UCS1304 UNIX and Shell Programming

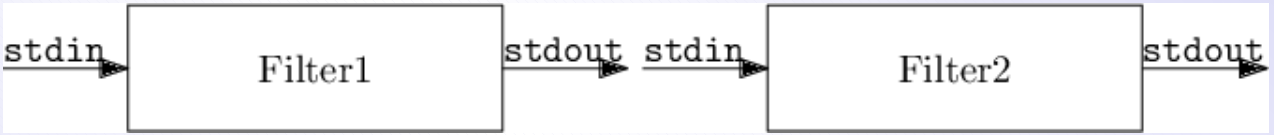
UNIT II REVIEW

S Milton Rajendram

21 August 2019

1. Filters

- Reads the input from a file or `stdin`, and writes its output to `stdout`.
- Text input, formatted as lines.



1.1. `head`, `tail`

- `head -m file`
- `tail -m file`
  Print the last `m` lines
- `tail -n +m`
  Print from `m` th line

- Combine `head` and `tail` to select any subsequence of lines

1.2. `cut`

- `cut` selects fields of the input (files or `stdin`)
- Input is text, sequence of lines, each line sequence of fields
- Specify character positions
  `cut -cn1,n2-n3,n4-n5` prints characters $n_1, n_2, n_2+1, \ldots, n_3, n_4, n_4+1, \ldots, n_5$
- Specify fields
  - Field separator: single tab
    ```
    cut -fn1,n2-n3,n4-n5
    ```
  - Specify field separator
    ```
    cut -d"c" -fn1,n2-n3,n4-n5
    cut -d":" -f7,1 /etc/passwd | head
    ```

- cut fields are printed in order; you cannot change the order.

1.3. paste

- `paste file1 file2`
  combines lines from files horizontally, separate them tab

  ```
  cut -f1,2 file1 > file2
  cut -f3 file1 > file3
  paste file3 file2
  ```

  prints columns (fields) `3, 1, 2` in that order.

- `-d"c"` delimiter characters

1.4. `sort`

- Text input, sequence of lines, each line sequence of fields (separated by string of blanks/tabs)
  - digits (30-39)
  - uppercase letters (41-5A)
  - lowercase letters (61-7A)
- Sort key `(field1, field2, ...)`
- Sort by fields
  - `sort -km file`
    sort by fields `m, m+1, m+2, ...` till the last field.
  - `sort -km,n file`
    sort by fields `m` to `n`
  - sort by field `m` only (no other field)
    `sort -km,m file`

- Specify field separator character c

  ```
  sort -t"c" file
  ```

  ```
  sort -t ':' -k 7 /etc/passwd
  ```

- Sort by number: `sort -n`
  - By default, `sort` sorts strings: e.g. as strings, 123 comes before 89.
  - `sort -n` sorts numbers: as numbers, 123 comes after 89.
- Sort reverse: `sort -r`
- Merge sorted files `sort -m files`
- Fold case `sort -f files`

1.5. Transliterate from one set to another

- `stdin` to `stdout`; `tr` does not read file

- `tr options string1 string2`
  `tr "aeiou" "AEIOU"`
- Each character in set1 (`string1`) is converted to the corresponding character in set2 (`string2`)
- If set2 is smaller than set1, unmatched characters in set1 are converted to the last chracter in set2.

```
echo "Shout for Joy!" | tr a-z A-Z
```

```
SHOUT FOR JOY!
```

- Delete characters in set
  `tr -d set`
- Convert the complement of set

```
echo "not to be contentious, gentle," | tr -c aeiou ?
```

- Squeeze (delete) repeated instances of characters `tr -s set`
  replace each sequence of a repeated character listed in `set`, with a single occurrence of that character

```
echo "aaabbbccc" | tr -s ab
```

```
cat phone.txt | tr -cs a-zA-Z "\n"
```

1.6. Comparing files

`cmp, diff, comm`

- Compare (`cmp`) `cmp file1 file2`
  Displays the line number and byte number of the first differing byte `cmp -s file1 file2`

- Difference (`diff`)

- `diff` always works on files (two files, two versions of a file) `diff file1 file2`
- Change command: `range1 operation range2`

Operation	Action
`r1ar2`	At position `r1` in `file1`, append lines at `r2` in `file2`
`r1cr2`	Change (replace) lines at `r1` with the lines at `r2` in `file2`.
`r1dr2`	Delete lines at `r1` in `file1`, which would have appeared at range `r2`

range is comma separated list of starting line and ending line

- Context format

`diff -c file1 file2`

Character	Meaning
blank	This line is shared by both files
–	This line was removed from the first file.
+	This line was added to the first file.

