can check if a regular expression matches (~) or does not match (!~) a field.
- Note that the pattern does not have to match the entire field.

```
# Does first field match "DR." or "Dr."?
$1 ~ /D[Rr]\./

# Does first field not match "#"?
$1 !~ /#/
```


Awk BEGIN and END Patterns

- The BEGIN pattern lets you specify a procedure before the first input line is processed.
- The END pattern lets you specify a procedure after the last input line is processed.

Awk Procedures

- An *awk* procedure specifies the processing to occur when a line matches the specified pattern.
- An *awk* procedure is contained within the '{' '}' and consists of commands separated by semicolons or newlines.
- Commonly used *awk* commands include:

```
# print arguments, if none print $0
print [args]

# assignment to variables
var = expr
```


Awk Control Structures

- An *awk* procedure can also contain control structures.
- if control structure form

```
if (condition) {commands} [else {commands}]
```

- while control structure form

```
while (condition) {commands}
```

- for control structure form

```
for (init; condition; increment) {commands}
```


Example Awk Scripts

```
# print the first two fields of each line
{ print $1, $2 }

# printf uses a C-like format string to print values
{ printf "%-10s %-8s\n", $1, $2 }

# prints lines with one or more characters
$0 !~ /^$/

# print 2nd field when its value is in the range 1..9
$2 > 0 && $2 < 10
{ print $2 }

# print sum of values of first field in each line
BEGIN {sum=0}
{sum += $1}
END {print sum}
```


Example Awk Scripts (cont.)

```
# print the maximum of the first two fields in each line
{ if ($1 > $2) print $1; else print $2 }

# print the maximum of the fields in a line
{ max = $1
  i = 2
  while (i <= NF) {
     if (max < $i)
        max = $i
     i++
  }
  print max
}

# print the sum of the values of the fields in each line
{ sum=0
  for (i=1; i <= NF; i++)
     sum = sum + $i
  print sum
}
```

Tr

- The *tr* unix utility is a character translation filter.
- The most common usage is to replace one character for another.
- It reads from standard input and writes to standard output.
- General forms.

```
tr string1 string2
    # replace char in string1 with corresponding char in string2

tr -d string
    # delete all occurrences of chars that appear in string

tr -s string1 string2
    # replace repeated chars in string1 with single char in string2
```


Tr Examples

```
tr "[a-z]" "[A-Z]"
    # convert lowercase letters to uppercase letters

tr '&' '#'
    # convert &'s to #'s

tr -s "\t" "\t"
    # squeeze consecutive tabs to a single tab

tr -d '\015'
    # deletes carriage return chars from a DOS file
```


Basename

- The *basename* unix utility strips off the portion of a pathname that precedes and including the last slash and prints the result to standard output.

```
basename /home/faculty/whalley/cop4342exec/plot.p
    # would print "plot.p"
```


Dirname

- The *dirname* unix utility strips off the last slash and all the charcters that follow it in the pathname.

```
dirname /home/faculty/whalley/cop4342exec/plot.p
    # would print "/home/faculty/whalley/cop4342exec"
```

- Often shell scripts are written that will determine the directory in which they reside.

```
dir=`dirname $0`
if [ -r $dir/trace.dat ]
then
    ...
fi
```

Sort

- The *sort* unix utility divides each line into fields separated by a whitespace (blanks or tabs) character and sorts the lines by field, from left to right.
- It writes to standard output.
- The -b option causes leading blank characters for each field to be ignored.
- The -r option causes the sort to be in reverse order.
- The -k *start*[,*stop*] option indicates the position of the next field to start sorting and one can optionally specify the field to stop sorting.
- Field positions are numbered starting from 1.
- If no files are specified, then it sorts the standard input.
- If more than one file is specified, then it concatenates all the files together before sorting.

`sort [-r] [-k start,stop]* [files]`

Sort Examples

- Assume the file "data.txt" contains the following lines.

```
Smith Bob 27 32312
Jones Carol 34 32306
Miller Ted 27 32313
Smith Alice 34 32312
Miller Ted 45 32300
```

- "sort data.txt" would give priority to the lowest position fields when sorting by default and produce the following output.

```
Jones Carol 34 32306
Miller Ted 27 32313
Miller Ted 45 32300
Smith Alice 34 32312
Smith Bob 27 32312
```


Sort Examples (cont.)

