Question:

Use a regular expression to convert from the first format to the second:

```
last name, first name initial

->

first name initial last name
```

Writing bash Shell Scripts

Mark G. Sobell

The Bourne Again Shell (`bash`)

- `bash` is a command interpreter and a high-level programming language
- `bash` is based on the original Bourne Shell (`sh`) with some elements of the Korn Shell (`ksh`)
- `bash` maintains backward compatibility with `sh` (run as `sh`)
- `bash` is almost POSIX compatible (`--posix` option)
- `bash` is very similar to the Z Shell (`zsh`) which is based on the Korn Shell.

Getting Warmed Up...

`#!` Specifies the shell (`/bin/sh` is a link to `/bin/bash`).
You must have read and execute access to a script to execute it as a command

```
[mark@tuna shelltalk.examples]$ PS1='\$ '

$ whereis zless
zless: /usr/bin/zless /usr/share/man/man1/zless.1.gz

$ file /usr/bin/zless
/usr/bin/zless: Bourne shell script text executable

$ ls -l /usr/bin/zless
-rwxr-xr-x  1 root root 41 Feb 16  2004 /usr/bin/zless

$ cat /usr/bin/zless
#!/bin/sh
/bin/zcat "$@" | /usr/bin/less
```

Next...

- Quoting Mechanisms
- Positional (Command Line) Parameters
- File Descriptors
- Redirecting Standard Output and Standard Error
- Using `exec` to Redirect I/O
- Command Substitution

Quoting Mechanisms (1)

An unquoted backslash (\) is the escape character. It forces the shell to take the next character (except a NEWLINE) literally. A quoted NEWLINE is ignored. You can quote a backslash by preceding it with a backslash or enclosing it between single quotation marks.

```
$ var=mark
$ echo $var and \$var
mark and $var
```

Quoting Mechanisms (2)

Single quotation marks (') prevent all expansion, that is, the shell interprets all characters enclosed between single quotation marks literally.

```
$ var=sam
$ echo $var and '$var'
sam and $var
```

Quoting Mechanisms (3)

Double quotation marks (") prevent expansion except for enclosed dollar signs (**$**) and sometimes back-slashes. Within double quotation marks backslashes can only quote dollar signs, double quotation marks, other backslashes, and NEWLINEs.

```
$ var=zach
$ echo $var and "$var" and "\$var"
zach and zach and $var
```

Single versus Double Quotation Marks

```
$ echo $PWD
/home/mark

$ alias dirA="echo Working directory is $PWD"
$ alias dirA
alias dirA='echo Working directory is /home/mark'

$ alias dirB='echo Working directory is $PWD'
$ alias dirB
alias dirB='echo Working directory is $PWD'

$ cd cars
$ dirA
Working directory is /home/mark
$ dirB
Working directory is /home/mark/cars
```

Positional Parameters (1)

```
$ cat bb
echo $# parameters
echo 1: $1
echo 2: $2
echo 3: $3

$ bb aaa bbb ccc
3 parameters
1: aaa
2: bbb
3: ccc

$ bb "aa a" bb c
3 parameters
1: aa a
2: bb
3: c
```

Positional Parameters (2)

`set` assigns values to positional parameters.

```
$ cat ee1
set apple pear cranberry pumpkin
echo $# parameters
echo 1: $1
echo 2: $2
echo 3: $3

$ ee1
4 parameters
1: apple
2: pear
3: cranberry
```

Positional Parameters (3)

```
$ cat ee2
set $*
echo $# parameters
echo 1: $1
echo 2: $2
echo 3: $3

$ ee2 earth water air fire
4 parameters
1: earth
2: water
3: air
```

"$*" Versus "$@" (1)

"$*" yields a single argument with SPACEs between elements

```
$ cat bb1
set "$*"
echo $# parameters
echo 1: $1
echo 2: $2
echo 3: $3

$ bb1 "a a" b c
1 parameters
1: a a b c
2:
3:
```

