LING 408/508: Computational Techniques for Linguists

Lecture 11

Today's Topics

- Let's practice learning how to write shell scripts ...

Exercise 1: print arguments to a script

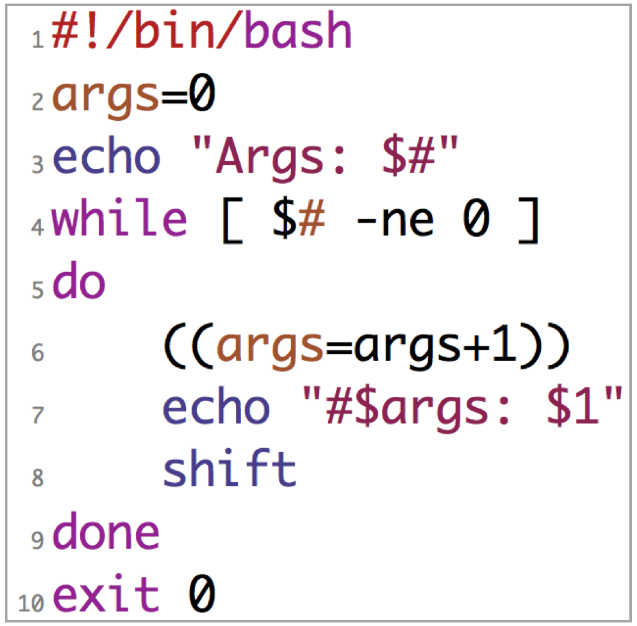
- Write a bash shell script that simple accepts command line arguments and prints out the number of arguments, and each argument numbered on a separate line
- Example:
  - **`bash args.sh a b c`**

```
Args: 3
#1: a
#2: b
#3: c
```

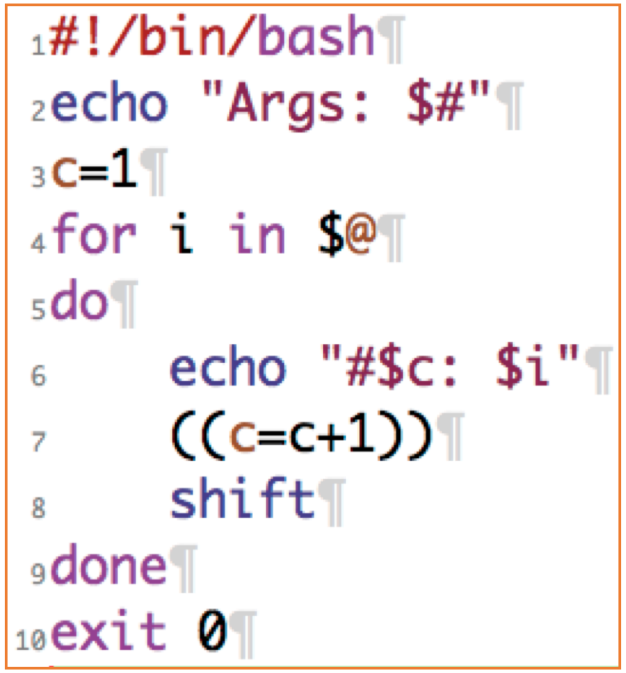
What you need to know to solve this:
1. $#
2. $1
3. shift
4. store a variable
5. increment a value by 1

Exercise 1: print arguments to a script

Many different ways to write the solution…
`args1.sh` vs. `args2.sh`

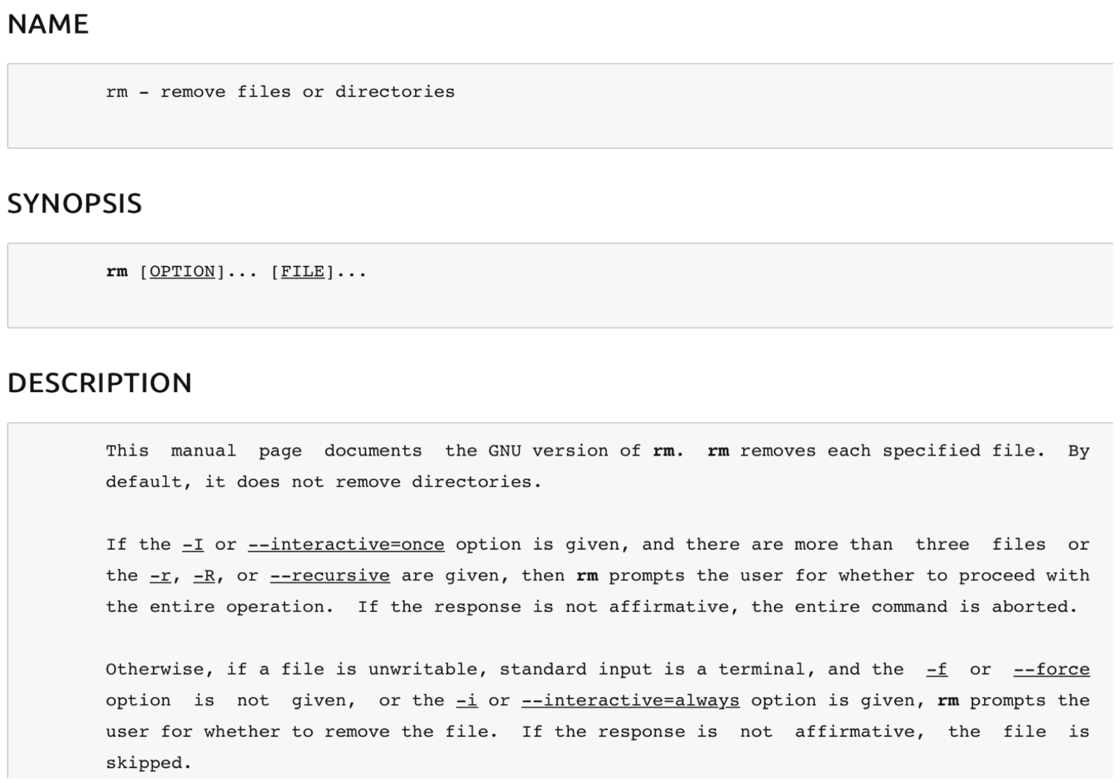


```bash
#!/bin/bash
args=0
echo "Args: $#"
while [ $# -ne 0 ]
do
    ((args=args+1))
    echo "#$args: $1"
    shift
done
exit 0
```



```bash
#!/bin/bash
echo "Args: $#"
c=1
for i in $@
do
    echo "#$c: $i"
    ((c=c+1))
    shift
done
exit 0
```

Exercise 2: deleting files



NAME

```
       rm - remove files or directories
```

SYNOPSIS

```
       rm [OPTION]... [FILE]...
```

DESCRIPTION

```
       This  manual  page  documents  the GNU version of rm.  rm removes each specified file.  By
       default, it does not remove directories.

       If the -I or --interactive=once option is given, and there are more than  three  files  or
       the -r, -R, or --recursive are given, then rm prompts the user for whether to proceed with
       the entire operation.  If the response is not affirmative, the entire command is aborted.

       Otherwise, if a file is unwritable, standard input is a terminal, and the  -f  or  --force
       option  is  not  given,  or the -i or --interactive=always option is given, rm prompts the
       user for whether to remove the file.  If the response is  not  affirmative,  the  file  is
       skipped.
```

Exercise 2: deleting files

Remove File/Directory

rm	removes a file (*dangerous!*)
rm –d	removes a directory (*empty*)
rm –r	recursive remove (*extreme danger!*)
rm –rf	forced recursive remove (!!!)