- Sorting on a particular field will cause that field to have the highest priority. If there are ties, then it uses the remaining fields after the specified field in the line and afterwards wraps around.
- "sort -k 3 data.txt" would produce:

```
Smith Bob 27 32312
Miller Ted 27 32313
Jones Carol 34 32306
Smith Alice 34 32312
Miller Ted 45 32300
```

- "sort -k 3,3 -k 1 data.txt" would produce:

```
Miller Ted 27 32313
Smith Bob 27 32312
Jones Carol 34 32306
Smith Alice 34 32312
Miller Ted 45 32300
```


Sort Examples (cont.)

- "sort -k 3 data.txt" would produce:

```
Miller Ted 45 32300
Jones Carol 34 32306
Smith Alice 34 32312
Smith Bob 27 32312
Miller Ted 27 32313
```

- "sort -k 2,2 -k 1,1 -k 4 data.txt" would produce:

```
Smith Alice 34 32312
Smith Bob 27 32312
Jones Carol 34 32306
Miller Ted 45 32300
Miller Ted 27 32313
```

Sort Examples (cont.)

- Use the -r option to reverse the order of the sort.
- "sort -r < data.txt" would produce:

```
Smith Bob 27 32312
Smith Alice 34 32312
Miller Ted 45 32300
Miller Ted 27 32313
Jones Carol 34 32306
```


Type of Sort

- By default, sort orders lines in alphabetical (actually ASCII) order.
- You can specify that a particular field is to be sorted in numerical order by including an 'n' after the field position.
- Assume the file "data2.txt" contains:

```
Smith 42
Jones 7
Miller 100
```

- "sort -k 2 data2.txt" would produce:

```
Miller 100
Smith 42
Jones 7
```

- "sort -k 2n data2.txt" would produce:

```
Jones 7
Smith 42
Miller 100
```


Fmt Utility

- The *fmt* utility is a simple text formatter that fills and joins lines to produce output up to the number of characters specified or 72 by default.
- General form.

`fmt [-w` *`width`*`] [`*`inputfiles`*`]`

Fmt Example

- Assume the following text is in a *tmp.txt* file.

```
Four score and seven years ago our fathers
brought forth on this continent, a new nation,
conceived in Liberty, and dedicated to the
proposition that all men are created equal.
```

- Using the command "fmt -w 30 tmp.txt", the output would be:

```
Four score and seven years
ago our fathers brought
forth on this continent,
a new nation, conceived in
Liberty, and dedicated to the
proposition that all men are
created equal.
```

Cut Utility

- The *cut* utility allows one to select a list of fields from one or more files and writes to standard output.
- If no files are specified, then it reads from standard input.
- General form.

  `cut [-d"<char>"] -f<num>[,<num>]* [file]*`

- The -d option allows you to change the field delimiter, which is by default a tab. The *cut* utility assumes only a single delimiter character separates each field.
- The -f indicates a list of fields to be cut. Fields are numbered starting from 1. The list can also include ranges of fields. At least one field must be specified.

Cut Examples

```
cut -d" " -f1 data.txt
   # print the first field from data.txt

cut -d" " -f2,4 data.txt
   # print the 2nd and 4th fields from data.txt

cut -d: -f1 /etc/passwd
   # print the user IDs from the passwd file

cut -d" " -f1,3-4 data.txt
   # print 1st, 3rd, 4th fields from data.txt
```


Paste Utility

- The *paste* utility merges files together by concatenating the corresponding lines of the specified input files and writes to standard output.
- If one input file is not as long as another, then paste will treat the shorter file as having empty lines at the end.
- By default, paste replaces each newline character with a tab, except for the newline character in the last file.
- You can replace the tab with some other character by using the -d option.
- General form.