"$*" Versus "$@" (2)

"$@" yields multiple arguments

```
$ cat bb2
set "$@"
echo $# parameters
echo 1: $1
echo 2: $2
echo 3: $3

$ bb2 "a a" b c
3 parameters
1: a a
2: b
3: c
```

File Descriptors

The three standard file descriptors are

- Standard Input (fd 0)
- Standard Output (fd 1)
- Standard Error (fd 2)

Redirecting Standard Output and Standard Error (1)

```
$ cat filea
This is file A.

$ cat fileb
cat: fileb: No such file or directory

$ cat filea fileb
This is file A.
cat: fileb: No such file or directory
```

Redirecting Standard Output and Standard Error (2)

The > token implies 1>.

```
$ cat filea fileb > holdstdout
cat: fileb: No such file or directory

$ cat holdstdout
This is file A.

$ cat filea fileb 2> holderrout
This is file A.

$ cat holderrout
cat: fileb: No such file or directory
```

Duplicating File Descriptors

You can duplicate an output file descriptor using the `>&` token. The `1` in `1>&2` is optional.

```
$ cat filea fileb 2> holderrout 1>&2

$ cat holderrout
This is file A.
cat: fileb: No such file or directory

$ cat filea fileb 1> holdstdout 2>&1

$ cat holdstdout
This is file A.
cat: fileb: No such file or directory
```

Within a shell script you can use the following construct to send an error message to standard error:

```
echo "You must supply at least one argument." 1>&2
```

Using /dev/tty

Redirect standard output to go to the user's screen:

```
echo -n "Enter your name: " > /dev/tty
```

Redirect standard input to come from the user's keyboard:

```
read name < /dev/tty
```

Using `exec` to Redirect Input/Output From Within a Shell Script

Use `exec` to redirect standard input from within a shell script:

```
exec < infile
```

Use `exec` to redirect standard output and error from within a shell script:

```
exec > outfile
exec 2> errfile
```

Redirect standard output for the rest of the shell script to go to the user's screen:

```
exec > /dev/tty
```

Setting Up Additional File Descriptors

You can set up file descriptors in addition to standard input, standard output, and standard error:

```
$ cat desc3
exec 3> 3out
echo hi there >&3

$ desc3

$ cat 3out
hi there
```

Command Substitution

```
program=$(basename $0)

program=`basename $0`

program=$(echo $0 | sed 's|.*/||')

manpath=$(man --path | tr : '\040')
```

Control Structures

Control structures alter the order of execution of commands within a shell script.

- `if`
- `if...then...else`
- `for...in`
- `for`
- `while`
- `select`

if

```
$ cat if1
echo -n "word 1: "
read word1
echo -n "word 2: "
read word2

if test "$word1" = "$word2"
    then
        echo "Match"
fi
echo "End of program."

$ if1
word 1: panda
word 2: panda
Match
End of program.
```

The `bash -x` Option (Debugging)

You can also use `set -x` in an interactive shell or script. Use `set +x` to turn off debugging.

```
$ bash -x if1
+ echo -n 'word 1: '
word 1: + read word1
dog
+ echo -n 'word 2: '
word 2: + read word2
pony
+ test dog = pony
+ echo 'End of program.'
End of program.
```

`echo` Statements (Debugging)

Sometimes the easiest way to debug
is to put a few `echo` statements in a script...