- Examples:
  - **`touch file.txt`**
  - `rm file.txt` (*you have default write permission*)
  - `touch protected.txt`
  - `chmod u-w protected.txt` (*u = user, -w = remove write permission*)
  - `rm protected.txt`

```
override r--r--r--  sandiway/staff for protected.txt?
```

  - `rm –f protected.txt` (*no interaction*: *forced removal*)
  - `rm –i file.txt` (*ask it to ask you for confirmation*)

```
remove file.txt?
```

Exercise 2: deleting files

At least two reasons:
1. another computer
2. shell scripts

best used in interactive shell

- *can put alias shortcut in Terminal startup ~/.bash_profile (MacOS) or ~/.bashrc*

- `alias rm="rm -i"` not recursively expanded (*considered dangerous*: *why*?)
- `alias` (*list defined aliases*)
- `unalias rm` (*remove alias*)
- Aliases don't work in shell scripts (`rm.sh`):

```
#!/bin/bash
if [ $# -ne 1 ]; then
     echo "usage: filename"
     exit 1
fi
touch $1
rm $1
```

`rm -i` won't be called!

define a function in ~/.bash_profile
(absolute path: otherwise recursive)

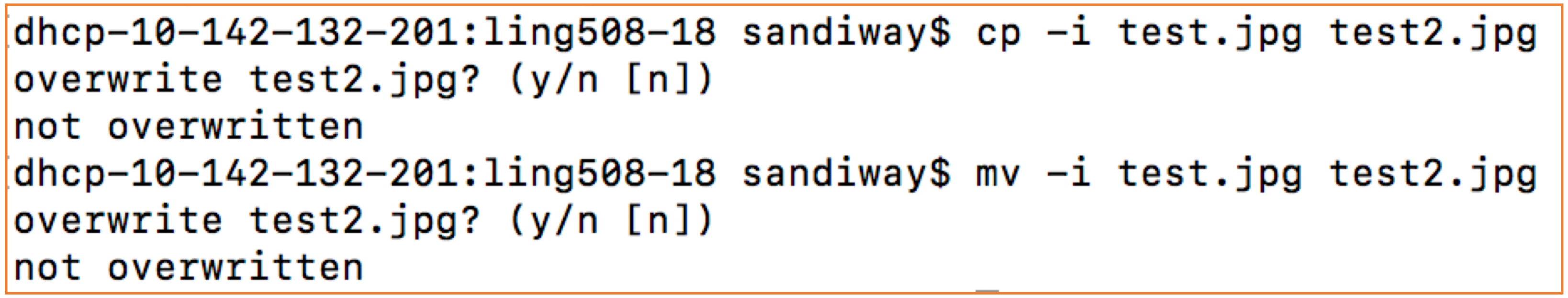
```
rm () {
    /bin/rm -i "$@"
}
export -f rm
```

Other commands with -i

- `-i` (interactive confirm option)

  before overwriting a file

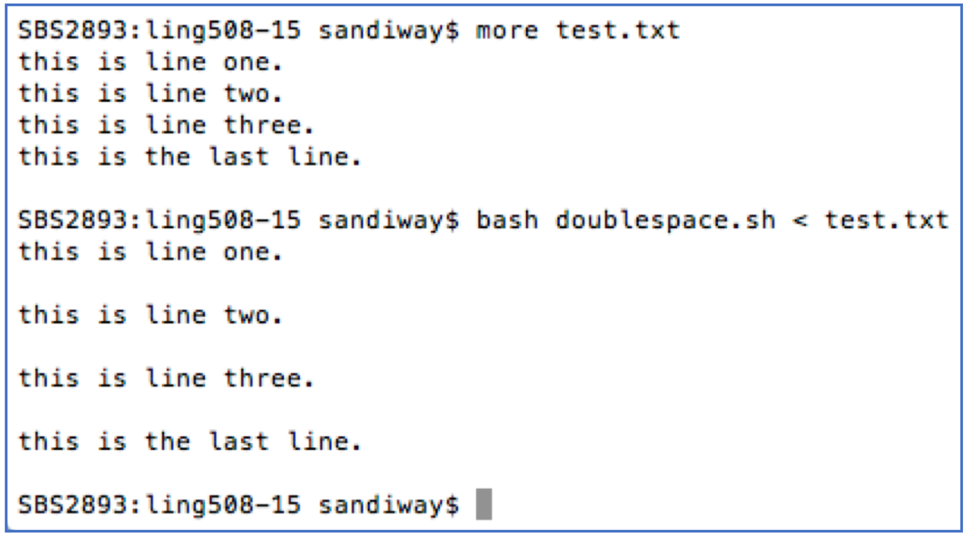
  - `mv -i` *rename file*
  - `cp -i` *copy file*



```
dhcp-10-142-132-201:ling508-18 sandiway$ cp -i test.jpg test2.jpg
overwrite test2.jpg? (y/n [n])
not overwritten
dhcp-10-142-132-201:ling508-18 sandiway$ mv -i test.jpg test2.jpg
overwrite test2.jpg? (y/n [n])
not overwritten
```

Exercise 3: double-spacing a text file

- Write a script that reads each line of a file, then writes the line back out, but with an extra blank line following. This has the effect of *double-spacing* the file.

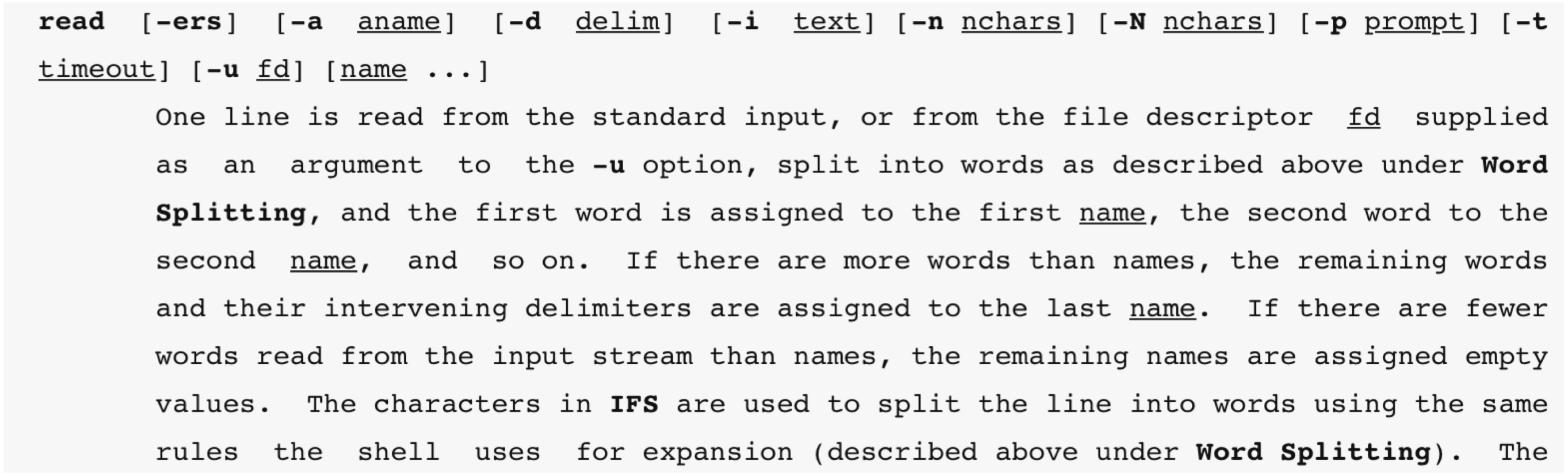


```
SBS2893:ling508-15 sandiway$ more test.txt
this is line one.
this is line two.
this is line three.
this is the last line.

SBS2893:ling508-15 sandiway$ bash doublespace.sh < test.txt
this is line one.

this is line two.

this is line three.

this is the last line.

SBS2893:ling508-15 sandiway$ 
```

What you need to know to solve this:

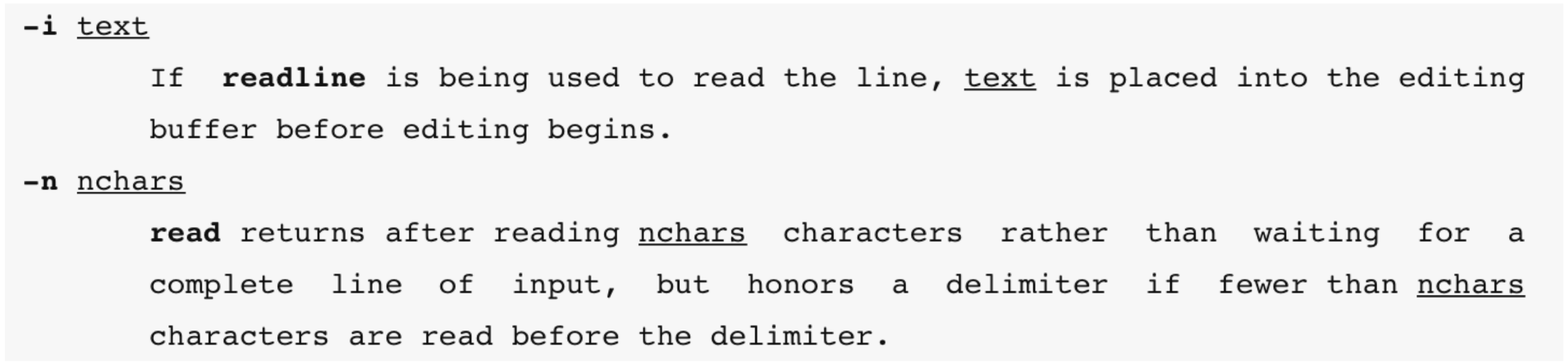
1. read
2. test [[…]]
3. while loop

Exercise 3: double-spacing a text file



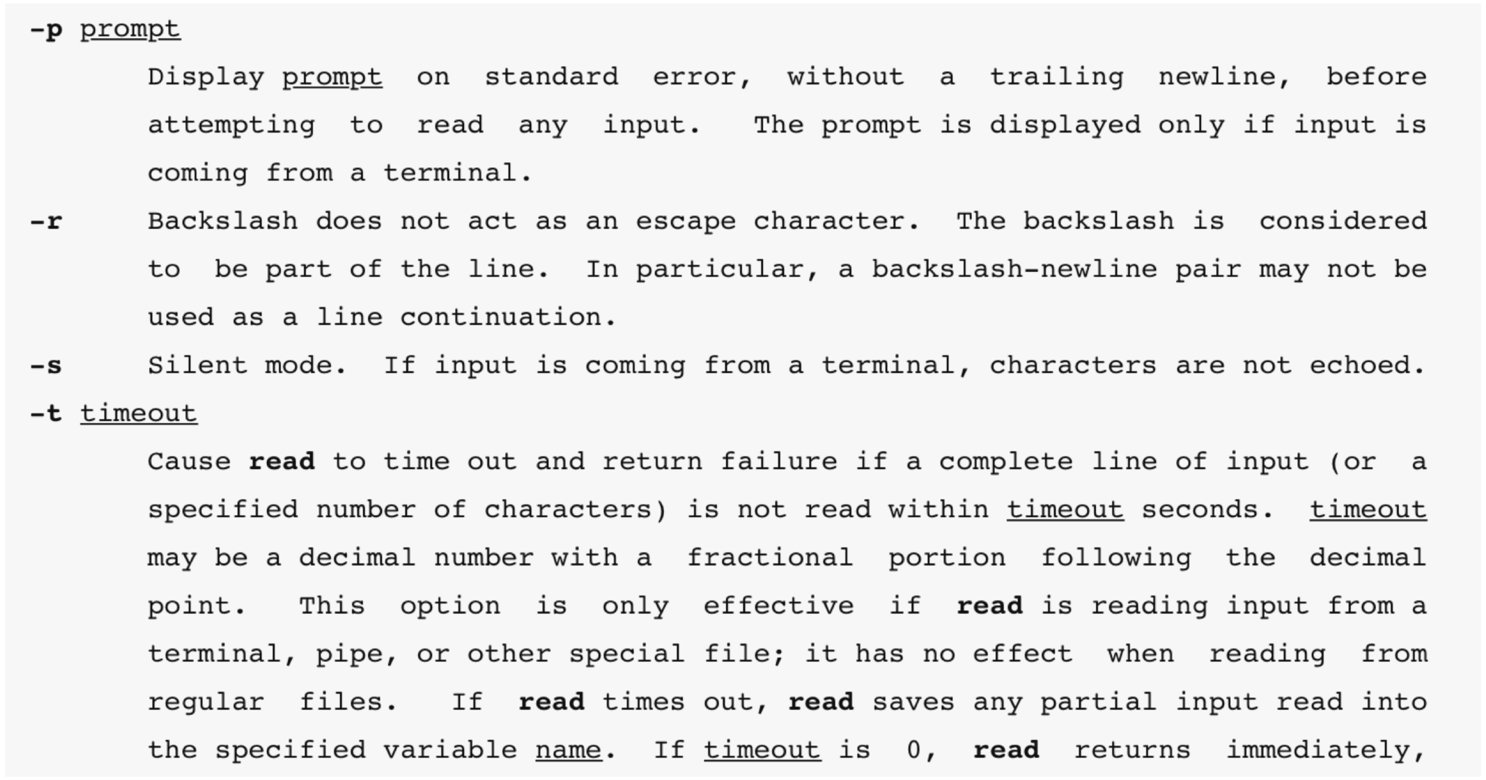
```
read  [-ers]  [-a  aname]  [-d  delim]  [-i  text] [-n nchars] [-N nchars] [-p prompt] [-t
timeout] [-u fd] [name ...]
       One line is read from the standard input, or from the file descriptor  fd  supplied
       as  an  argument  to  the -u option, split into words as described above under Word
       Splitting, and the first word is assigned to the first name, the second word to the
       second  name,  and  so on.  If there are more words than names, the remaining words
       and their intervening delimiters are assigned to the last name.  If there are fewer
       words read from the input stream than names, the remaining names are assigned empty
       values.  The characters in IFS are used to split the line into words using the same
       rules  the  shell  uses  for expansion (described above under Word Splitting).  The
```

Exercise 3: double-spacing a text file



```
-i text
       If  readline is being used to read the line, text is placed into the editing
       buffer before editing begins.
-n nchars
       read returns after reading nchars  characters  rather  than  waiting  for  a
       complete  line  of  input,  but  honors  a  delimiter  if  fewer than nchars
       characters are read before the delimiter.
```

