Introduction to Cluster Computing:

Linux, shell scripting, queuing systems, cluster architecture

Instructor: Dr. Peggy Lindner (plindner@uh.edu)

Lecture 3 (awk & sed)

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

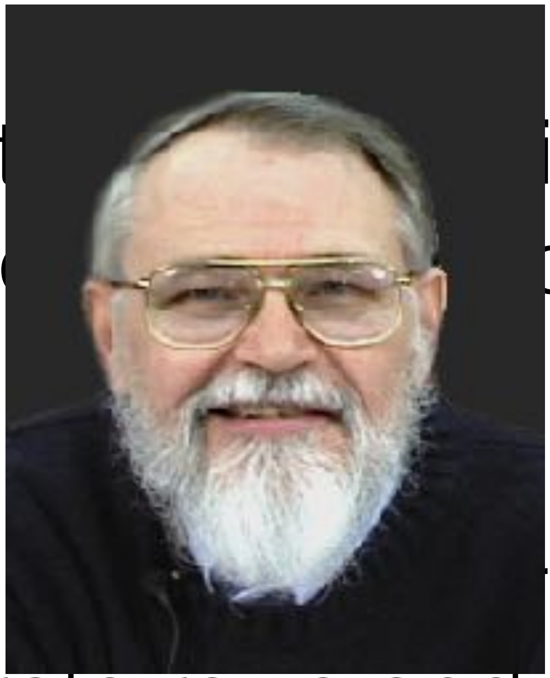
Introduction to awk

- AWK: a programming language - interpreted (not compiled) for pattern scanning and processing text-based data in files or data streams.

- The n s n t its three devel c W d Brian Kerni

Aho

Weinberger


Kernighan

- It is a f rt rammable filter"). Many UNIX utilities generate rows and columns of information. AWK is an excellent tool for processing these rows and columns, and is easier to use AWK than most conventional programming languages.

Further reading::
sed & awk, by Dale Dougherty, O'Reilly & Associates, 1991, ISBN 0-937175-59-5.

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

awk applications

- Works well on record-type data, read one line at a time, parses each line into fields
- Unix file processing, e.g., as part of a pipe
- avoid read loop in shells, Perl, more intuitive syntax than shell scripts
- best limited to line-at-a-time processing (Performs user-defined tests against each line, performs actions on matches)
- Input validation
  - Every record have same # of fields?
  - Do values make sense (negative time, hourly wage > $1000, etc.)?
- Filtering out certain fields
- Searches
  - Who got a zero on lab 3?
  - Who got the highest grade?

Form of an awk program

- The essential organization of an AWK program follows the form:
  pattern { action }

- Actions =
  - if (conditional) statement else statement
  - while (conditional) statement
  - break
  - continue
  - variable=expression
  - print expression-list

- `awk` program consists of:
  - An optional BEGIN segment
    - For processing to execute prior to reading input
  - pattern - action pairs
    - Processing for input data
    - For each pattern matched, the corresponding action is taken
  - An optional END segment
    - Processing after end of input data

```
BEGIN {action}
pattern {action}
pattern {action}
 .
 .
 .
```

Further reading::
sed & awk, by Dale Dougherty, O'Reilly & Associates, 1991, ISBN 0-937175-59-5.

Running an AWK Program

- There are several ways to run an Awk program
  - `$ awk '{ print $3 }' input.txt`
    - Can write little one-liners on the command line (very handy), e.g. print the 3rd field of every line
  - `awk -f script.awk input.txt`
    - Execute an awk script file
  - `#!/bin/awk -f`
    - Or, use this sha-bang as the first line, and give your script execute permissions

Patterns and Actions

- Search a set of files for *patterns*.
- Perform specified *actions* upon lines or fields that contain instances of patterns.
- Does not alter input files.
- Process one input line at a time
- This is similar to **sed**

Pattern-Action Structure

- Every program statement has to have a *pattern* **or** an *action* **or** both
- Default *pattern* is to match all lines
- Default *action* is to print current record
- Patterns are simply listed; actions are enclosed in **{ }**
- `awk` scans a sequence of input lines, or records, one by one, searching for lines that match the pattern
  - Meaning of match depends on the pattern

Patterns

- Selector that determines whether *action* is to be executed
- *pattern* can be:
  - the special token **BEGIN** or **END**
  - regular expression (enclosed with //)
  - relational or string match expression
  - **`!`** negates the match
  - arbitrary combination of the above using **`&&`** **`||`**
    - **`/NYU/`** matches if the string "NYU" is in the record
    - **`x > 0`** matches if the condition is true
    - **`/NYU/ && (name == "UNIX Tools")`**