- Unified format

```
diff -u file1 file2
```

- Patch (`patch`)
  - Create a diff file

```
diff -Naur file1.txt file2.txt > patchfile.txt
```

```
patch < patchfile.txt
```

2. Regular Expressions

- Text input, sequence of lines `grep pattern files`
- Pattern is a RE

```
for each line in the inputs:
   if the pattern matches a string in the line:
       print the line
```

2.1. Symbols (atoms)

A symbol matches a single character

- Literal character matches itself.
- Dot matches any single character, except newline.

- Anchors
  - `^r` matches `r` at the beginning of line
  - `r$` matches `r` at the end of line
  - `\<r` matches `r` at beginning of word
  - `r\>` matches `r` at end of word
- Character class matches any one of the characters in a set.
  - In a character class, metacharacters are considered literal characters
  - Complement: `^` as the first character in a class

    `grep -h '[^bg]zip' file`

    matches any string ending with `zip` except `bzip` and `gzip`
  - Range

    `grep -h '^[A-Za-z0-9]' file`

    matches a letter or a digit at the start of the line
- Back references

- `\1, \2, \ldots, \9`

```
echo "precept upon precept | grep -E '(precept).*\1' file
```

matches a string starting with `precept` and ending with another `precept`. `\1` refers to the matching string of the first RE – `precept` in this case.

2.2. Extended `grep`: `egrep` or `grep -E`

- Extended grep

`c`	any non-special character `c` matches itself
`\c`	turn off any special meaning of character `c`
`^`	beginning of line
`$`	end of line
`[...]`	any one of characters in `...`; ranges like `a-z` are legal
`[^...]`	any single character not in `...`; ranges are legal
`\n`	string matched by `n`'th `\( ... \)` group (grep only)
`r*`	zero or more occurrences of `r`
`r+`	one or more occurrences of `r` (egrep only)
`r?`	zero or one occurrence of `r` (egrep only)
`r1r2`	`r1` followed by `r2`
`r1 | r2`	`r1` or `r2` (egrep only)
`\(r\)`	tagged regular expression `r` (grep only); can be nested
`(r)`	regular expression `r` (egrep only); can be nested
	No regular expression matches a newline.

- Specific to egrep

```
r?
r+
r{m,n}
(r)
```

2.3. RE for identifiers in C

```
echo "varname" | grep -E '^[A-Za-z_][A-Za-z0-9_]*$'
```

Note _ is considered a letter.

2.4. RE for phone numbers

```
echo "2229-4254" | grep -E '^[1-9][0-9]{3}-?[0-9]{4}$'
echo "22294254" | grep -E  '^[1-9][0-9]{3}-?[0-9]{4}$'
echo "2229 4254" | grep -E  '^[1-9][0-9]{3}-?[0-9]{4}$'
```

2.5. RE for Roman numbers

Write a RE for matching Roman numbers.

2.6. Options

- `-i --ignore-case`
  Ignore case.
- `-v --invert-match`
  Invert match.
- `-c --count`
  Print the number of matches
- `-l --files-with-matches`
  Print the name of each file that contains a match
- `-L --files-without-match`

Invert `-l`.

- `-n --line-number`

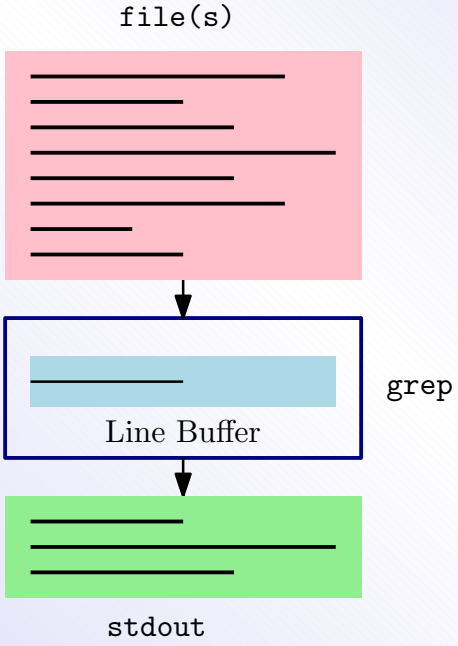
  Prefix each matching line with the number of the line within the file.

- `-h --no-filename`

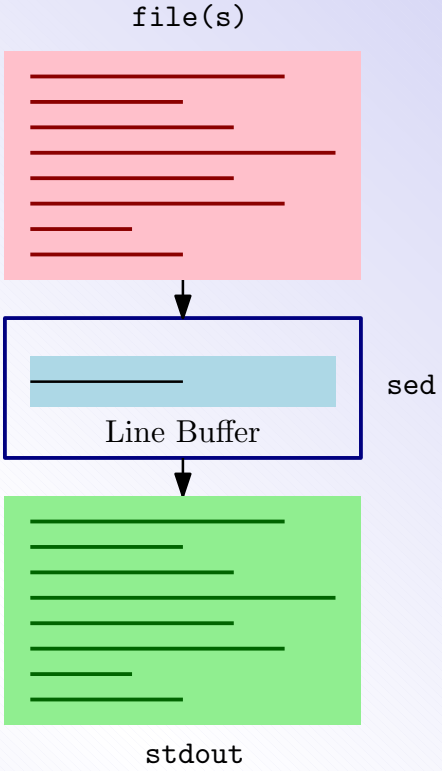
  For multi-file searches, suppress the output of filenames.