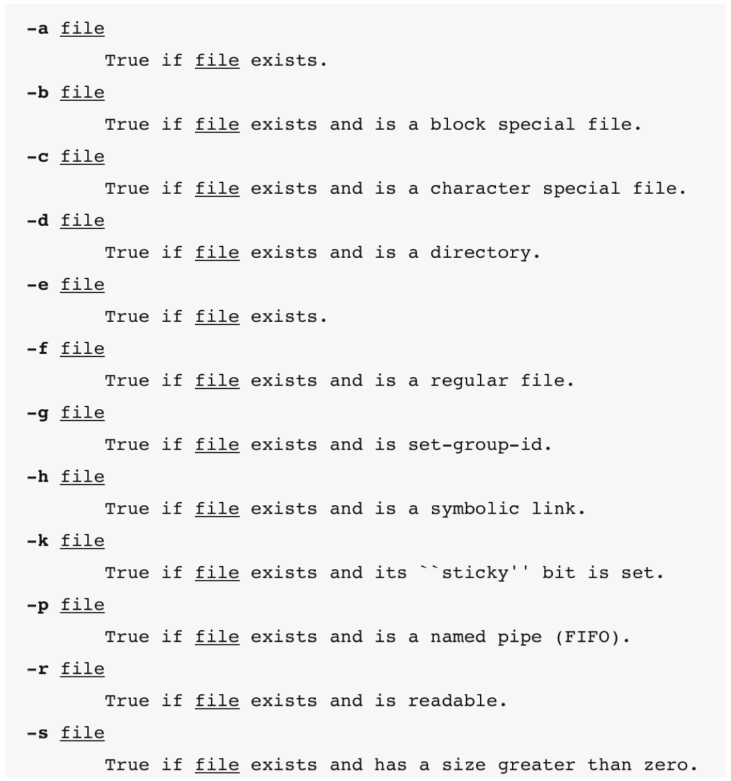
Exercise 3: double-spacing a text file



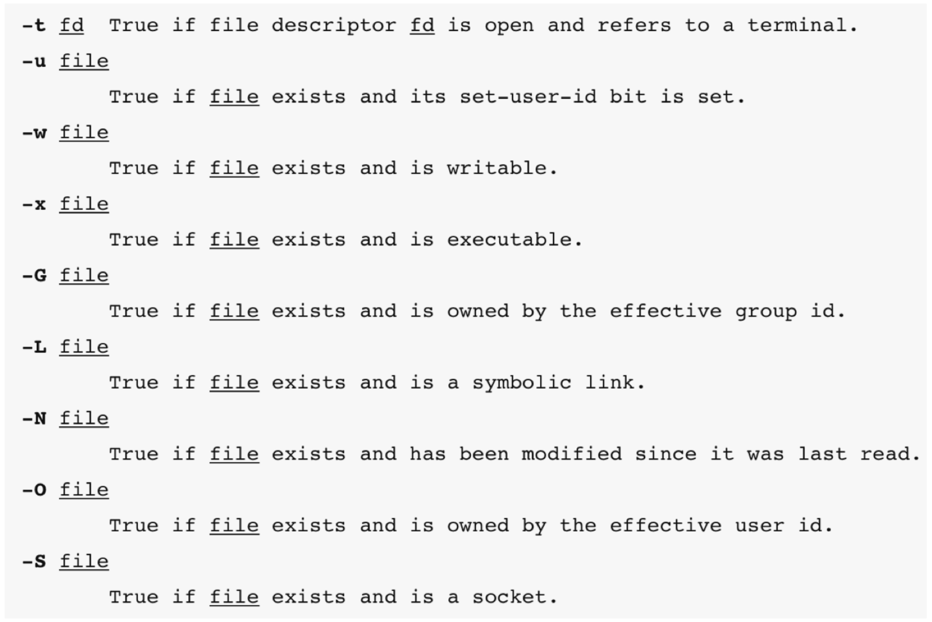
```
-p prompt
       Display prompt  on  standard  error,  without  a  trailing  newline,  before
       attempting  to  read  any  input.   The prompt is displayed only if input is
       coming from a terminal.
-r     Backslash does not act as an escape character.  The backslash is  considered
       to  be part of the line.  In particular, a backslash-newline pair may not be
       used as a line continuation.
-s     Silent mode.  If input is coming from a terminal, characters are not echoed.
-t timeout
       Cause read to time out and return failure if a complete line of input (or  a
       specified number of characters) is not read within timeout seconds.  timeout
       may be a decimal number with a  fractional  portion  following  the  decimal
       point.   This  option  is  only  effective  if  read is reading input from a
       terminal, pipe, or other special file; it has no effect  when  reading  from
       regular  files.   If  read times out, read saves any partial input read into
       the specified variable name.  If timeout is  0,  read  returns  immediately,
```

Exercise 3: double-spacing a text file

test [[..]]



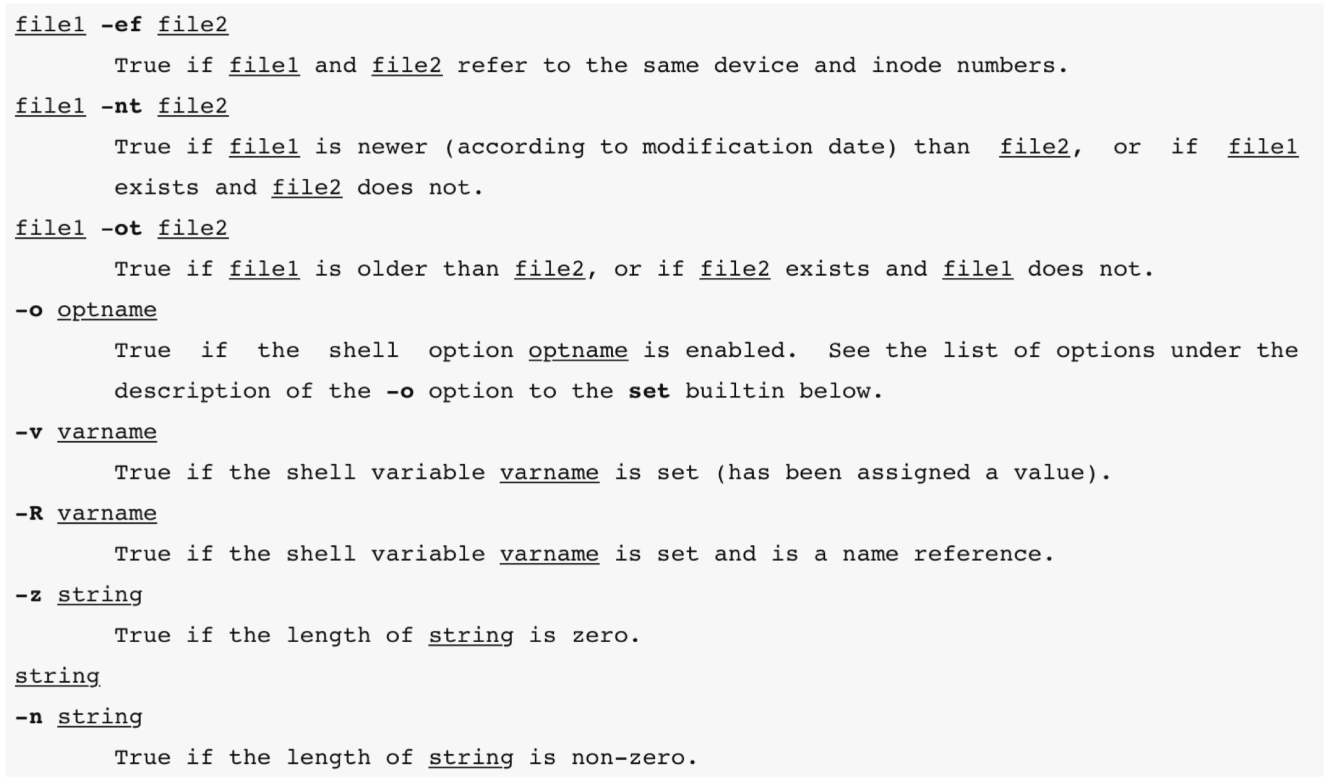
```
-a file
       True if file exists.
-b file
       True if file exists and is a block special file.
-c file
       True if file exists and is a character special file.
-d file
       True if file exists and is a directory.
-e file
       True if file exists.
-f file
       True if file exists and is a regular file.
-g file
       True if file exists and is set-group-id.
-h file
       True if file exists and is a symbolic link.
-k file
       True if file exists and its ``sticky'' bit is set.
-p file
       True if file exists and is a named pipe (FIFO).
-r file
       True if file exists and is readable.
-s file
       True if file exists and has a size greater than zero.
```



```
-t fd  True if file descriptor fd is open and refers to a terminal.
-u file
       True if file exists and its set-user-id bit is set.
-w file
       True if file exists and is writable.
-x file
       True if file exists and is executable.
-G file
       True if file exists and is owned by the effective group id.
-L file
       True if file exists and is a symbolic link.
-N file
       True if file exists and has been modified since it was last read.
-O file
       True if file exists and is owned by the effective user id.
-S file
       True if file exists and is a socket.
```

