Seat belts and Airbags for bash

Michael Potter

March 25, 2025

Why bash?

- Simple to get started.
- Actively developed and ported.
- Includes advanced features.
- Allows piping commands together.

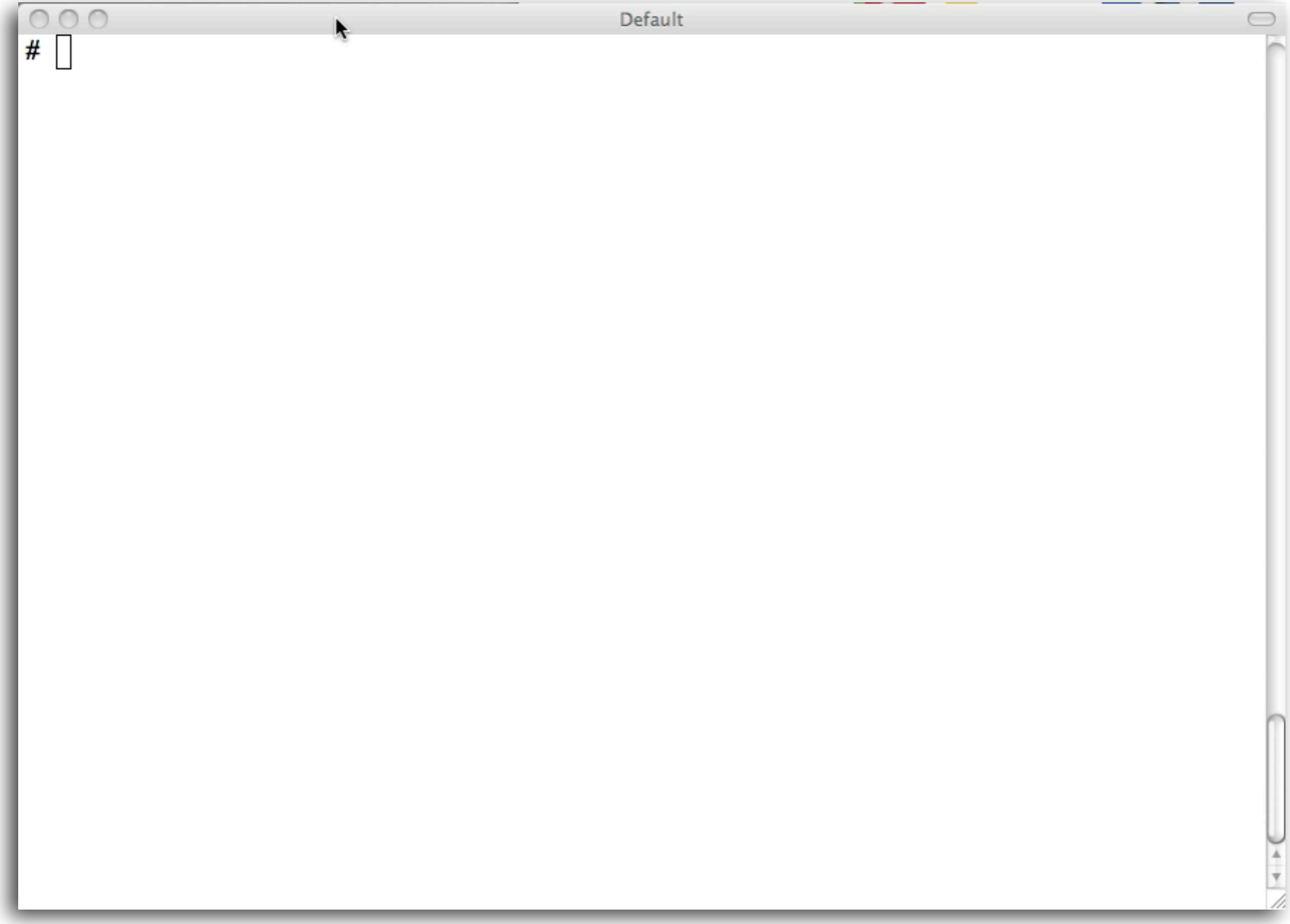
We have a
Focused Goal
Today

The Demos follow a pattern

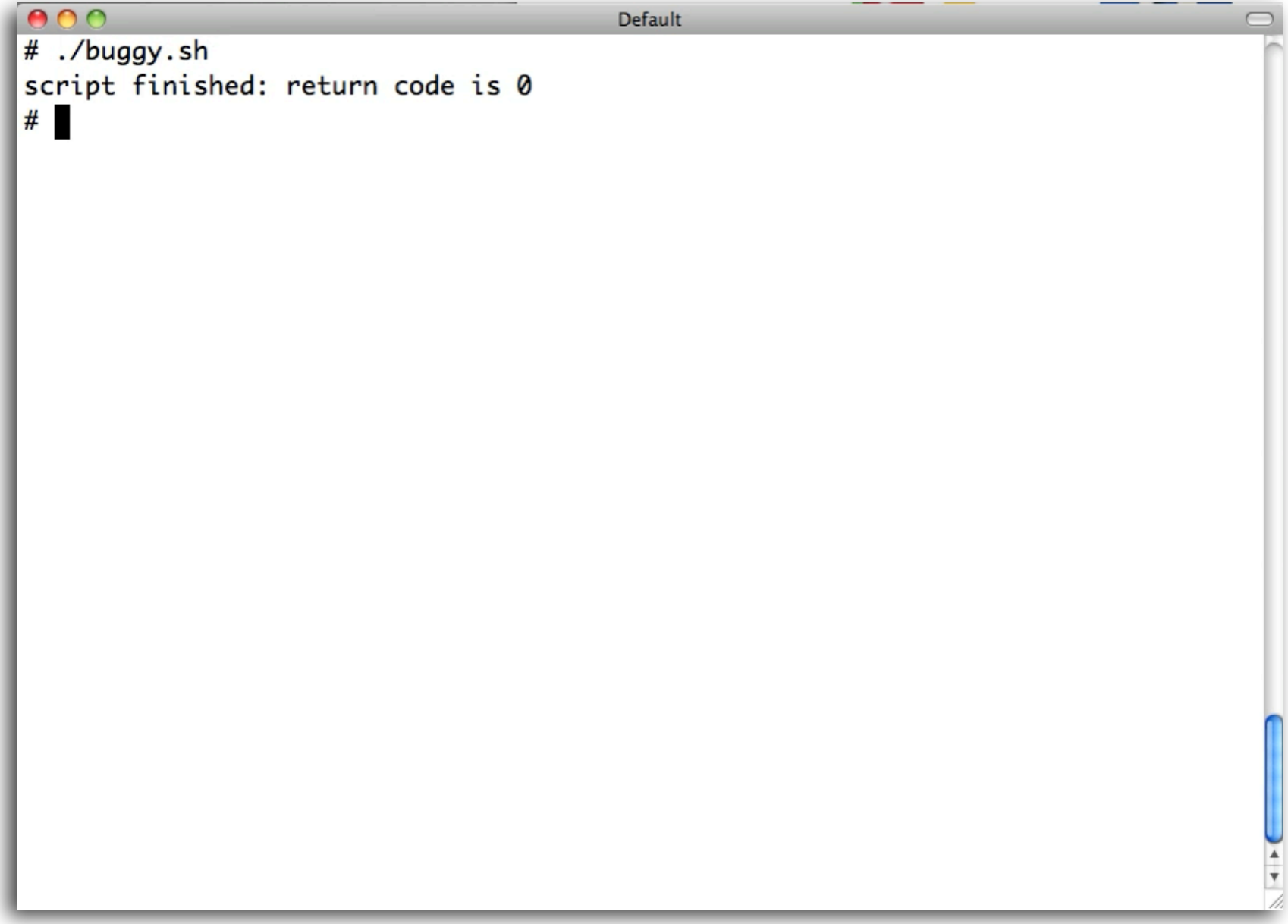
The demo script

```
#!/opt/local/bin/bash
set -o option
echo "My process list:"    >outputfile.txt

ps -ef 2>&1 |grep "^$USR"  >outputfile.txt

echo "script finished: return code is $?"
```