  `paste [-d"<char>"] files`

Paste Example

- Paste is often used together with the cut utility.

```
# cut the first field of file1.txt and file2.txt
cut -d" " -f1 file1.txt > tmp1.txt
cut -d" " -f1 file2.txt > tmp2.txt

# paste these fields together separated by a blank
paste -d" " tmp1.txt tmp2.txt > file3.txt

# remove the temporary files
rm tmp1.txt tmp2.txt
```

Head and Tail Utilities

- The *head* utility prints the first *num* lines of one or more files to stdout.
- The *tail* utility prints the last *num* lines of one or more files to stdout.
- General form.

```
head [-[c]<num>] [files]
tail [-[c]<num>] [files]
tail [-[c]<num>] [-f] [file]
```

- If -<*num*> is not specified, then 10 lines are printed by default.
- The -c<*num*> option indicates to output the last *num* characters instead of lines.
- The -f option indicates that *tail* will monitor the *file* every second and update it if it changes.

Head and Tail Examples

```
head data.txt
   # print first 10 lines of data.txt

head -c10 data.txt
   # print the first 10 characters of data.txt

head data1.txt data2.txt
   # print the first 10 lines of data1.txt and data2.txt

grep Smith report.tex | head -5
   # print the first 5 lines containing Smith in report.tex

tail -100 data.txt
   # print last 100 lines of data.txt

tail -f tmp.out
   # monitor the last 10 lines of tmp.out
```


Sed

- The *sed* utility is a Stream EDitor that reads one or more files, makes changes according to a script of editing commands, and writes the results to standard output.
- The "*script*" is a sed command.
- You can specify more than one command by using the -e option (multiple scripts can be specified on the command line) or use the -f option and place the sed commands in a file.

```
sed "script" [file]*
sed -e "script" [-e "script"]* [file]*
sed -f scriptfile [file]*
```


Sed, Ed, and Ex

- The *sed* utility performs many *ed* or *ex* commands.
- *ed* and *ex* are line editors, which were the original type of editors.
- Line editing commands are easier for scripting.
- Line editing commands can be used in the *vi* editor and many of these commands can be used in the *sed* utility.

Line Addressing

```
n     # absolute line number n
i,j   # range of lines from line i to line j
.,$   # '.' means current line, '$' means last line
+     # go forward one line
-     # go back one line
+n    # go forward n lines
-n    # go backward n lines
```


Line Editing Commands

```
addr                # goto addr
[addr] p            # print line(s) to standard output
/<pattern>          # goto next line that matches pattern
?<pattern>          # goto previous line that matches pattern
[addr] a text .     # append text (line by line) after addr
                    # until a '.' is typed in the first column
[addr] i text .     # insert text (line by line) before addr
                    # until a '.' is typed in the first column
[addr] d            # delete specified line(s) at addr
[addr] c text .     # replace specified line(s) with text
                    # until a '.' is typed in the first column
[addr] s/pat1/pat2/
                    # substitute pat for pat1 at addr
```


Using Editing Commands in Sed

- Sed editing commands are implicitly global (applied to every line in the file).
- Example sed scripts:

```
50d           # delete line 50
/^#/d         # deletes all lines beginning with '#'
/Smith/,$d    # find first line containing "Smith" and
              # deletes that line and all remaining lines
s/old/new/    # substitute "old" with "new" in the first
              # occurrence of each line
s/old/new/g   # substitute "old" with "new" each place it occurs
s/old/new/2   # substitute "old" with "new" on the second
              # occurrence of each line
```


Performing Multiple Commands on a Match

- Can use curly braces to perform multiple commands on a single match.
- Example: Change an html file to make each list element within an ordered list a paragraph and change the ordered list to an unordered list.

```
/^<ol>/,/^<\/ol>/{    # find lines comprising an ordered list
   s/^<ol>/<ul>/      # change start to be an unordered list
   s/^<li>/<p><li>/   # change list elems to be a paragraph
   s/^<\/ol>/<\/ui>/  # change end to be an unordered list
}
```