Exercise 3: double-spacing a text file

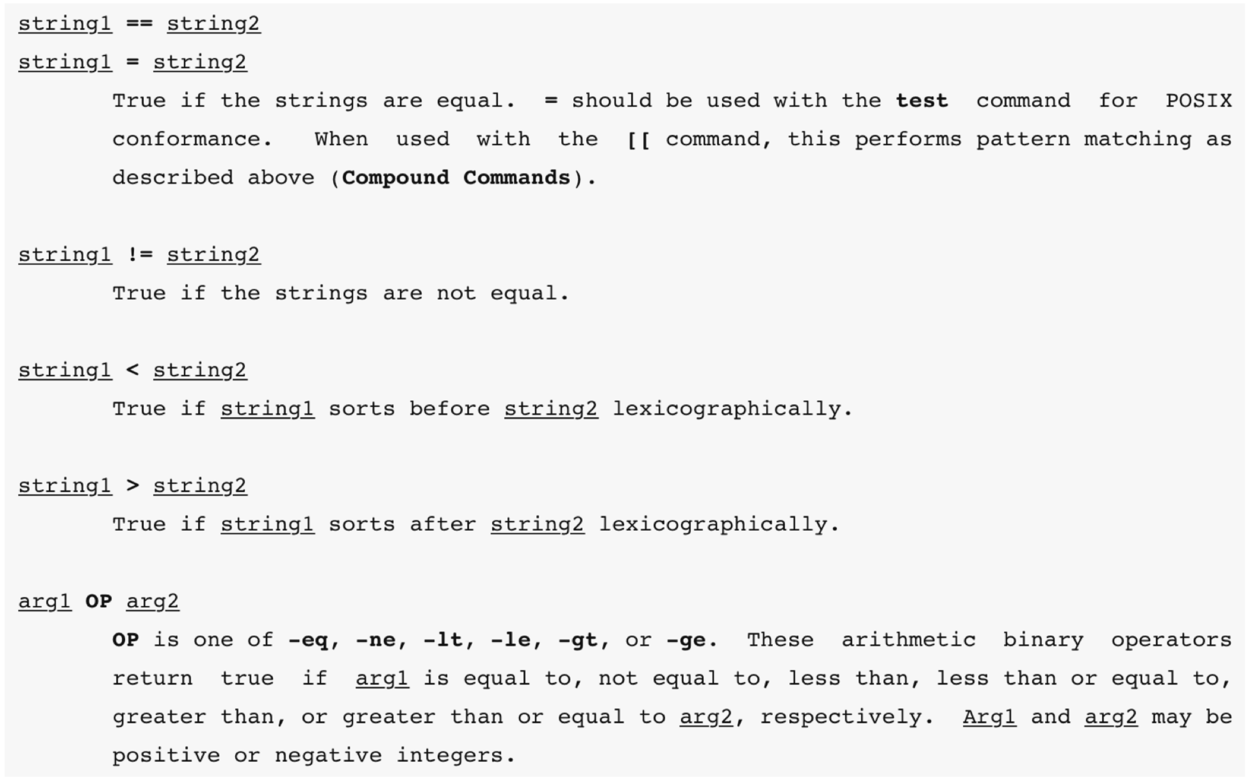
`test [[ .. ]]`



```
file1 -ef file2
       True if file1 and file2 refer to the same device and inode numbers.
file1 -nt file2
       True if file1 is newer (according to modification date) than  file2,  or  if  file1
       exists and file2 does not.
file1 -ot file2
       True if file1 is older than file2, or if file2 exists and file1 does not.
-o optname
       True  if  the  shell  option optname is enabled.  See the list of options under the
       description of the -o option to the set builtin below.
-v varname
       True if the shell variable varname is set (has been assigned a value).
-R varname
       True if the shell variable varname is set and is a name reference.
-z string
       True if the length of string is zero.
string
-n string
       True if the length of string is non-zero.
```

Exercise 3: double-spacing a text file

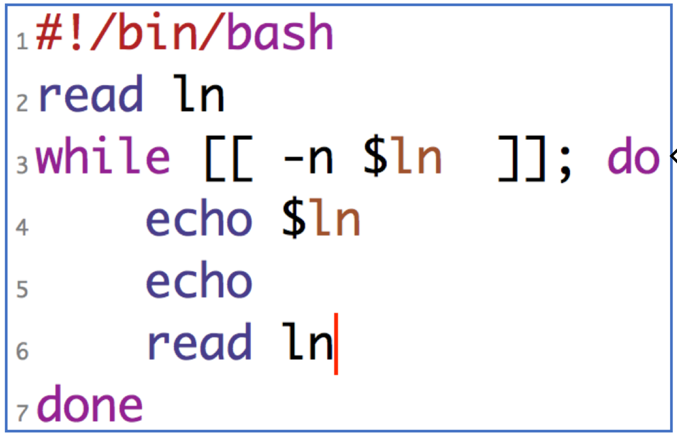
test [[..]]



```
string1 == string2
string1 = string2
       True if the strings are equal.  = should be used with the test  command  for  POSIX
       conformance.   When  used  with  the  [[ command, this performs pattern matching as
       described above (Compound Commands).

string1 != string2
       True if the strings are not equal.

string1 < string2
       True if string1 sorts before string2 lexicographically.

string1 > string2
       True if string1 sorts after string2 lexicographically.

arg1 OP arg2
       OP is one of -eq, -ne, -lt, -le, -gt, or -ge.  These  arithmetic  binary  operators
       return  true  if  arg1 is equal to, not equal to, less than, less than or equal to,
       greater than, or greater than or equal to arg2, respectively.  Arg1 and arg2 may be
       positive or negative integers.
```

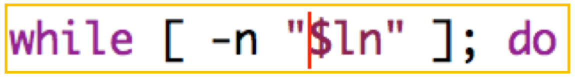
Exercise 3: double-spacing a text file

- *double-spacing* the file (`doublespace.sh`):



```bash
#!/bin/bash
read ln
while [[ -n $ln  ]]; do
      echo $ln
      echo
      read ln
done
```