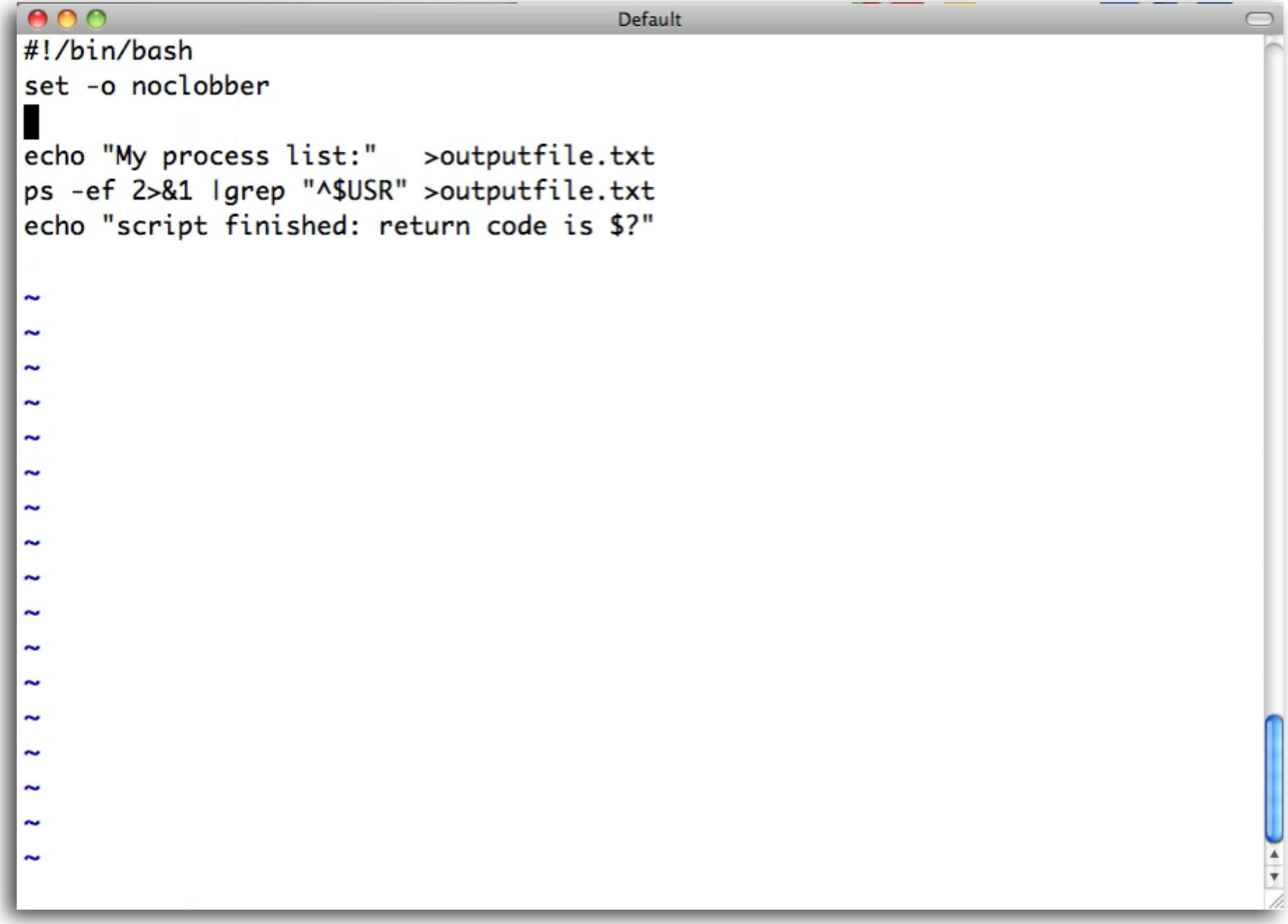
noclobber demo



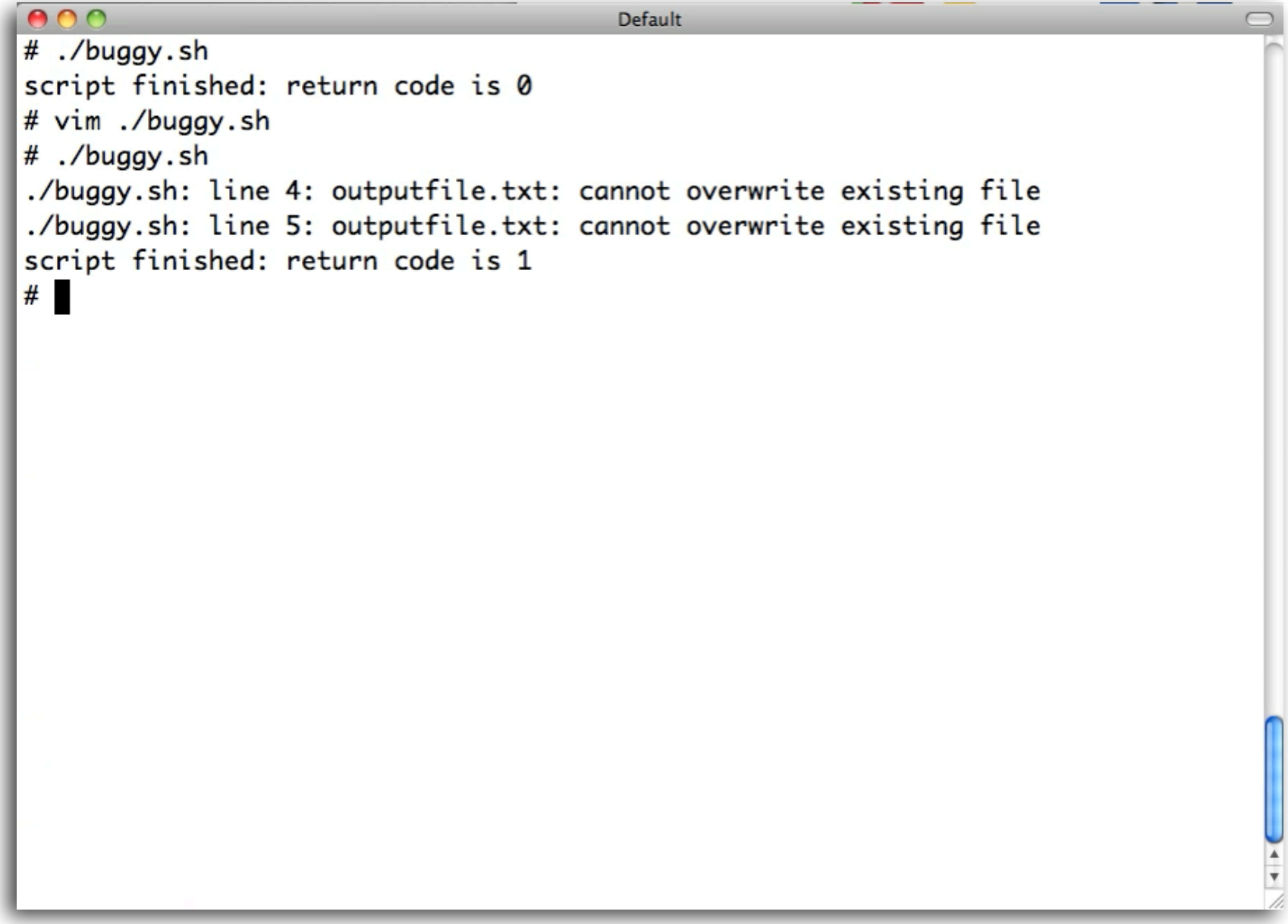
noclobber demo



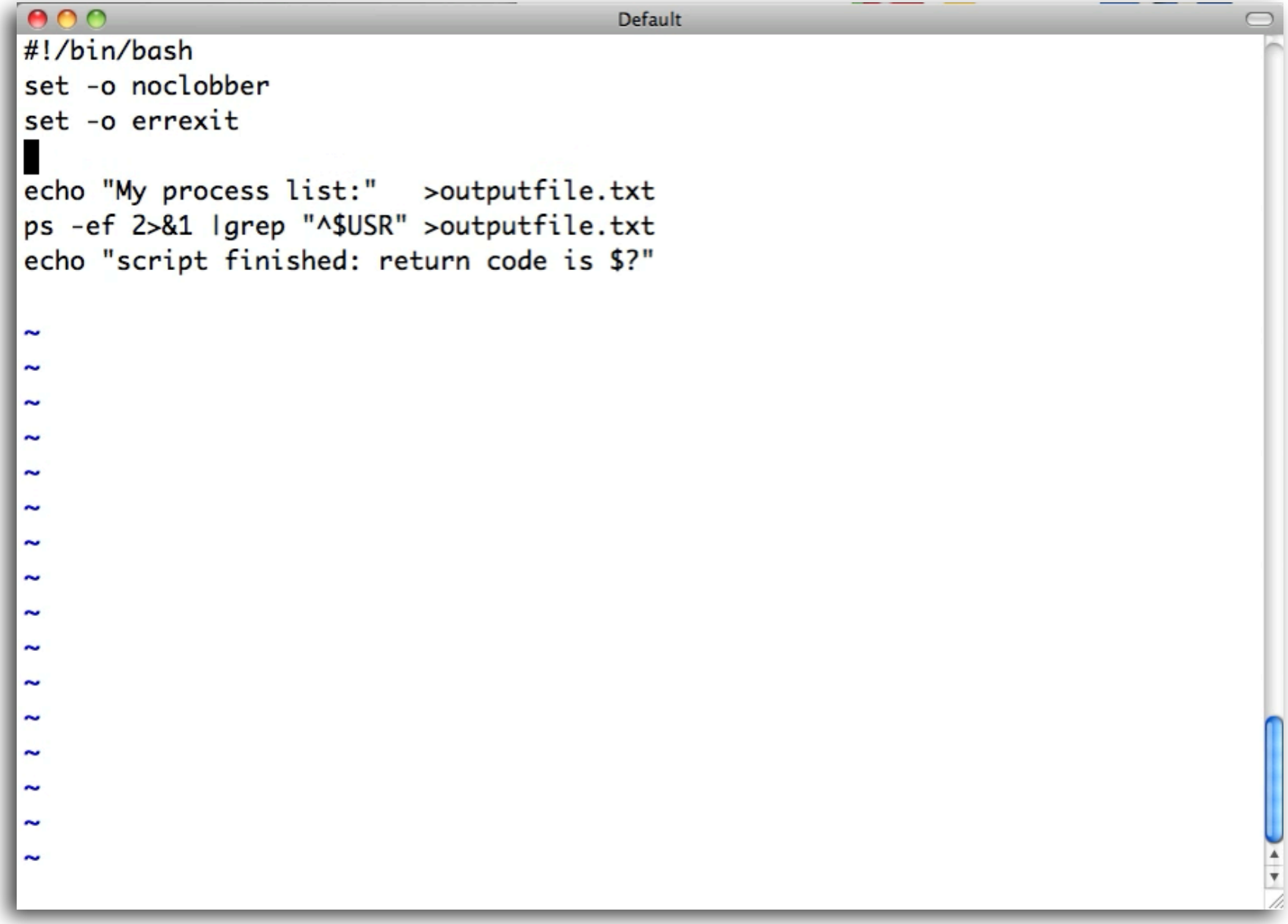
noclobber demo



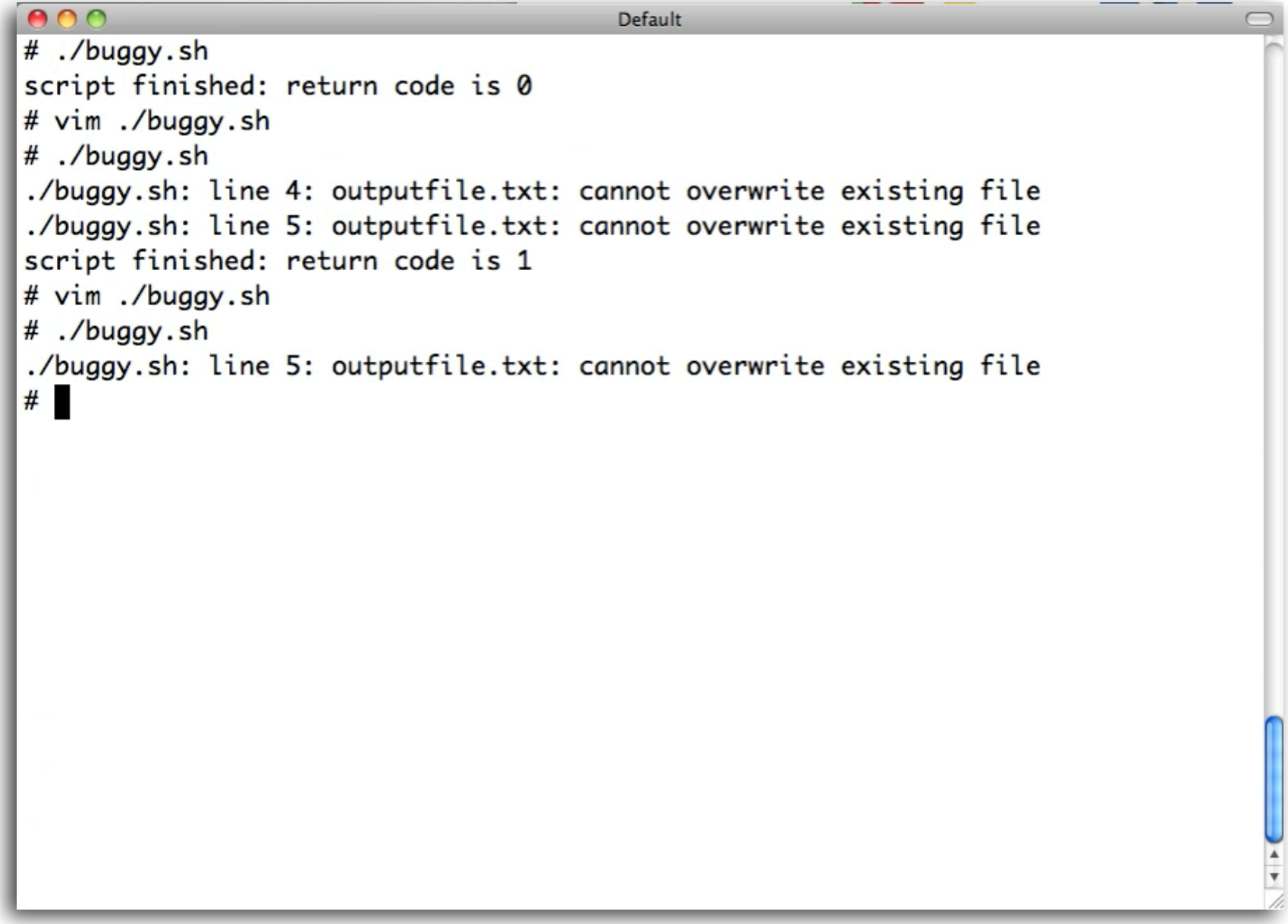
noclobber demo



noclobber demo



noclobber demo



noclobber demo

Default

```
#!/bin/bash
set -o noclobber