Using `if` to Check Arguments

You can use `[ ]` in place of `test`. Use = to compare strings and `-eq` to compare numbers.

```
$ cat chkargs
if [ $# -eq 0 ]
    then
        echo "Supply at least one argument." 1>&2
        exit 1
fi
echo "Program running."

$ chkargs
Supply at least one argument.

$ chkargs abc
Program running.
```

if...then...else

Double hyphens allow you to specify an argument that begins with a hyphen.

```
$ cat out
if [ $# -eq 0 ]
    then
        echo "Usage: out [-v] filenames..." 1>&2
        exit 1
fi
if [ "$1" = "-v" ]
    then
        shift
        less -- "$@"
    else
        cat -- "$@"
fi
```

rpt1 Script (1)

```
$ cat rpt1
#!/bin/bash

# Generates a report based on today's date

# Function to display errors
errrpt ()
{
echo -e "\n\nErrors from run of $sname:\n" > /dev/tty
cat $errfile > /dev/tty
}

# Ignore a suspend (SIGSTP) signal
trap '' 20

# Rename output file and exit if run is interrupted
# with SIGHUP, SIGINT, SIGQUIT, SIGTERM
trap 'mv $outfile ${outfile}ERROR; exit 1' 1 2 3 15

# Run error report on any exit
trap 'errrpt' 0
```

rpt1 Script (2)

```
# Set up program name to display with error messages
sname=$(basename $0)

# Get login name of user
logname=$(who am i | cut --delimiter=\  --fields=1)

# Get date in YYMMDD format
dat=$(date +%y%m%d)

infile=${dat}77A
outfile=${dat}77B
errfile=${dat}77.err

# Set up standard input, standard output, and
# standard error for the rest of the script
exec < $infile
exec > $outfile 2> $errfile
```

`rpt1` Script (3)

```
if [ ! -f $infile ]; then
        echo "$sname: File $infile not available." 1>&2
        exit 1
        fi

if [ ! -w . ]; then
        echo "$sname: No write permission to \
working directory." 1>&2
        exit 1
        fi

# Send prompt to user's terminal
echo -n "Enter your name: " > /dev/tty

# Read response from user's terminal
read name < /dev/tty
```

rpt1 Script (4)

```
bad command to generate an error

# Let the user know what is going on.
echo "Running report..." > /dev/tty

# Pause here so we can demonstrate trap
sleep 5

# Put user's name at top of report
echo "Report run by $name (logged in as $logname)."

# Following is the report. It gets its input
# from $infile and sends its output to $outfile
gawk '{print NR, $0}'
```

rpt1 Script (5)

```
$ ls
04111577A  rpt1

$ cat 04111577A
This is my report
This is my report
This is my report

$ rpt1
Enter your name: Mark
Running report...


Errors from run of rpt1:

./rpt1: line 57: bad: command not found
```

rpt1 Script (6)

```
$ ls
04111577.err  04111577A  04111577B  rpt1

$ cat *err
./rpt1: line 57: bad: command not found

$ cat *B
Report run by Mark (logged in as mark).
1 This is my report
2 This is my report
3 This is my report
```

`lnks` Script (1)

Finds hard links to its first argument in the working directory tree.
A second argument causes `lnks` to search in that directory's tree.

```
$ cat lnks
#!/bin/bash
# Identify links to a file
# Usage: lnks file [directory]

if [ $# -eq 0 -o $# -gt 2 ]; then
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
if [ -d "$1" ]; then
    echo "First argument cannot be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
else
    file="$1"
fi
```

`lnks` Script (2)

```
if [ $# -eq 1 ]; then
    directory="."
elif [ -d "$2" ]; then
    directory="$2"
else
    echo "Second argument must be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
# Check to make sure file exists and is a regular file:
if [ ! -f "$file" ]; then
    echo "lnks: $file not found or special file" 1>&2
    exit 1
fi
```

`lnks` Script (3)

```
# Check link count on file
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
    echo "lnks: no other link to $file" 1>&2
    exit 0
fi

# Get the inode of the given file
set $(ls -i "$file")

inode=$1

# Find and print the files with that inode number
echo "lnks: using find to search for links..." 1>&2
find "$directory" -xdev -inum $inode -print
```

for...in (1)

```
$ cat fruit
for fruit in apples oranges pears bananas
do
    echo "$fruit"
done
echo "Task complete."

$ fruit
apples
oranges
pears
bananas
Task complete.
```

`for...in` (2)