−n = non-zero



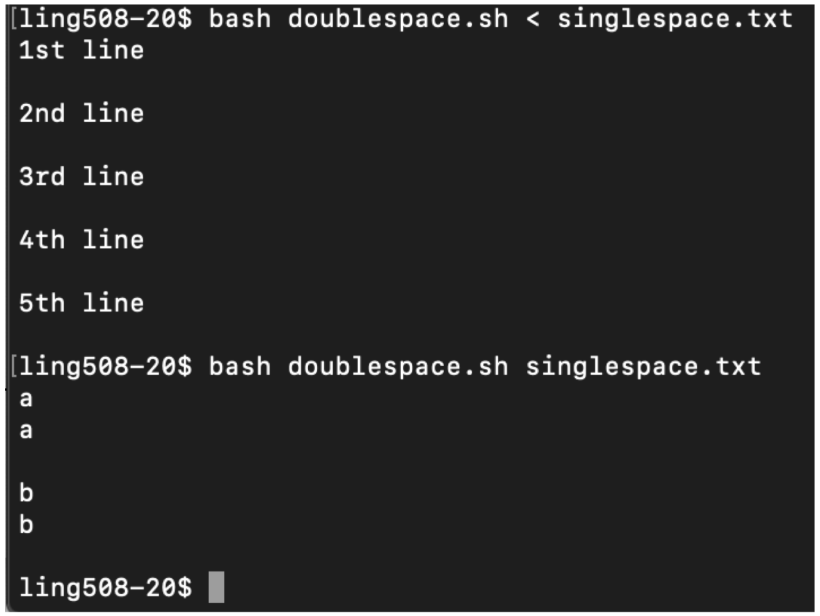
```bash
while [ -n "$ln" ]; do
```

also works

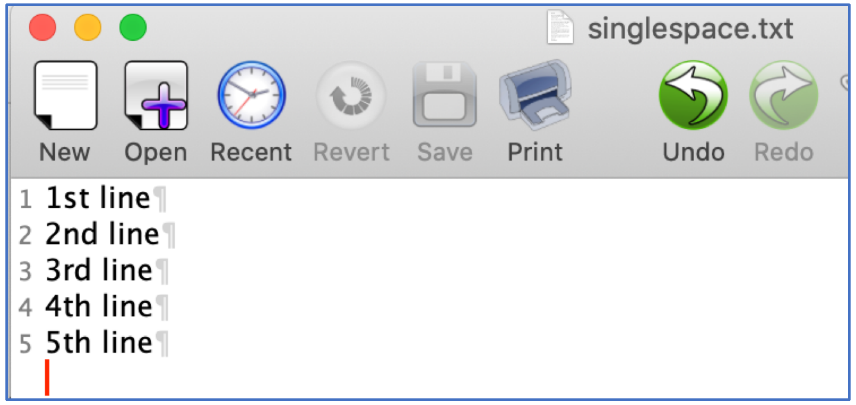
```
read –r
```

If this option is given, backslash does not act as an escape character.

Exercise 3: double-spacing a text file



```
ling508-20$ bash doublespace.sh < singlespace.txt
1st line

2nd line

3rd line

4th line

5th line

ling508-20$ bash doublespace.sh singlespace.txt
a
a

b
b

ling508-20$ 
```

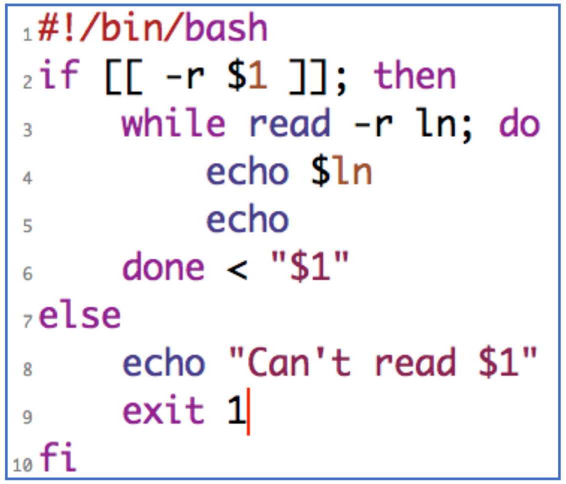


singlespace.txt

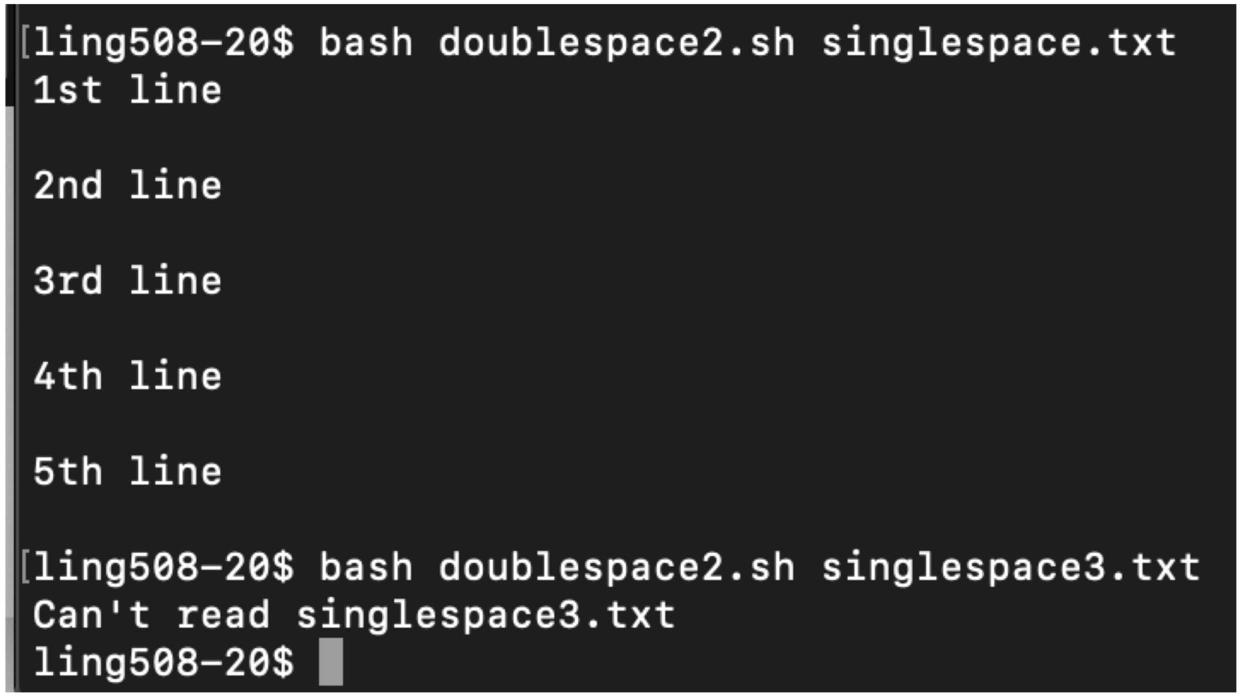
```
1st line
2nd line
3rd line
4th line
5th line
```

Exercise 3: double-spacing a text file

- *double-spacing* the file (`doublespace2.sh`):



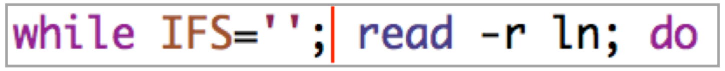
```
#!/bin/bash
if [[ -r $1 ]]; then
    while read -r ln; do
        echo $ln
        echo
    done < "$1"
else
    echo "Can't read $1"
    exit 1
fi
```



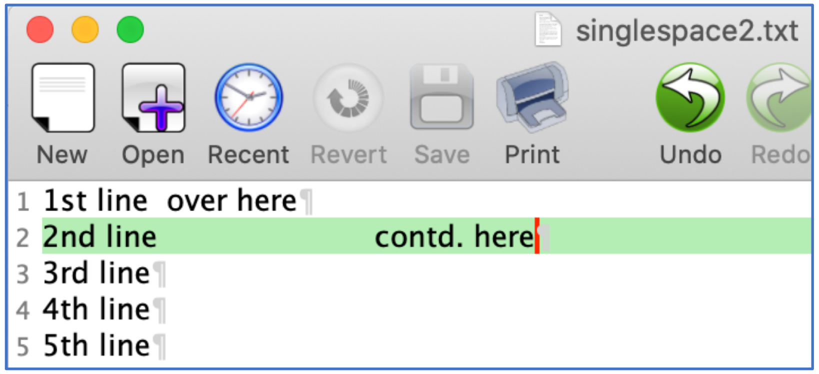
```
[ling508-20$ bash doublespace2.sh singlespace.txt
1st line

2nd line

3rd line

4th line

5th line

[ling508-20$ bash doublespace2.sh singlespace3.txt
Can't read singlespace3.txt
ling508-20$ 
```