3. SED

- **s**tream **ed**itor



Prints only the matching lines of the input



Prints same lines as in the input (except those inserted and deleted)

```
sed commands ...  filenames
```

1. read next line
2. edit the line using commands
3. write edited line
4. goto step 1

- Syntax of sed script

```
selector1 command1
selector2 command2
...
selectorm commandm
```

Save the secript in a file `script` and apply it on `infiles`

```
sed -f script infiles
```

or

```
sed -e 'selector1 command1' -e 'selector2 command2' ... filename(
```

- Meaning

```
for each line in the inputs:
  for each command in commands:
     if line is in selector of command:
        apply command on line
```

3.1. Line selector (Address)

- Line number

  n

  ```
  sed -n '9p' file # print line 9
  ```

  Option -n turns of the default printing. We want to explicitly print line 9, using the command p

  ```
  sed '3s/  */:/g' file # line 3, replace a string of one or more
  ```

- Last line

- `$`

```
sed -n '$p' file       # print the last line
sed '$s/  */:/g' file # do the substitution in the last line
```

- Regular expression `/regexp/`

```
sed -n '/flight/p' file # print every line matching flight
```

- Range of line numbers

  `addr1,addr2`

```
sed -n '5,9p'                # print lines 1 to 5
sed '1,5s/  */:/g' file      # do the substitution in lines 1 to 5
sed -n '/wing/,/flight/p' # line matching wing to line matching
```

- `first ~ step`

```
sed -n '1~5p'    # lines 1, 1+5, 1+2*5, ...
```

- `addr1,+n`

```
sed -n '6,+3p'   # lines 6, 6+1, 6+2, 6+3
```

- addr!

```
sed -n '/flight/!p' file # lines not matching flight
```

3.2. Commands (Editing Operations)

- p
  Print the current line.
- s/regexp/replacement/
  - &, the matching string
  - \1 through \9, matching strings of groups \(\)
- =
  Output the current line number.

- `i`
  Insert text before the current line.
- `a`
  Append text after the current line.
- `c`
  Change lines to following text as in `a`
- `d`
  Delete the current line.
- `q`
  Exit sed without processing any more lines.
- `y/set1/set2`
  Transliterate. Both sets must be of the same length.

3.3. s command

```
echo "aaabbbccc" | sed 's/b/B/'  # without g option only the first
echo "aaabbbccc" | sed 's/b/B/g' # option g (global) all matches in
sed 's/  */\t/g' file # option g (global) all matches in a line
sed 's/^  *//' file   # replacement is empty = delete the match
sed 's/$/\n/' file    # doublespace
echo a b c d| sed 's/.* /Y/'   # longest (greedy) match
Yd
```

3.4. Other commands

- q

```
sed '5q' file   # quit after line 5
```

- d

```
sed '4,6d' file # delete lines 4,5,6
```

- i

```
# insert ========================== before line 2
sed '2i ==========================' file
```

- a

```
# insert ========================== before line 1
sed '1a ==========================' file
```

3.5. Back reference

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/'
file
```

```
# misstep 1: simple, straight
[0-9]{2}/[0-9]{2}/[0-9]{4}$
# misstep 2: group each of the three matches (...), (...), (...)
([0-9]{2})/([0-9]{2})/([0-9]{4})$
# misstep 3: refer to the matched groups \1, \2, \3
([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2
# misstep 4: escape forward slash
([0-9]{2})\/([0-9]{2})\/([0-9]{4})$/\3-\1-\2
# step 4: escape parentheses and braces
\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2
# step 5: escape parentheses and braces
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' fi
```

3.6. Insert command

```
# insert 3 lines before line 1
1 i\
\
Linux Distributions Report\

# substitute
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
# transliterate
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

3.7. A Few Examples

```
sed -n '20,30p' # Print only lines 20 through 30
sed '1,10d'     # Delete lines 1 through 10 (~tail -n +11~)
```

```
sed ' 1,/^$/d'   # Delete up to and including first blank line
sed '$d'         # Delete last line
```

4. AWK

- Powerful filter
- Programming language

4.1. Structure of AWK Program

- AWK program is a sequence of pattern-action statements

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

- `action` is a sequence of statements

```
pattern {
            statement
```

```
            statement
            ...
        }
pattern {
            statement
            statement
            ...
        }
...
```