Actions

- *action* may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams.
- *action* is performed on every line that matches *pattern*.
  - If *pattern* is not provided, *action* is performed on every input line
  - If *action* is not provided, all matching lines are sent to standard output.
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

Variables

- awk scripts can define and use variables (not declared nor typed)

- No character type (only strings and floats with support for ints)

```
BEGIN { sum = 0 }        # prints each field on the line
{ sum ++ }               for( i=1; i<=NF; ++i )
END { print sum }         print $i
```

- Some variables are predefined
  - FS – the input field separator
  - OFS – the output field separator
  - NF – # of fields; changes w/each record
  - NR – the # of records read (so far). So, the current record #
  - FNR – the # of records read so far, reset for each named file
  - $0 – the entire input line

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

Example - Handling Text

- One major advantage of Awk is its ability to handle strings as easily as many languages handle numbers

- Awk variables can hold strings of characters as well as numbers, and Awk conveniently translates back and forth as needed

- Print pay for those employees who actually worked

```
$ awk '$3>0 {print $1, $2*$3}' emp.data
 Kathy 40
 Mark 100
 Mary 121
 Susie 76.5
```

Beth	4.00	0
Dan	3.75	0
Kathy	4.00	10
Mark	5.00	20
Mary	5.50	22
Susie	4.25	18

String Manipulation

- String Concatenation
  - New strings can be created by combining old ones

```
    { names = names $1 " " }
END { print names }
```

- Printing the Last Input Line
  - Although NR retains its value after the last input line has been read, $0 does not

```
    { last = $0 }
END { print last }
```

Example 1 - CSV file

- Create email address from last column

```
$ cat getEmails.awk
#!/bin/awk -f

BEGIN { FS = "," }
{ printf( "%s's email is: %s@school.edu\n", $2, $3 ); }
```

```
$ getEmails.awk students.csv
 john's email is: js12@school.edu
 fred's email is: fj84@school.edu
 sue's email is: sb23@school.edu
 ralph's email is: rf86@school.edu
 jim's email is: jj22@school.edu
 nancy's email is: nc54@school.edu
 anna's email is: ab67@school.edu
 sam's email is: sr77@school.edu
 lisa's email is: guitarHottie@school
```

```
smith,john,js12
jones,fred,fj84
bee,sue,sb23
fife,ralph,rf86
james,jim,jj22
cook,nancy,nc54
banana,anna,ab67
russ,sam,sr77
loeb,lisa,guitarHottie
```

Example 2 - CSV file

- Create email address from last column

```
#!/bin/awk -f
BEGIN { FS = ","; OFS = "-*-"; }
{ print $1, $2, $3; }
```

```
$ out.awk students.csv
 smith-*-john-*-js12
 jones-*-fred-*-fj84
 bee-*-sue-*-sb23
 fife-*-ralph-*-rf86
 james-*-jim-*-jj22
 cook-*-nancy-*-nc54
 banana-*-anna-*-ab67
 russ-*-sam-*-sr77
 loeb-*-lisa-*-guitarHottie
```

Built-in Functions

- `awk` contains a number of built-in functions. *length* is one of them.
- 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.

Built-In Functions cont.

- Arithmetic
  - *sin, cos, atan, exp, int, log, rand, sqrt*
- Output
  - print, printf
- Special
  - system - executes a Unix command
    - system(“clear”) to clear the screen
    - Note double quotes around the Unix command
  - exit - stop reading input and go immediately to the END pattern-action pair if it exists, otherwise exit the script

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"
    }
```

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
 }
}
```

- Do While

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

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)
}
```

Arrays

- Array elements are not declared

- Array subscripts can have ***any*** value:
  - Numbers
  - Strings! (*associative arrays*)

- They are implemented using hash tables, e.g.

```
arr[3]="value"
grade["Korn"]=40
```

- It is possible to loop over all indices that have currently been assigned values.

```
for (name in Total)
  print name, Total[name];
```

Example - Arrays

- Calculate total scores

```
$ cat total.awk
{ Total[$1] += $2}
END {
 for (i in Total)
    print i, Total[i];
}
```