set -o errexit

rm -f outputfile.txt
echo "My process list:"   >outputfile.txt
ps -ef 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"
```

What did we learn?

- `set -o noclobber`
  - used to avoid overlaying files

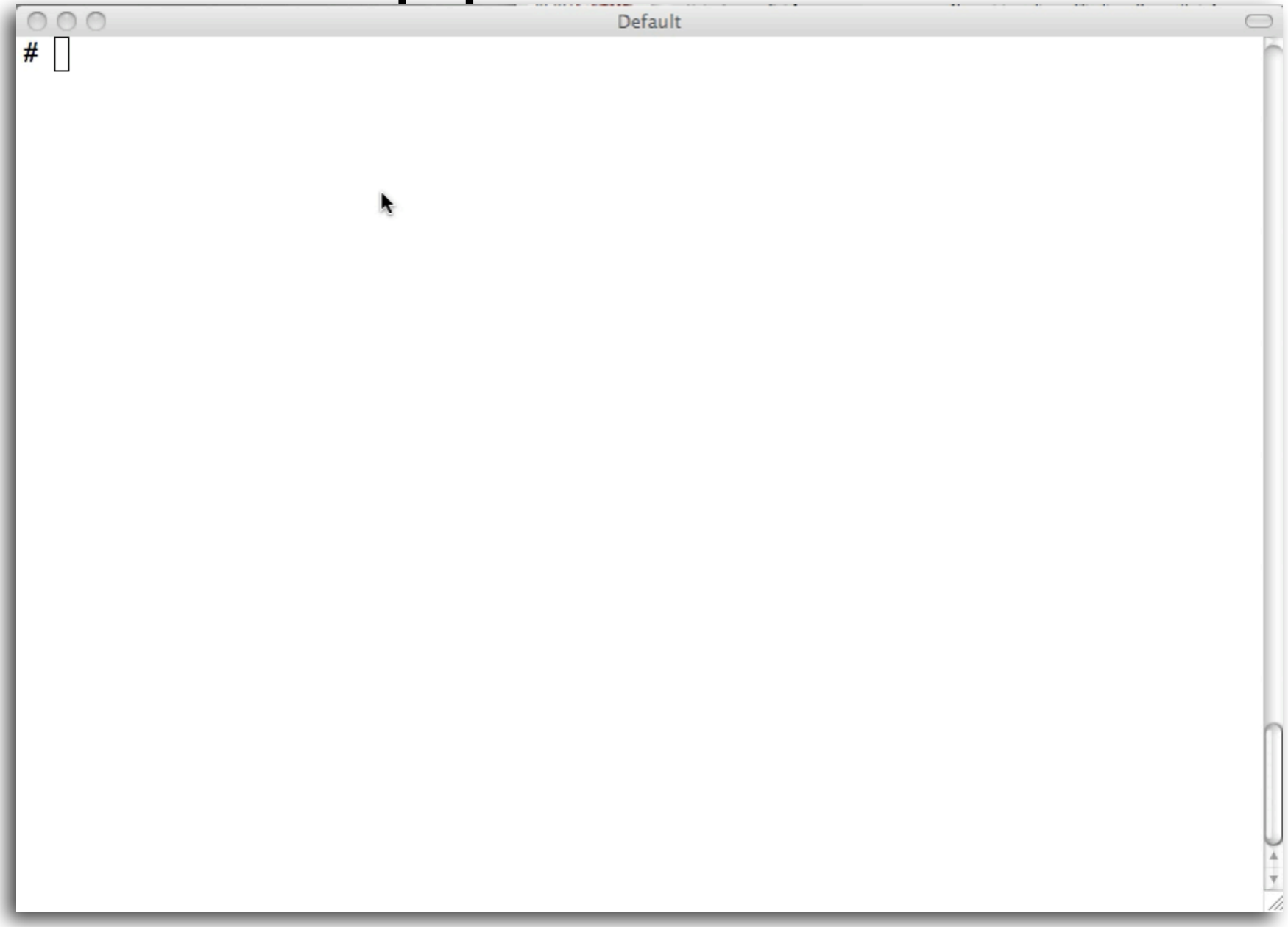
- `set -o errexit`
  - used to exit upon error avoiding cascading errors