Exercise 3: double-spacing a text file

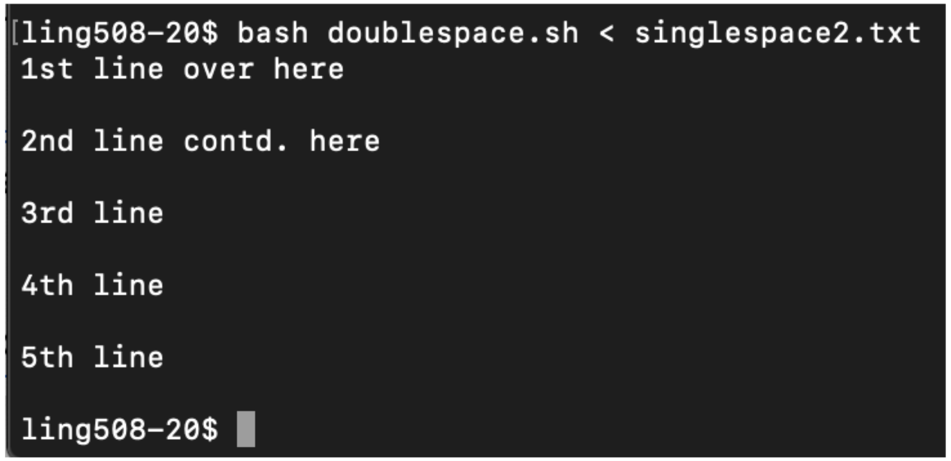
- **whitespace trim problem workaround:**



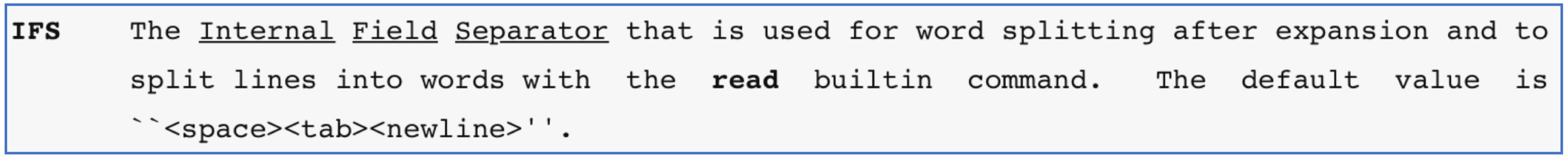
```
while IFS=''; read -r ln; do
```



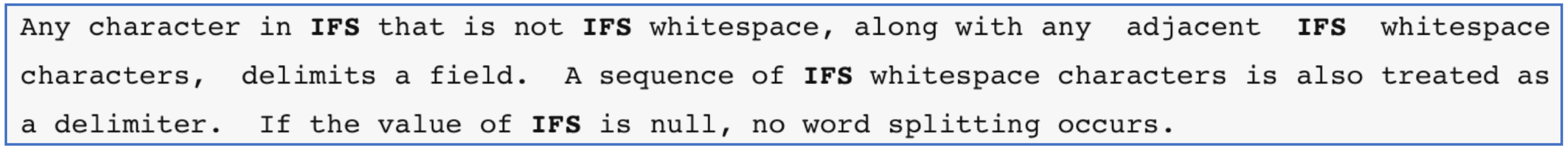
```
1st line  over here
2nd line                    contd. here
3rd line
4th line
5th line
```



```
ling508-20$ bash doublespace.sh < singlespace2.txt
1st line over here

2nd line contd. here

3rd line

4th line

5th line

ling508-20$
```



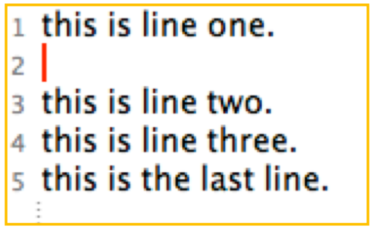
IFS The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the **read** builtin command. The default value is ``<space><tab><newline>''.



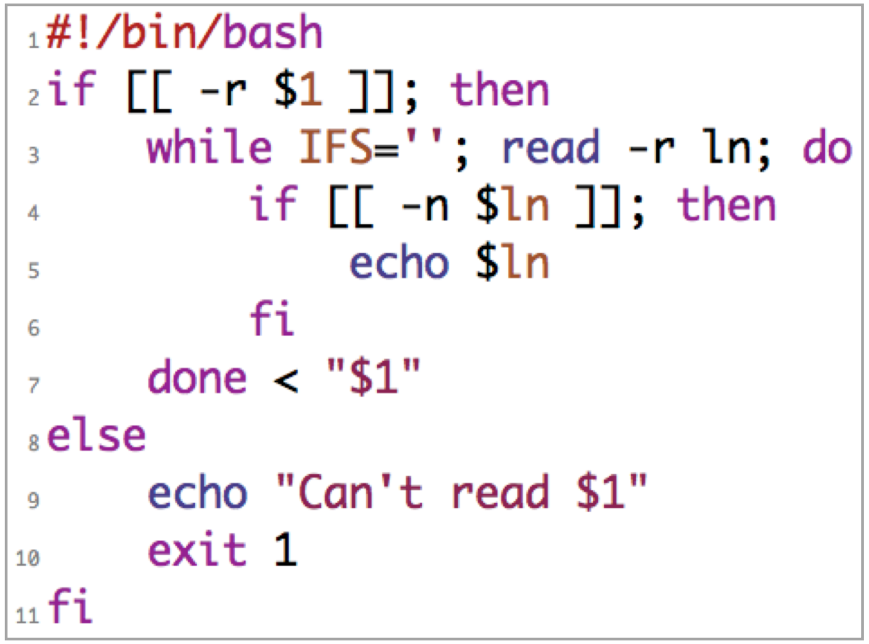
Any character in **IFS** that is not **IFS** whitespace, along with any adjacent **IFS** whitespace characters, delimits a field. A sequence of **IFS** whitespace characters is also treated as a delimiter. If the value of **IFS** is null, no word splitting occurs.

Exercise 4: all except blank lines

- **Changing the line spacing of a text file:**
- write a script to echo all lines of a file except for blank lines (`nonblank.sh`).



```
this is line one.

this is line two.
this is line three.
this is the last line.
```

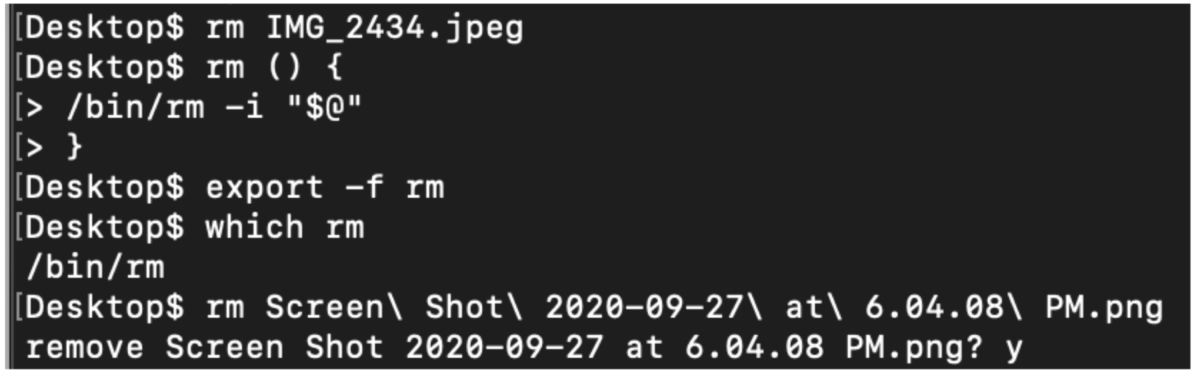


```bash
#!/bin/bash
if [[ -r $1 ]]; then
    while IFS=''; read -r ln; do
        if [[ -n $ln ]]; then
            echo $ln
        fi
    done < "$1"
else
    echo "Can't read $1"
    exit 1
fi
```

Useful tidbits

- Pre-programmed interaction:
  - (*here* document: inline file)

```
rm () {
    /bin/rm -i "$@"
}
export -f rm
```