```
$ awk -f total.awk scores
 Sue 198
 Sam 120
 Fred 245
```

```
Fred 90
Sue 100
Fred 85
Sam 70
Sue 98
Sam 50
Fred 70
```

Useful One-Liners

- Line count
```
awk 'END {print NR}’
```

- head
```
awk 'NR<=10'
```

- Number of fields per line
```
awk ‘{print $NF}’
```

- Total number of fields
```
awk '{ nf = nf + NF} END { print nf }'
scores
```

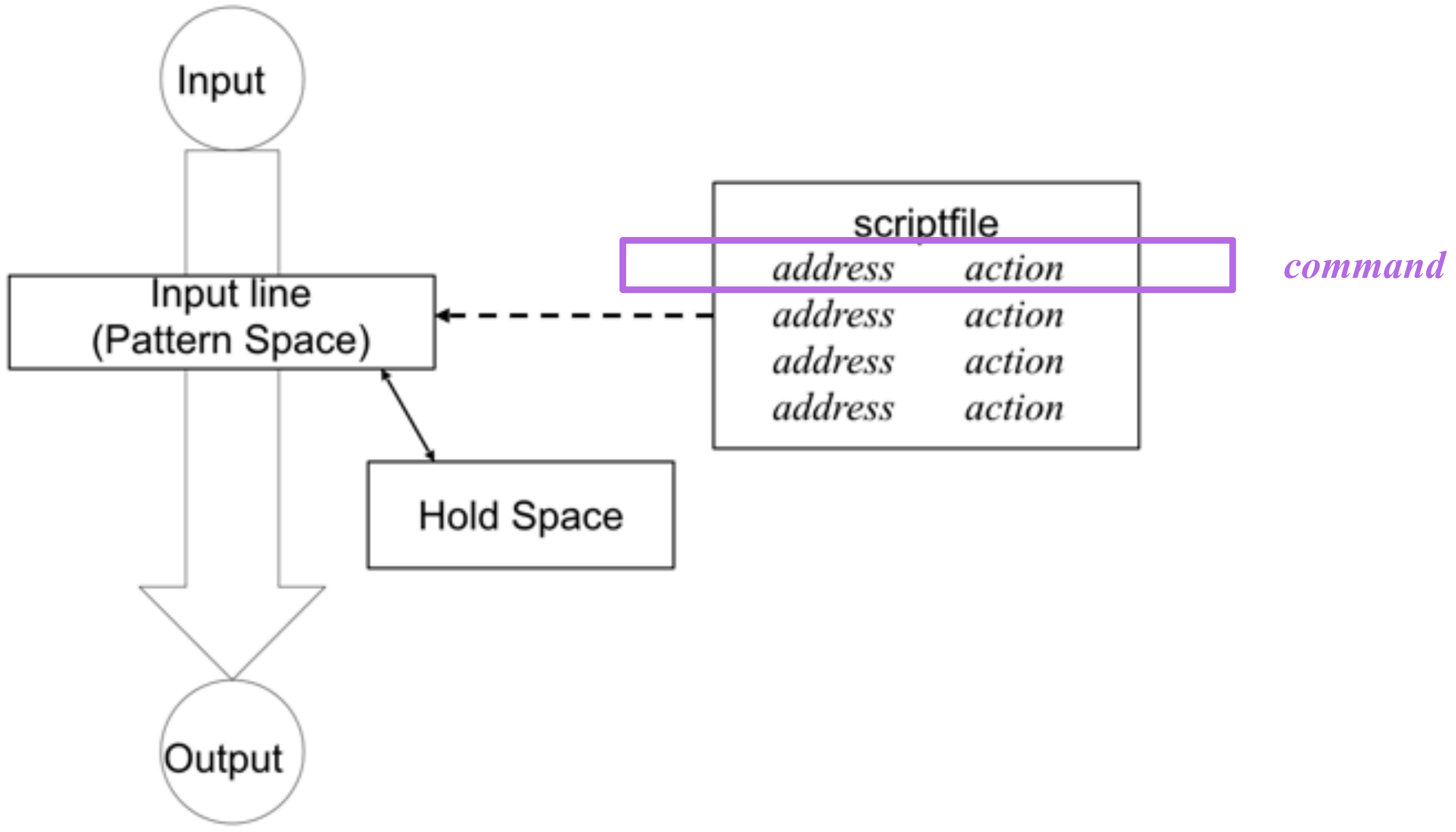
Awk Features over Sed

- Convenient numeric processing
- Variables and control flow in the actions
- Convenient way of accessing fields within lines
- Flexible printing
- Built-in arithmetic and string functions
- C-like syntax

Introduction to sed

- Sed- Stream-oriented, Non-Interactive, Text Editor
- string replacement (Look for patterns one line at a time, like grep) - changes lines of a file
- Line-oriented tool for pattern matching and replacement (stream editor)
- Not really a programming language (cf. awk)
- E.g., apply same change to lots of source files
- Filter, i.e., does not modify input file

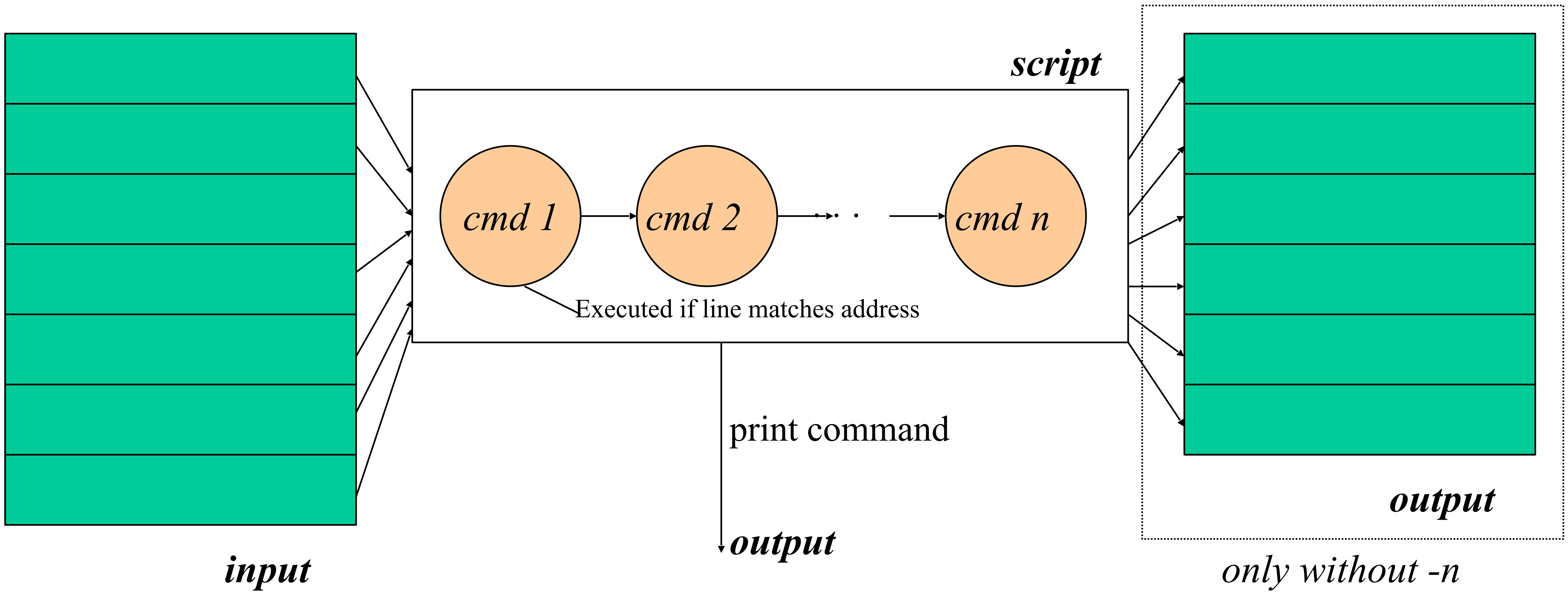
sed architecture



- Each sed command consists of up to two *addresses* and an *action*, where the *address* can be a regular expression or line number.

Sed Flow of Control

- `sed` then reads the next line in the input file and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input file



sed Syntax

- Syntax: *sed [-n] [-e] ['command'] [file…]*

  sed [-n] [-f scriptfile] [file…]

  - *-n* - only print lines specified with the print command (or the 'p' flag of the substitute ('s') command)
  - *-f* scriptfile - next argument is a filename containing editing commands
  - *-e* command - the next argument is an editing command rather than a filename, useful if multiple commands are specified
  - If the first line of a scriptfile is "***#n***", sed acts as though ***-n*** had been specified

sed Commands

- sed commands have the general form
  - *[address[, address]][!]command [arguments]*
- *sed* copies each input line into a *pattern space*