- `echo “My process list:” |>outputfile.txt`
  - used to intentional clobber file

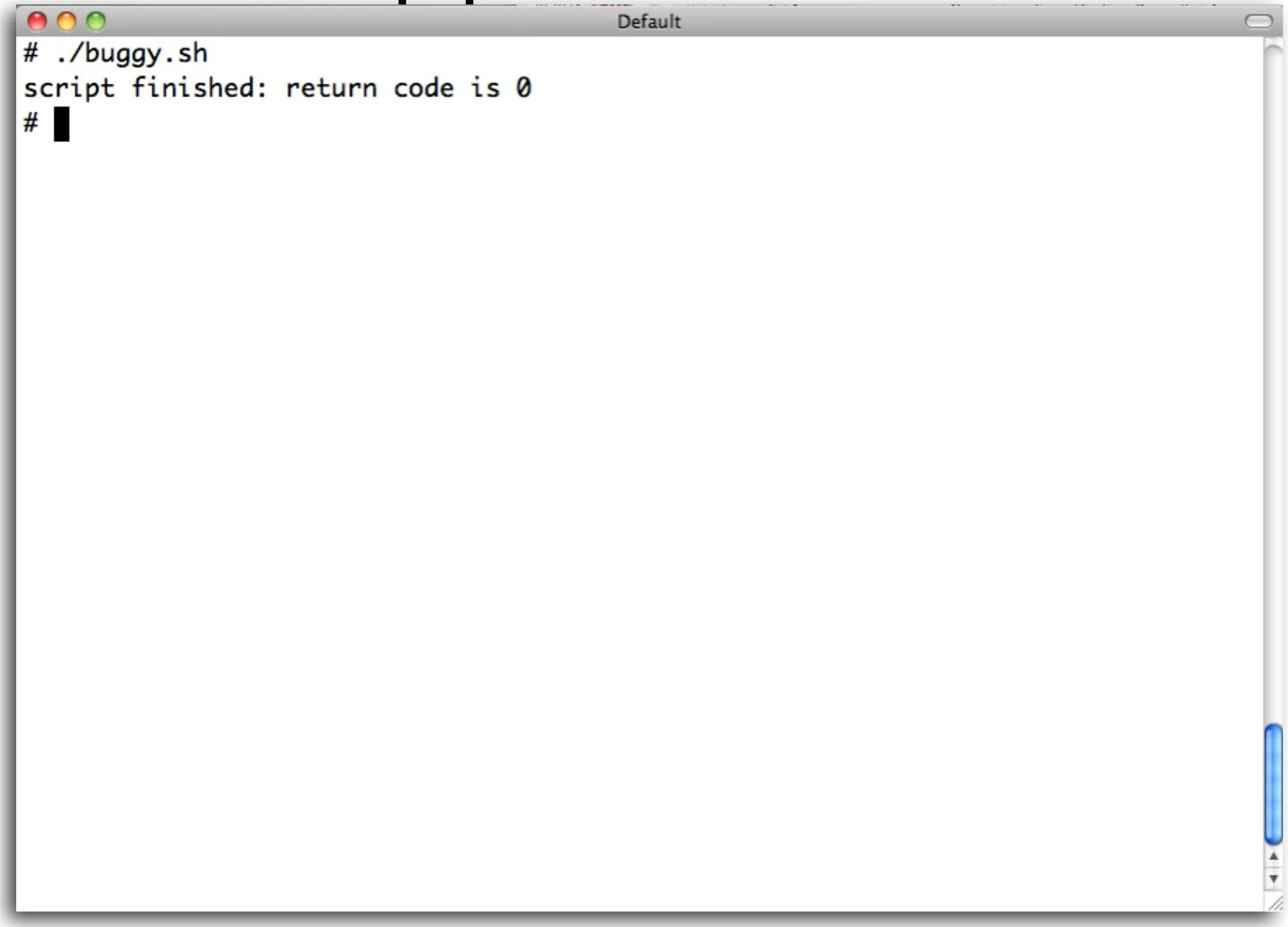
```
rm -f outputfile.txt
echo “My process list:” >outputfile.txt
# should be:
echo “My process list:” |>outputfile.txt
```