Lists directories in the working directory.

```
$ cat dirfiles
for i in *
do
    if [ -d "$i" ]
        then
            echo "$i"
    fi
done
```

for

Similar to for...in but uses command line arguments for the list.

```
$ cat for_test
for arg
do
    echo "$arg"
done

$ for_test panda horse dog lama
panda
horse
dog
lama
```

while

Note the format of a numeric expression and assignment.

```
$ cat count
#!/bin/bash
number=0
while [ "$number" -lt 10 ]
    do
        echo -n "$number"
        ((number=$number + 1))
    done
echo

$ count
0123456789
```

`break` and `continue`

```
$ cat brk
for index in 1 2 3 4 5 6 7 8 9 10
    do
        if [ $index -le 3 ] ; then
            echo "continue"
            continue
        fi
        echo $index
        if [ $index -ge 8 ] ; then
            echo "break"
            break
        fi
    done
$ brk
continue
continue
continue
4...8 (whoops, no room.)
break
```

select (1)

```
$ cat fruit2
#!/bin/bash
PS3="Choose your favorite fruit from these: "
select FRUIT in apple banana blueberry kiwi \
orange watermelon STOP
do
    if [ "$FRUIT" == "" ]; then
        echo -e "Invalid entry.\n"
        continue
    elif [ $FRUIT = STOP ]; then
        echo "Thanks for playing!"
        break
    fi
    echo "You chose $FRUIT as your favorite."
    echo -e "That is choice number $REPLY.\n"
done
```

select (2)

```
$ fruit2
1) apple        3) blueberry    5) orange       7) STOP
2) banana       4) kiwi         6) watermelon
Choose your favorite fruit from these: 5
You chose orange as your favorite.
That is choice number 5.

Choose your favorite fruit from these: 99
Invalid entry.

Choose your favorite fruit from these: 7
Thanks for playing!
```

COLUMNS establishes menu width and
LINES establishes height.

Here Document

```
$ cat birthday
grep -i "$1" <<+
Alex    June 22
Barbara February 3
Darlene May 8
Helen   March 13
Jenny   January 23
Nancy   June 26
+

$ birthday Mark

$ birthday Nancy
Nancy   June 26
```

bundle (1)

```
$ cat bundle
#!/bin/bash
# bundle:  group files into distribution package
# Thanks to Brian W. Kernighan and Rob Pike,
# The Unix Programming Environment

echo "# To unbundle, bash this file"
for i
do
    echo "echo $i 1>&2"
    echo "cat >$i <<'End of $i'"
    cat $i
    echo "End of $i"
done
```

bundle (2)

```
$ cat file1
This is a file.
It contains two lines.

$ cat file2
This is another file.
It contains
three lines.
```

bundle (3)

```
$ bash -x bundle file1 file2
+ echo '# To unbundle, bash this file'
# To unbundle, bash this file

+ echo 'echo file1 1>&2'
echo file1 1>&2
+ echo 'cat >file1 <<'\''End of file1'\'''
cat >file1 <<'End of file1'
+ cat file1
This is a file.
It contains two lines.
+ echo 'End of file1'
End of file1
...
```