  - If the address of the command matches the line in the *pattern space*, the command is applied to that line
  - If the command has no address, it is applied to each line as it enters *pattern space*
  - If a command changes the line in *pattern space*, subsequent commands operate on the modified line
- When all commands have been read, the line in *pattern space* is written to standard output and a new line is read into *pattern space*

Addressing

- An address can be either a line number or a pattern, enclosed in slashes (*/pattern/*)
- A pattern is described using *regular expressions* (BREs, as in **grep**)
- If no pattern is specified, the command will be applied to **all** lines of the input file
- To refer to the last line: **`$`**
- Most commands will accept two addresses
  - If only one address is given, the command operates only on that line
  - If two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively
- The **`!`** operator can be used to negate an address, ie; *address! command* causes *command* to be applied to all lines that do ***not*** match *address*

Commands & Address together

- command is a single letter
- Example: Deletion - d `[address1][,address2]d`
  - Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
  - A new line of input is read and editing resumes with the first command of the script.

`$ sed d scores`	deletes the all lines
`$ sed 6d scores`	deletes line 6
`$ sed /^$/d theraven.txt`	deletes all blank lines
`$ sed 1,10d theraven.txt`	deletes lines 1 through 10
`$ sed 1,/^$/d theraven.txt`	deletes from line 1 through the first blank line
`$ sed /^$/,$d theraven.txt`	deletes from the first blank line through the last line of the file
`$ sed /^$/,10d theraven.txt`	deletes from the first blank line through line 10

- Using regular expression we can have complex forms, e.g.

  `sed /^ya*y/,/[0-9]$/d` deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit

Multiple Commands

- Braces **{ }** can be used to apply multiple commands to an address