command1 | *command2*

pipefail demo



pipefail demo

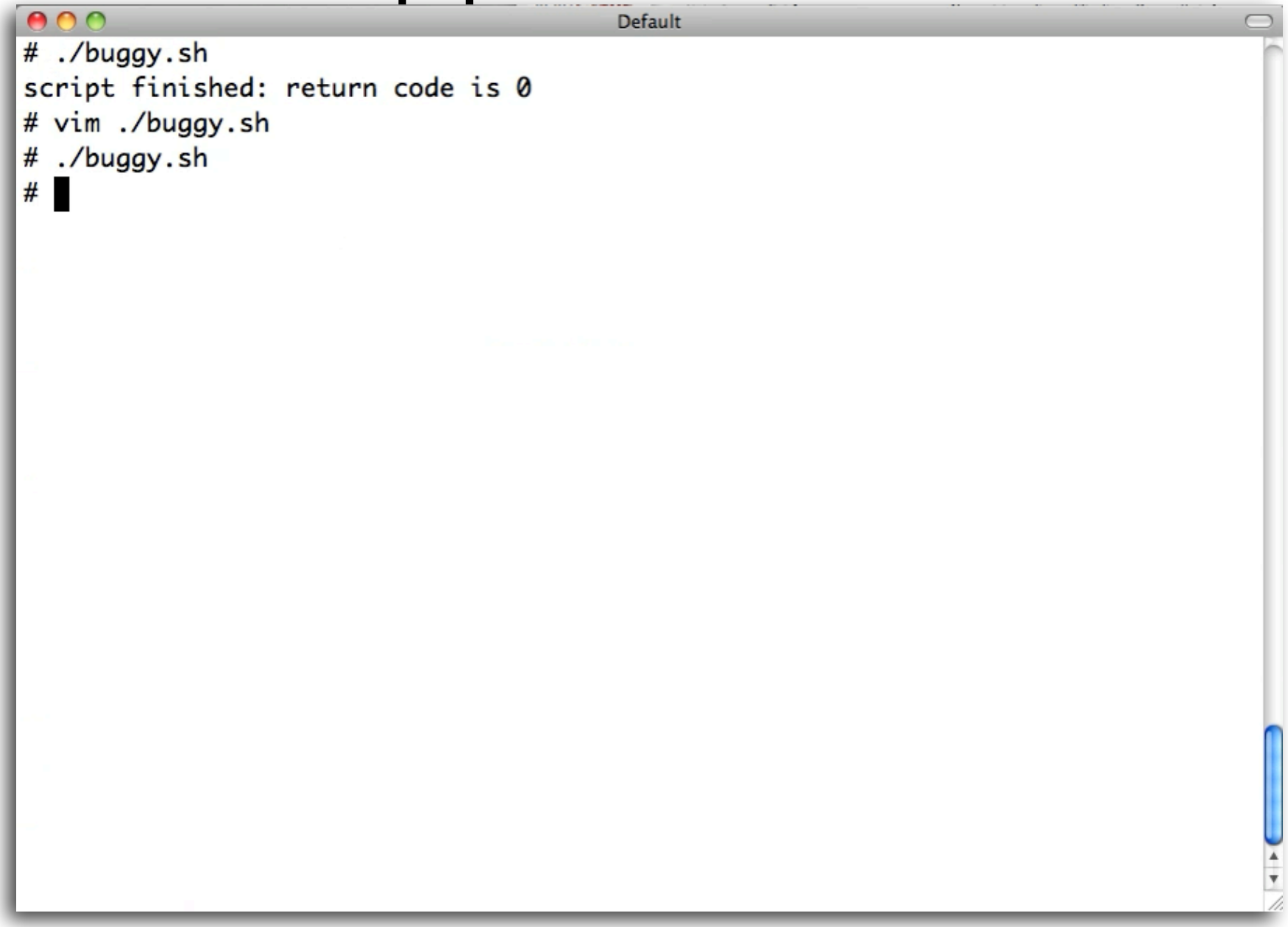


pipefail demo

Default

```
#!/bin/bash
set -o noclobber
set -o errexit
set -o pipefail

rm -f outputfile.txt
echo "My process list:"   >outputfile.txt
ps -ef 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

pipefail demo

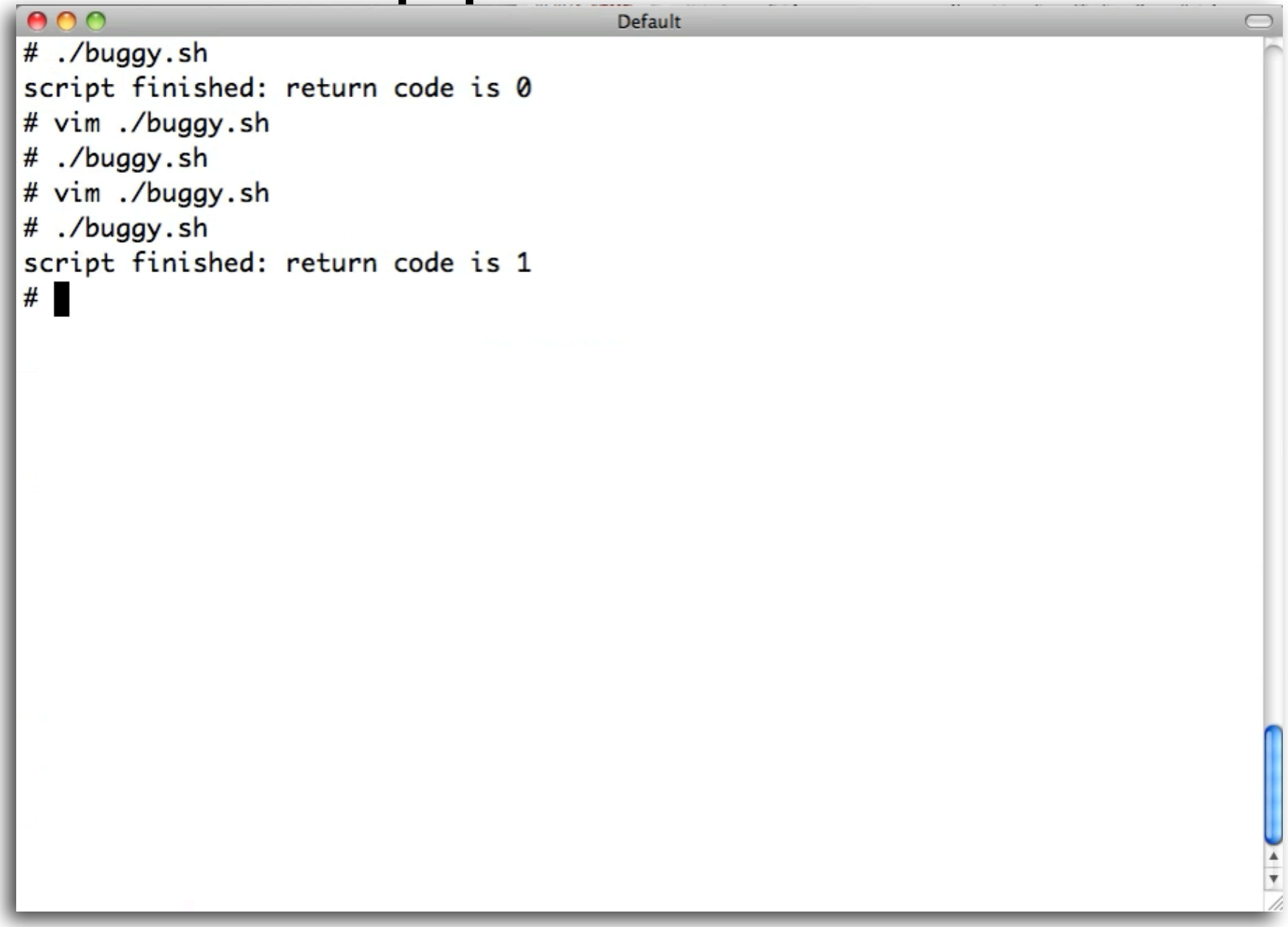


pipefail demo

Default

```
#!/bin/bash
set -o noclobber
#set -o errexit
set -o pipefail

rm -f outputfile.txt
echo "My process list:"   >outputfile.txt
ps -ef 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