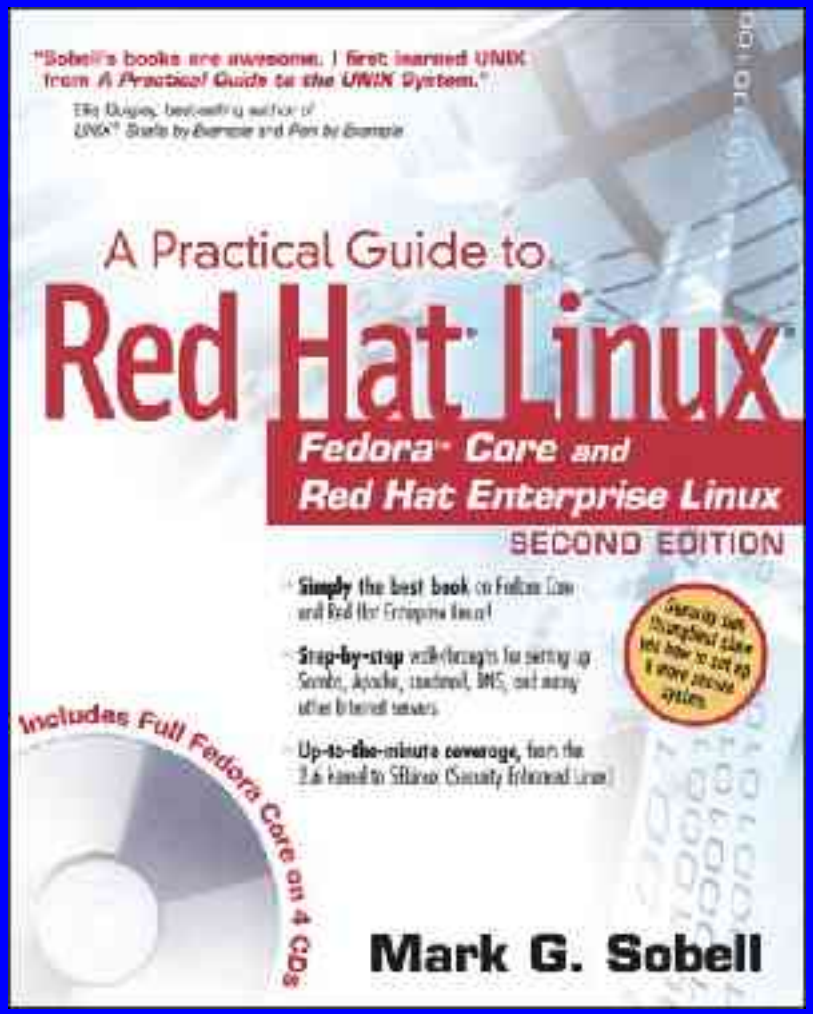
bundle (4)

```
$ bundle file1 file2 > bothfiles

$ cat bothfiles
# To unbundle, bash this file
echo file1 1>&2
cat >file1 <<'End of file1'
This is a file.
It contains two lines.
End of file1
echo file2 1>&2
cat >file2 <<'End of file2'
This is another file.
It contains
three lines.
End of file2

$ bash bothfiles
file1
file2
```

A Practical Guide to Red Hat Linux...



A Practical Guide to Red Hat Linux...

A Practical Guide to Red Hat Linux...

PART IV System Administration

11 System Administration: Core Concepts
12 Files, Directories, and Filesystems
13 Downloading and Installing Software
14 Printing with CUPS
15 Rebuilding the Linux Kernel
16 Administration Tasks
17 Configuring a LAN

A Practical Guide to Red Hat Linux...

PART V Using Clients and Setting Up Servers

18 OpenSSH: Secure Network Communication

19 FTP: Transferring Files Across a Network

20 sendmail: Setting Up Mail Clients, Servers, and More

21 NIS: Network Information Service

22 NFS: Sharing Filesystems

23 Samba: Integrating Linux and Windows

24 DNS/BIND: Tracking Domain Names and Addresses

25 iptables: Setting Up a Firewall

26 Apache (httpd): Setting Up a Web Server

A Practical Guide to Red Hat Linux...

Next Summer: A Practical Guide to Linux Commands, Editors, and Shell Programming

Next Summer: A Practical Guide to Linux Commands, Editors, and Shell Programming

Part IV. programming tools

9 programming tools

10 bash III (programming)

11 awk

12 sed

Part V. command reference

(covers about 80 utilities in man page style with extensive examples and notes)

Answer

Question:

Use a regular expression to convert from the first format to the second:

```
last name, first name initial

->

first name initial last name
```

Answer:

```
:1,$s/\([^,]*\), \(.*\)/\2 \1/
```