Referencing the Search String

- The ampersand (&) represents the extent of the pattern that was matched and can be referenced in the replacement string.
- Example:

```
s/[A-Z][A-Z][A-Z]*/"&"/g
    # quote all acronyms

s/[Uu]nix/"&"/g/
    # quote all references to Unix or unix
```


Referencing Portions of the Search String

- Can reference portions of a search string by enclosing them in escaped parentheses, "\(" and "\)", and referencing them by \<num> in the replacement string.
- Example: Reversing the order of the first two integer values separated by a colon in a file.

```
s/^\([0-9]*\):\([0-9]*\)/\2:\1/
```


Gnuplot

- GNU is a free software foundation and provides a significant number of free software packages. *gnuplot* is one of these packages.
- *gnuplot* takes a sequence of plotting commands and generates plots.
- It can take its commands in an interactive (from standard input) or batch (from a file) mode.
- For detailed documentation, see:
  - http://www.gnuplot.info/documentation.html
- General form.
  - gnuplot [files]

Some Common Gnuplot Commands

```
set term <termtype> <fontsize> # set output type and font size
set output <filename>          # redirect output to a file
set xlabel <string>            # assign label for x axis
set ylabel <string>            # assign label for y axis
set xr [<min>:<max>]           # set min and max of x range
set yr [<min>:<max>]           # set min and max of y range
set xtics <start>,<incr>,<end> # label tics on x axis
set ytics <start>,<incr>,<end> # label tics on y axis
set boxwidth <width>           # set default width of boxes
set key <location>             # set location of legend
```

Gnuplot Plot Command

- General form.

  plot "*<datafile>*" using *<dataitem>*[:*<dataitem>*]
  title "*<title>*" with *<style>*

- where:

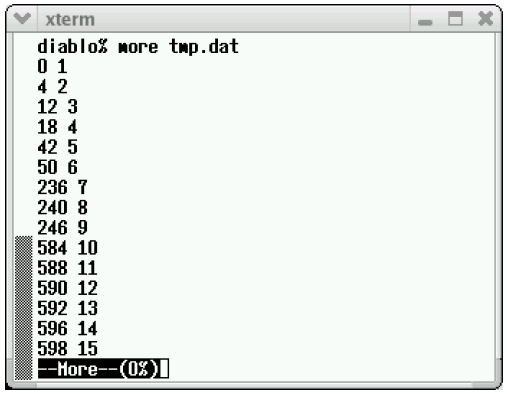
  <datafile> # name of the input data file
  <dataitem> # a variety of values including field number
  # of each data record in the *<datafile>*
  <title> # title of legend
  <style> # style in which the data is plotted, includes
  # lines, points, boxes, dots, etc.

Example Gnuplot

```
set term pdf 24            # set output as pdf and fontsize 24
set output "graph.pdf"     # set output file as "graph.pdf"
set xlabel "instruction"   # set x label to be "instruction"
set ylabel "address"       # set y label to be "address"
set key bottom right       # set legend position to bottom right
plot "tmp.dat" using 2:1 title "trace" with lines
                           # input data file is "tmp.dat", will plot
                           # field 2 in the x axis, field 1 in y axis,
                           # legend title is "trace", plot will be with
                           # lines connecting the data points
```

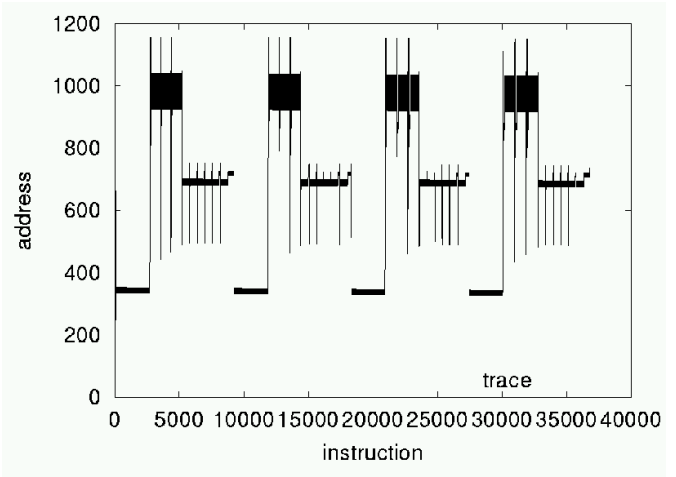

Example Gnuplot Input Data

- Below is a snapshot of the first portion of the input data in "tmp.dat".