pipefail demo

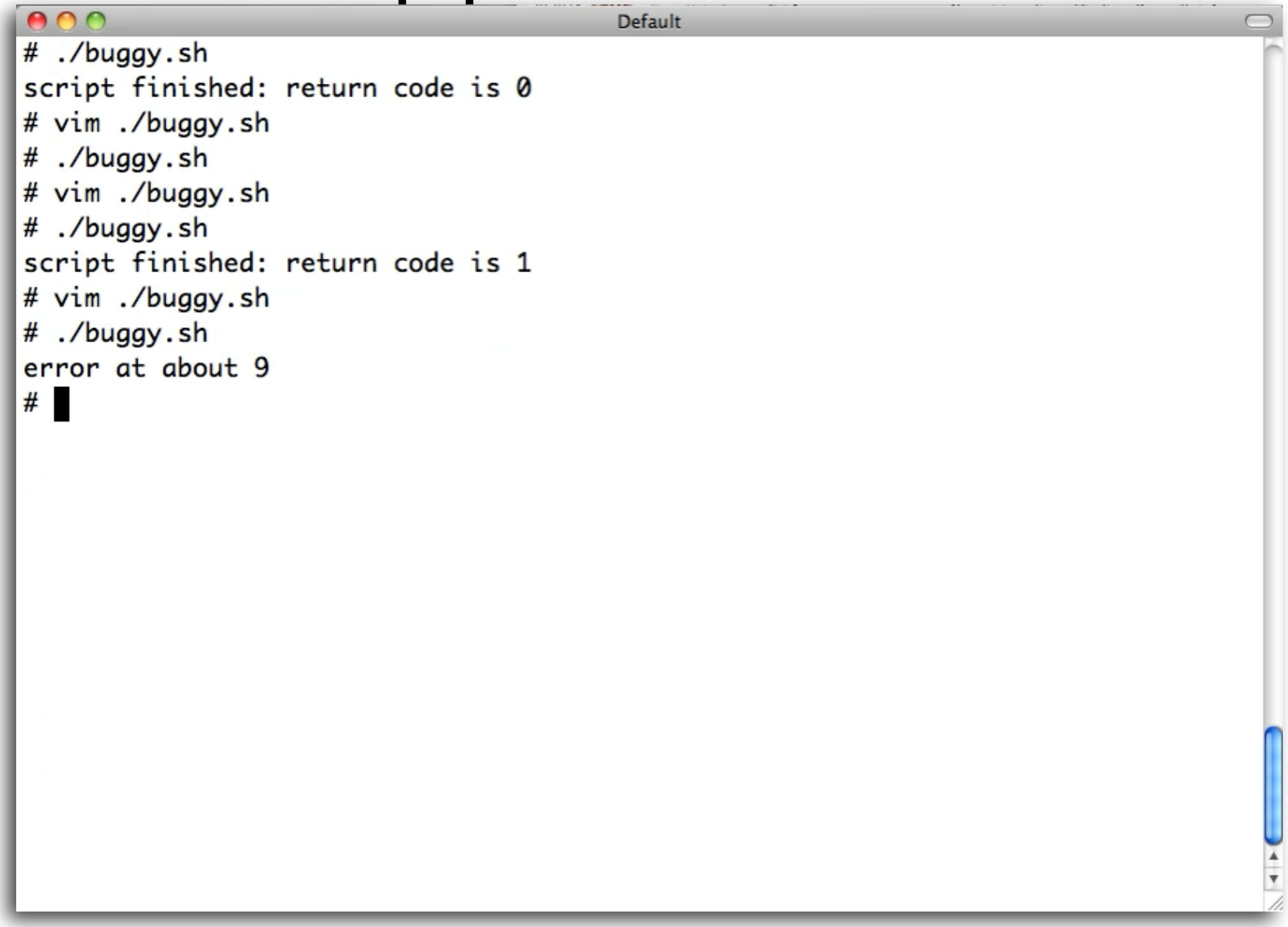


pipefail demo

Default

```
#!/bin/bash
set -o noclobber
set -o errexit
set -o pipefail
trap 'echo error at about $LINENO' ERR

rm -f outputfile.txt
echo "My process list:"   >outputfile.txt
ps -ef 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

pipefail demo



pipefail demo

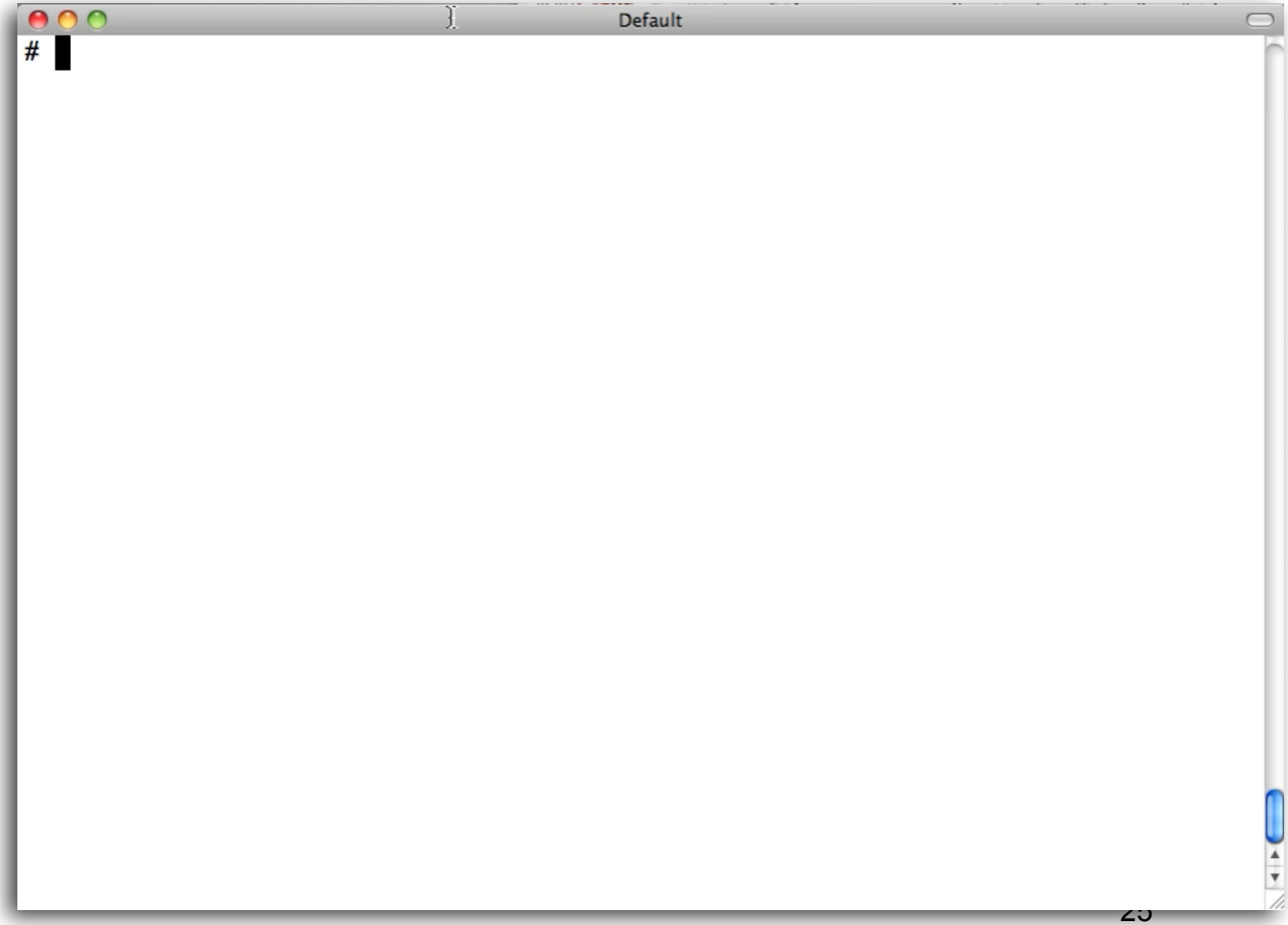
Default

```
#!/bin/bash
set -o noclobber
set -o errexit
set -o pipefail
trap 'echo error at about $LINENO' ERR

rm -f outputfile.txt
echo "My process list:"   >outputfile.txt
ps aux 2>&1 |grep "^$USR" >>outputfile.txt
echo "script finished: return code is $?"

~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

What did we learn?

- `set -o pipefail`
  - unveils hidden failures
- `set -o errexit`
  - can exit silently
- `trap` *`command`* `ERR`
  - corrects silent exits
- `$LINENO`
  - enhances error reporting

nounset demo



What did we learn?

- `set -o nounset`
  - exposes unset variables

the final demo script

```
#!/opt/local/bin/bash

set -o noclobber
set -o errexit
set -o pipefail
set -o nounset
trap 'echo error at about $LINENO' ERR

mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USER"  >>outputfile.txt

echo "script finished: return code is $?"
```

the final demo script

```
#!/opt/local/bin/bash


source ./strict.sh      || exit 1


mv outputfile.txt outputfile.bak
echo "My process list:"    >outputfile.txt
ps aux 2>&1 |grep "^$USER"  >>outputfile.txt

echo "script finished: return code is $?"
```

stringent.sh

https://github.com/pottmi/stringent.sh/

```
set -o errexit
set -o noclobber
set -o nounset
set -o pipefail

function traperr
{
   echo "ERR:${BASH_SOURCE[1]}:${BASH_LINENO[0]}
" >&2
}

set -o errtrace
trap traperr ERR
```

fail.sh

```
#!/bin/bash

source ./stringent.sh || exit 1

echo "before going to fail" >&2

false    # force a failure

echo "after going to fail" >&2
```

```
$ ./fail1.sh
before going to fail
ERROR: ./fail1.sh:7
$
```

fail.sh

```
#!/bin/bash

source ./stringent.sh || exit 1
function goingtofail
{
    echo "before going to fail" >&2
    echo "start going to fail" >&2
    false   # forced failure
    echo "after going to fail" >&2
    echo "end going to fail" >&2
}


goingtofail
```

```
$ ./fail.sh
before going to fail
start going to fail
ERROR: ./fail.sh:5
$
```

trapperr needs improvement

```
function traperr
{
    echo "ERR:${BASH_SOURCE[1]}:${BASH_LINENO[0]}" >&2
    if (( $BASH_SUBSHELL >= 1 ))
    then
        kill $$
    fi
    if (( $nestlevel <= 2 ))
    then
        echo "ERR:${BASH_SOURCE[1]}:${BASH_LINENO[0]}" >&2
    else
        echo "ERR:${FUNCNAME[1]}(${BASH_SOURCE[1]}:${BASH_LINENO[0]})" >&2
        for (( i = 2 ; i < $nestlevel ; i++ ))      do
            echo "    ${FUNCNAME[$i]}(${BASH_SOURCE[$i]}:" \
                "${BASH_LINENO[($i-1)]})" >&2
        done
    fi
}
declare -i nestlevel
nestlevel=${#FUNCNAME[@]}
```

```
before going to fail
start going to fail
ERR:./fail.sh:6
end going to fail
after going to fail
```

```
before going to fail
start going to fail
ERR:./fail.sh:6
Terminated
```

```
before going to fail
start going to fail
ERR:goingtofail(./fail.sh:6)
      main(./fail.sh:16)
Terminated
```