```
[/pattern/[,/pattern/]]{
command1
command2
command3
}
```

- Strange syntax:
  - The *opening brace* must be the last character on a line
  - The *closing brace* must be on a line by itself
  - Make sure there are no spaces following the braces

Print

- The Print command (`p`) can be used to force the pattern space to be output, useful if the *-n* option has been specified
- Syntax: `[address1[,address2]]p`
- Note: if the *-n* option has not been specified, `p` will cause the line to be output twice!
- Examples:

  `sed 1,5p scores` will display lines 1 through 5

  `sed /^$/,$p theraven.txt` will display the lines from the first blank line through the last line of the file

Substitute

- Syntax: `[address(es)]s/pattern/replacement/[flags]`
  - pattern - search pattern
  - replacement - replacement string for pattern
  - flags - optionally any of the following

n	a number from 1 to 512 indicating which occurrence of pattern should be replaced
g	global, replace all occurrences of pattern in pattern space
p	print contents of pattern space

- Examples:

`s/Puff Daddy/P. Diddy/`

Substitute P. Diddy for the first occurrence of Puff Daddy in *pattern space*

`s/Tom/Dick/2`

Substitutes Dick for the second occurrence of Tom in the *pattern space*

`s/wood/plastic/p`

Substitutes plastic for the first occurrence of wood and outputs (prints) *pattern space*

Replacement Patterns

- Substitute can use several special characters in the *replacement* string
  - `&` - replaced by the entire string matched in the regular expression for pattern
  - `\n` - replaced by the nth substring (or subexpression) previously specified using “\(“ and “\)”
  - `\` - used to escape the ampersand (&) and the backslash (\)
- Examples:

```
$ echo "the UNIX operating system ..." | sed 's/.NI./
   wonderful &/'
"the wonderful UNIX operating system …"

$sed 's/\(.*\):\(.*\)/\2:\1/' test1
second:first
two:one
```

Append, Insert, and Change

- Syntax for these commands is a little strange because they **must** be specified on multiple lines
- **append** *[address]a\*

  text
- **insert** *[address]i\*

  text
- **change** *[address(es)]c\*

  text
- append/insert for single lines only, not range

Using `!`

- If an address is followed by an exclamation point (`!`), the associated command is applied to all lines that don't match the address or address range
- Examples:

  `1,5!d` would delete all lines except 1 through 5

  `/black/!s/cow/horse/` would substitute "horse" for "cow" on all lines except those that contained "black"

"The brown cow" -> "The brown horse"

"The black cow" -> "The black cow"

Transform

- The Transform command (`y`) operates like `tr`, it does a one-to-one or character-to-character replacement
- Transform accepts zero, one or two addresses

  ```
  [address[,address]]y/abc/xyz/
  ```

  - every *a* within the specified address(es) is transformed to an *x*. The same is true for *b* to *y* and *c* to *z*
- Example:

  ```
  $ sed y/abcdefghijklmnopqrstuvwxyz/
  ABCDEFGHIJKLMNOPQRSTUVWXYZ/ the raven.txt
  ```

  changes all lower case characters on the addressed line to upper case
- If you only want to transform specific characters (or a word) in the line, it is much more difficult and requires use of the *hold space*

Quit

- Quit causes `sed` to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This can be used to save time when you only want to process some portion of the beginning of a file
- Example: to print the first 100 lines of a file (like *head*) use:
  - `sed '100q' filename`
  - `sed` will, by default, send the first 100 lines of *filename* to standard output and then quit processing

Sed Pros/Cons

- Pros:
  - Regular expressions
  - Fast
  - Concise

- Cons:
  - Hard to remember text from one line to another
  - Not possible to go backward in the file
  - No way to do forward references like `/..../+1`
  - No facilities to manipulate numbers
  - Cumbersome syntax