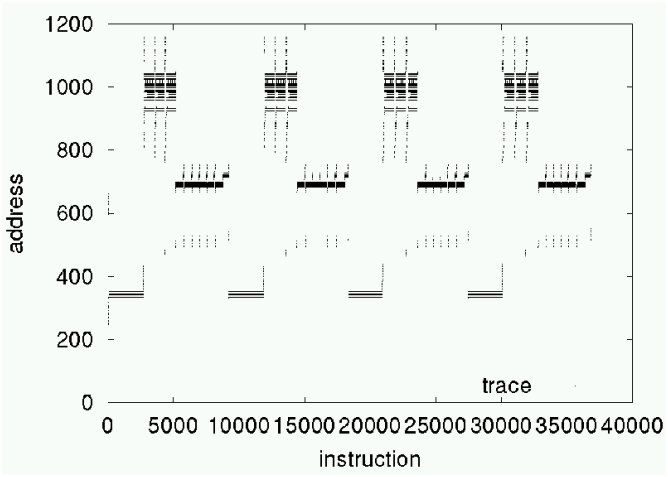
Example Gnuplot Output

- Below is the output of gnuplot using the example commands and input data shown in the previous two slides.



Another Example Gnuplot Output

- Below is the same output, but plotted as dots instead of lines.



Split Utility

- The *split* utility allows one to split a file into smaller files.
- General form:

  `split [-`*`linecount`*`] [-b `*`bytecount`*`[k|m]] [`*`file`*` [`*`name`*`]]`

- It splits the specified file into files called *name*aa, *name*ab, *name*ac, etc.
- If *name* is not specified, then it is "x" by default.
- You can either specify a *linecount* or *bytecount* to control the amount of characters put in each file.
- The "k" and "m" indicates byte factors of 2^10 and 2^20, respectively.
- If both a *linecount* and *bytecount* are not specified, then it is 1000 lines by default.

Split Examples

```
split -500 log.txt log.split.
   # splits log.txt into files called log.split.aa, log.split.ab, ...
   # that are each 500 lines (except for the last file)

split -b 128k log.txt
   # splits log.txt into files called xaa, bab, ... that are each
   # 131,072 bytes (except for the last file)
```


Csplit Utility

- The *csplit* utility splits a file according to context where one can give search patterns at which to break each section.
- General form:

  `csplit [-f `*`pref`*`] `*`file pat1`*`[`*`offset1`*` [{`*`cnt1`*`}]`
  `... `*`patn`*`[`*`offsetn`*` [{`*`cntn`*`}]`

- It splits the file into *pref*00, *pref*01, etc.
- If a *pref* is not given, then "xx" is used at the start of each file.
- The *pat*'s are the search patterns.
- Each file contains lines up to but not including the next pattern matched.
- The *offset* indicates the number of extra or fewer lines to include from the point of the match.
- The {*cnt*} indicates the maximum number of times the pattern should be used.

Csplit Examples

```
csplit report.txt /I\./ /II\./ /III\./
   # splits reports into 4 files called xx00, xx01, xx02, and xx03

csplit prog.c '/^}$/+1' "{*}"
   # splits prog.c into different routines where it is assumed that
   # each routine ends with a '}' on a line by itself
```


Uniq Utility

- The *uniq* utility reads a file and compares adjacent lines.
- The -u option indicates to print lines that are unique.
- The -d option indicates to print lines that occur more than once.
- The default is to print only the first occurrence of each line.
- General form:

  `uniq [-d] [-u] [`*`input_file`* `[`*`output_file`*`]]`

Uniq Examples

- Uniq is often useful after the sort command.

```
sort names.txt | uniq -d
   # show names that appear more than once

sort names.txt | uniq -u
   # show names that were unique

sort scores.txt | uniq
   # show scores that occurred
```