PIPESTATUS

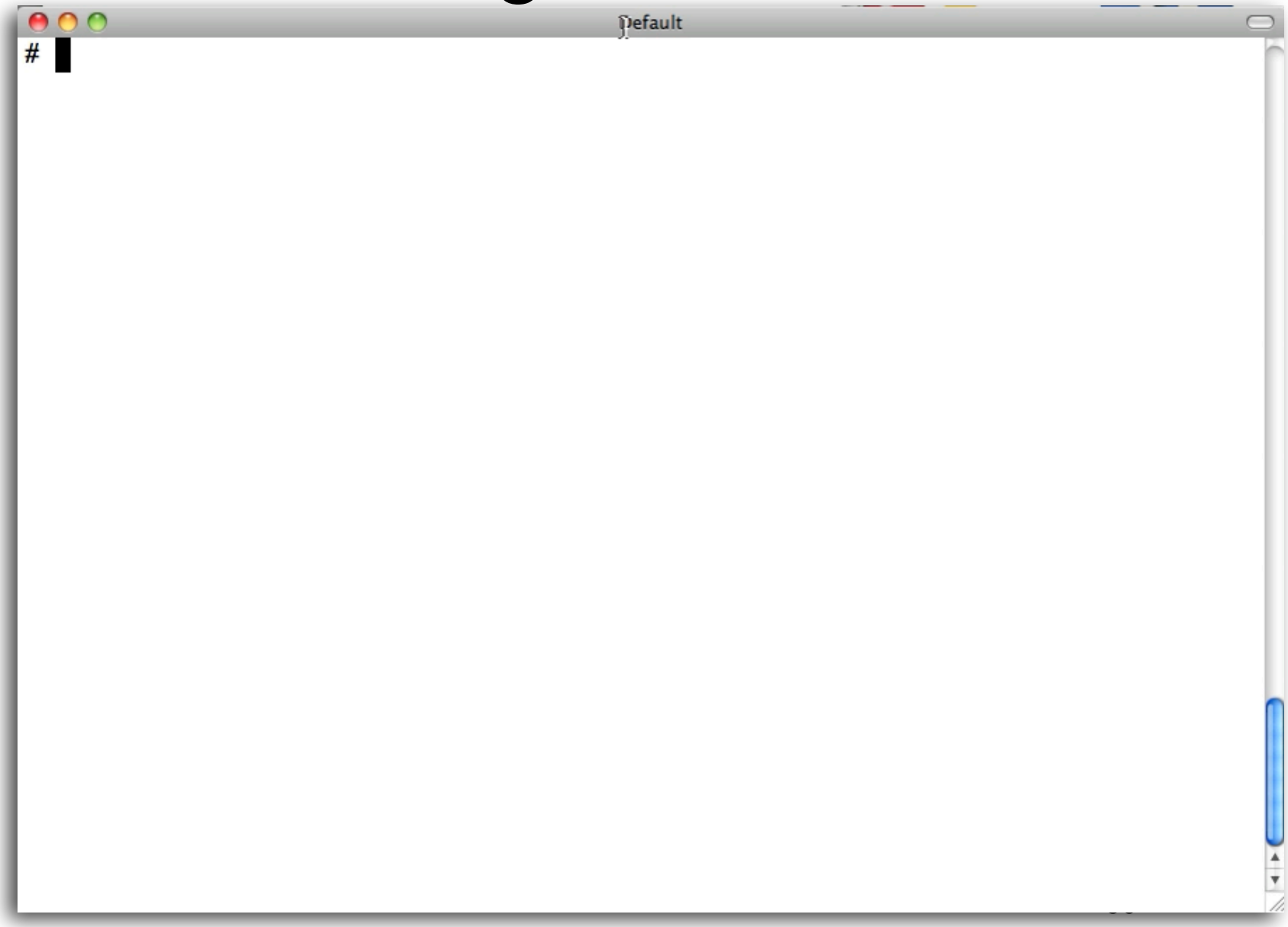
```
bash-3.1$  ps -ef 2>&1 |grep "^$USR" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 1 0  $? = 0
bash-3.1$  set -o pipefail
bash-3.1$  ps -ef 2>&1 |grep "^$USR" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 1 0  $? = 1
bash-3.1$  ps aux 2>&1 |grep "^$USER" >/dev/null
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 0 0  $? = 0
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
 PIPESTATUS = 0  $? = 0
```

PIPESTATUS is transient

```
bash-3.1$  true | false | true
bash-3.1$  echo "PIPESTATUS = ${PIPESTATUS[*]}  \$? = $?"
   PIPESTATUS = 0 1 0  $? = 1
bash-3.1$  true | false | true
bash-3.1$  declare -a SAVEPS=( $? ${PIPESTATUS[@]})
bash-3.1$  echo "SAVEPS = ${SAVEPS[*]}"
   SAVEPS = 1 0 1 0
```

Variables

Integer Demo



What did we learn

- `stringent.sh`
  - Proven to be a good idea
- `declare -i` *`variable`*
  - non-integer values caught sooner
- unset variables used as an int are 0
  - unless caught with `set -o nounset`
- `$(( ... ))`
  - arithmetic syntax

Arithmetic Syntax

- intA=$((($intB + 5) * 2))
  - Allowed anywhere a variable is allowed
- let "intA = ($intB + 5) * 2"
  - returns 0 or 1
- ((intA = ($intB + 5) * 2))
  - equivalent to let
- intA=\($intB+5\)*2
  - no spaces allowed
  - Special characters must be escaped
  - intA must be declare -i
- intA=$[($intB + 5) * 2]
  - deprecated

Two More

- external command
  - intA=$(echo “($intB + 5) * 2” | bc)
  - Supports decimal math
  - Supports large integers

- eval
  - Char=B
  - eval “intA=\$(((\$int$Char + 5) * 2))”

Eval Usage

```
function DD
{
   local NAME=$1
   local VALUE=$2
   eval "DD_$NAME=\$VALUE"
}

DD BOB BUILDER

echo "var DD_BOB = " $DD_BOB


var DD_BOB =  BUILDER



# bash ddtest.bash

var DD_BOB =  BUILDER
```

```
eval “DD_$NAME=\$VALUE"

      DD_$NAME=\$VALUE
      DD_BOB=$VALUE
      DD_BOB=BUILDER
```

Variable format

```
$MYVAR}
```

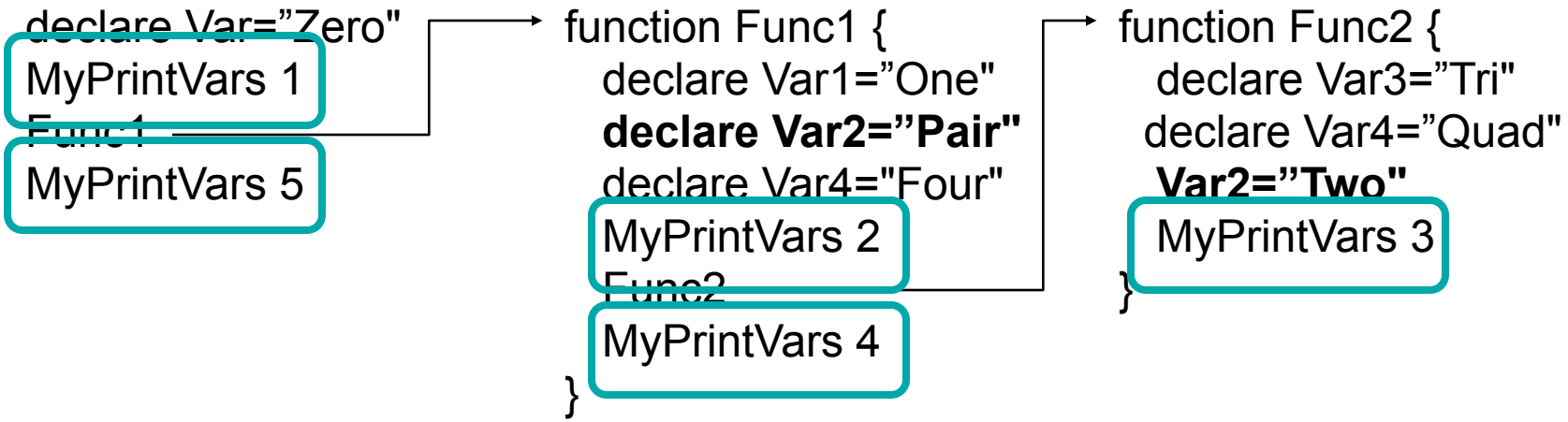
Variable format

```
${MYVAR%%.txt
```

local variables

- weak
- good enough
- not just local, local and below
- two ways to declare:
  - declare
  - local
- $1, $2, ... are not scoped the same