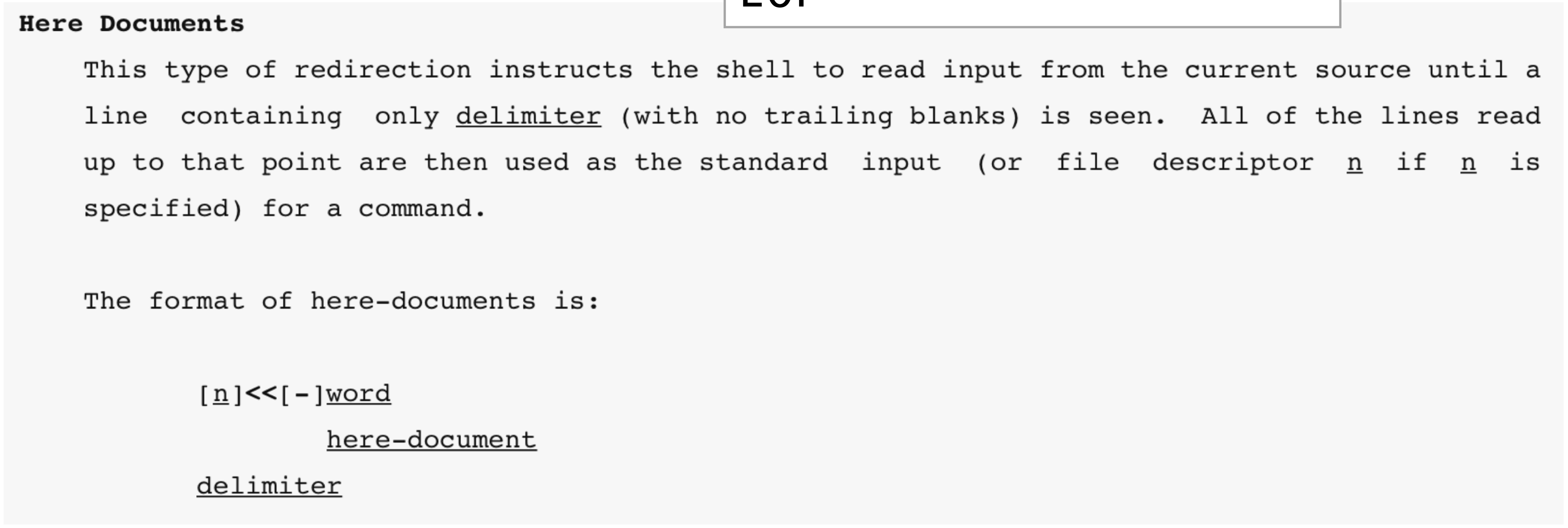
```
[Desktop$ rm IMG_2434.jpeg
[Desktop$ rm () {
[> /bin/rm -i "$@"
[> }
[Desktop$ export -f rm
[Desktop$ which rm
/bin/rm
[Desktop$ rm Screen\ Shot\ 2020-09-27\ at\ 6.04.08\ PM.png
remove Screen Shot 2020-09-27 at 6.04.08 PM.png? y
```

confirm.sh

```
#!/bin/bash
if [ $# -ne 1 ]; then
    echo "usage: filename"
    exit 1
fi
touch $1
rm $1
```

Useful tidbits

```
bash confirm.sh <<EOF
y
EOF
```

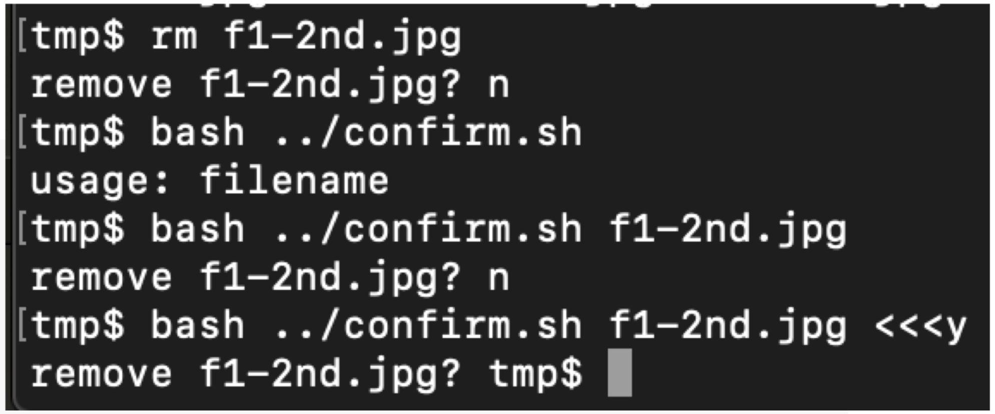


Here Documents

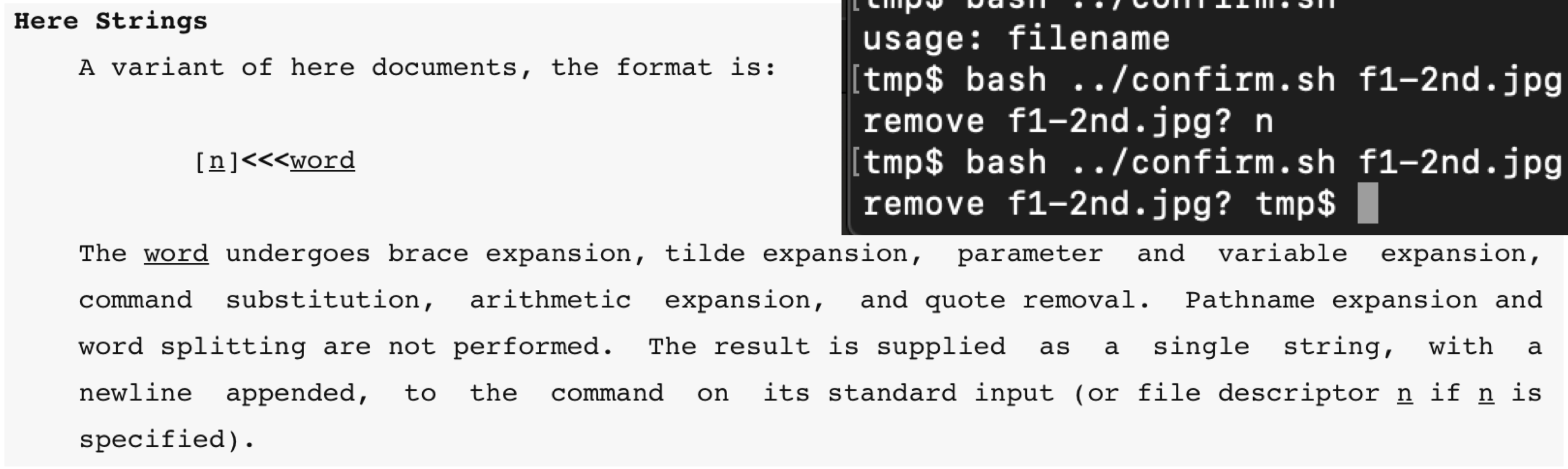
```
    This type of redirection instructs the shell to read input from the current source until a
    line  containing  only delimiter (with no trailing blanks) is seen.  All of the lines read
    up to that point are then used as the standard  input  (or  file  descriptor  n  if  n  is
    specified) for a command.

    The format of here-documents is:

           [n]<<[-]word
                   here-document
           delimiter
```

Useful tidbits

```
bash confirm.sh <<<y
```



```
tmp$ rm f1-2nd.jpg
remove f1-2nd.jpg? n
tmp$ bash ../confirm.sh
usage: filename
tmp$ bash ../confirm.sh f1-2nd.jpg
remove f1-2nd.jpg? n
tmp$ bash ../confirm.sh f1-2nd.jpg <<<y
remove f1-2nd.jpg? tmp$
```



Here Strings

A variant of here documents, the format is:

```
[n]<<<word
```

The word undergoes brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion and word splitting are not performed. The result is supplied as a single string, with a newline appended, to the command on its standard input (or file descriptor n if n is specified).