- Input: sequence of lines, each line sequence of fields. In AWK, lines are also referred to as records.

```
awk '$3 > 0 {print $1, $2 * $3}' file
```

```
awk '$3 == 0 {print $1}' file
```

- Pattern or action can be omitted, not both
- Process

```
for each input line:
  for each pattern-action statement:
     if the line matches the pattern:
        apply action on line
```

- Pattern is a condition – matching line means condition is true

4.2. Simple Use

- Two data types: string, numbers
- Each line is a sequence of fields – blanks/tabs is field separator
- Read one line at a time, split it into fields
- Fields are numbered $1, $2, ...
- $0 is the entire line

Print every line

```
cat emp.data
```

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

```
# emp.data has 3 fields: employee name, hourly rate, number of hou
awk '{ print $0 }' emp.data
```

Print certain fields

```
awk '{ print $1, $3}' emp.data
```

Number of fields

- `NF` built-in variable

```
awk '{ print NF, $1, $NF }' emp.data
```

Computing and printing

```
awk '{ print $1, $2 * $3 }' emp.data
```

Line numbers

- `NR` Number of records (lines)

```
awk '{ print NR, $0 }'
```

Putting text in the output

```
awk '{ print "total pay for", $1, "is", $2 * $3 }' emp.data
```

4.3. Selection

- Without patterns, action is on all lines
- Pattern selects lines
- Pattern is a Boolean expression

Selection by comparison

```
awk '$2 >= 5' emp.data
```

Selection by computation (arithmetic, comparision)

```
awk '$2 * $3 > 50 { printf("$%.2f for %s\n", $2 * $3, $1) }' emp.dat
```

Selection by text content

Select lines in which the first field is "Susie"

```
awk '$1 == "Susie"' emp.data
```

Select "matching" lines by RE – lines that contain "Susie" anywhere

```
awk '$0 ~ /Susie/' emp.data
```

```
Susie 4.25 18
```

~ is RE match operator. `$0 ~ /Susiee/` means
Does current line `$0` match `Susie`?

Combination of patterns (arithmetic, comparison, logical operators)

- Logical operators

Logical operation	AWK operator
AND	&&
OR	||
NOT	!

```
awk '$2 >= 4 && $3 >= 20' emp.data
```

- Multiple pattern-action statements in command line

```
awk -e '$2 >= 4' -e '$3 >= 20' emp.data
```

BEGIN and END blocks

- Special patterns
- BEGIN matches before the first line of the first input file is read

- END matches after the last line of the last file has been processed.

```
awk -e 'BEGIN { print "NAME RATE HOURS"; print "" }' -e '{ print
```

4.4. Computing wih AWK

- Action is a sequence of statements, separated by newlines or semicolons.

Counting

```
# numeric variables are automatically initialized to 0
$3 > 15 { emp = emp + 1 }
END { print emp, "employees worked more than 15 hours" }
```

```
3 employees worked more than 15 hours
```

Computing sums and averages

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

```
6 employees
total pay is 337.5
average pay is 56.25
```

Computing maximum

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

```
highest hourly rate: 5.50 for Mary
```

Printing the last line

NR is global, $0 is not.

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

```
Susie 4.25 18
```

Built-in functions

- Math: square root, log, random
- String: length

```
{ print $1, length($1) }
```

```
Beth 4
Dan 3
Kathy 5
```

```
Mark 4
Mary 4
Susie 5
```

Counting lines, words, and characters

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

```
6 lines, 18 words, 84 characters
```

- $0 does not include newline.

4.5. Control-Flow Statements

- Modeled on C

If-Else statement

```
$2 > 6 { n = n+ 1; pay = pay + $2 * $3 }
END    { if (n > 0)
           print n, "employees, total pay is", pay, "average pay is
         else
           print "no employees are paid more than $6/hour"
       }
```

```
no employees are paid more than $6/hour
```

While statement

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

```
1000 0.06 1060.00
1000 0.06 1123.60
1000 0.06 1191.02
2000 0.12 2240.00
2000 0.12 2508.80
2000 0.12 2809.86
```

```
2000 0.12 3147.04
```

For statement

```
# interest2 - compute compound interest
# input: amount rate years
# output: compounded value at the end of each year
{ for (i = 1; i <= $3; i = i + 1)
     printf("%d\t%.2f\t%.2f\n", $1, $2, $1 * (1 + $2) ^ i)
}
```

```
1000 0.06 1060.00
1000 0.06 1123.60
1000 0.06 1191.02
2000 0.12 2240.00
2000 0.12 2508.80
```

```
2000 0.12 2809.86
2000 0.12 3147.04
```

4.6. Arrays

```
# reverse - print input lines in reverse order
    { line[NR] = $0 }
END { for (i = NR; i > 0; i = i - 1)
            print line[i]
    }
```

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