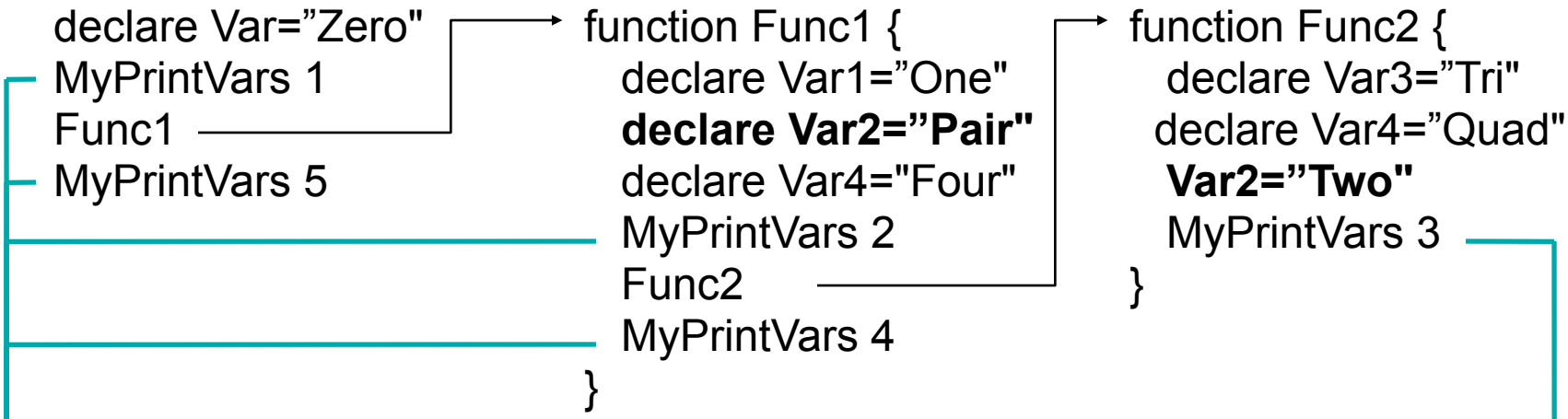
Scoping


declare Var=”Zero"
MyPrintVars 1
Func1
MyPrintVars 5
function Func1 {
declare Var1=”One"
declare Var2=”Pair"
declare Var4="Four"
MyPrintVars 2
Func2
MyPrintVars 4
}
function Func2 {
declare Var3=”Tri"
declare Var4=”Quad"
Var2=”Two"
MyPrintVars 3
}

Handling undefined variables

```
function MyPrintVars {
    echo -n "$1 "
    echo -n "Var1=${Var1:-notset}"
    echo -n "Var2=${Var2:-notset}"
    echo -n "Var3=${Var3:-notset}"
    echo -n "Var4=${Var4:-notset}"
    echo    "Var5=${Var5:-notset}"
}
```

Scoping



1 Var=Zero Var1=notset **Var2=notset** Var3=notset Var4=notset

2 Var=Zero Var1=One **Var2=Pair** Var3=notset Var4=Four

3 Var=Zero Var1=One **Var2=Two** Var3=Tri Var4=Quad

4 Var=Zero Var1=One **Var2=Two** Var3=notset Var4=Four

5 Var=Zero Var1=notset **Var2=notset** Var3=notset Var4=notset

readonly variables

- Two ways to declare
  - declare -r
  - readonly
- One way trip
- Used with -i to create readonly integers
- readonly can be used on system variables
  - e.g. keep users from changing their prompt
  - not documented!

Syntax check in make

- bash -n myscript.bash
- bin/myscript : src/myscript.bash
  - bash -n $< && cp $@ $<

conditionals

if *command*

if (( ))

if let

if [ ]

if test

if [[ ]]

if *command*

```
set +o errexit
grep Jim /etc/passwd
declare -i Status=$?
set -o errexit
if (( $Status == 0 ))
then
   echo “Jim is a user”
fi
```

```
if grep Jim /etc/passwd
then
   echo “Jim is a user”
fi
```

bang has side effects!

```
! grep Jim /etc/passwd
declare -i Status=$?
if (( $Status != 0 ))
then
   echo “Jim is a user”
fi
```

What did we learn?

- set +o errexit turns off errexit
  - errexitoff for stringent.sh
- Save $? to a permanent variable
- ! turns off errexit for a single command
- zero is true, non-zero is false
- if (( )) used for numeric tests

gotcha

- **`if [[ $Age > 20 ]]`**
  - bad, 3 year old buys beer!
  - "`>`" is a string comparison operator
- **`if [ $Age > 20 ]`**
  - bad, everyone buys beer!
  - "`>`" is a redirection operator
- **`if [[ $Age -gt 20 ]]`**
  - Good, but fails if `$Age` is not numeric
  - "`[[]]`" are for comparing text.
- **`if (( $Age > 20 ))`**
  - best, "`(())`" is for comparing numbers.
  - "`$`" on `Age` is optional

test and [

```
bash-3.1$ which test
/bin/test
bash-3.1$ which [
/bin/[
bash-3.1$ ls -i /bin/[ /bin/test
6196593 /bin/[
6196593 /bin/test
```

So?

if [[ ]]

[versus [[

- `[[ $a == z* ]]`
  - True if $a starts with an "z”.
- `[[ $a == "z*" ]]`
  - True if $a is exactly equal to “z*”.
- `[ $a == z* ]`
  - Error if $a has a space.
  - Error if more than one filename starts with z.
  - True if a filename exists that starts with z and is exactly $a.
  - True if no filenames exist that start with z and $a equals z*.
- `[ "$a" == "z*" ]`
  - True if $a is exactly equal to z*.

the rules

- use [
  - when you “want” to use file globbing
  - Should be very rare
- use ((
  - when you want to do math/numeric
- use [[
  - for everything else

regular expressions

- Introduced with version 3.0
- Implemented as part of [[ ]]
- Uses binary operator =~
- Supports extended regular expressions
- Supports parenthesized subexpressions

regular expression

```
declare MyStr="the quick brown fox"

[[ $MyStr == "the*" ]]    # false: must be exact
[[ $MyStr == the* ]]      # true: pattern match

[[ $MyStr =~ "^the" ]]    # true
[[ $MyStr =~ “brown” ]] # true
[[ $MyStr =~ "the  *quick  *brown" ]]    # true
```

subexpressions

```
declare MyStr="the quick brown fox"

if [[ $MyStr =~ "the ([a-z]*) ([a-z]*) ([a-z]*)" ]]
then
   echo "${BASH_REMATCH[0]}" # the quick brown
   echo "${BASH_REMATCH[1]}" # quick
   echo "${BASH_REMATCH[2]}" # brown
fi
```

bad expressions

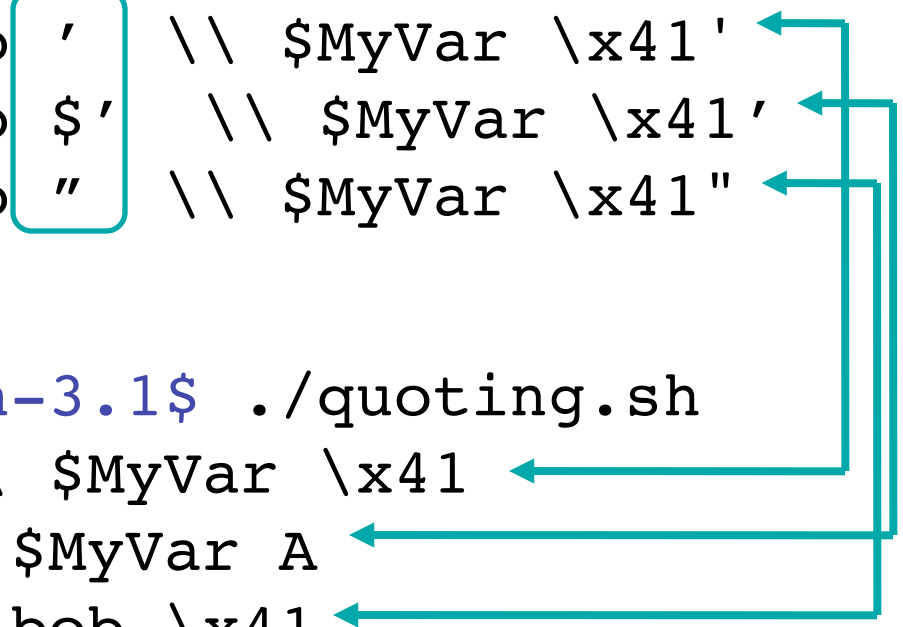
```
declare MyStr="the quick brown fox"

if [[ $MyStr =~ "the [a-z) ([a-z*)" ]]
then
   echo "got a match"
elif (( $? == 2 ))
then
   : # no match, colon is no-op command
else
   traperr "Assertion Error: Regular expression error"
   exit 1
fi
```

gotcha

- **`cp $srcfile $dstfile`**
  - broken if $srcfile has a space
- **`cp "$srcfile" "$dstfile"`**
  - broken if srcfile begins with -
- **`cp -- "$srcfile" "$dstfile"`**
  - `Will clobber dstfile`
- **`cp -n -- "$srcfile" "$dstfile"`**
  - `-n prevents clobbering destination`

quoting

```
declare MyVar="bob"
echo  '   \\ $MyVar \x41'
echo  $'   \\ $MyVar \x41'
echo  "   \\ $MyVar \x41"


bash-3.1$ ./quoting.sh
  \\ $MyVar \x41
  \ $MyVar A
  \ bob \x41
```



quoting recommendation

- quote variables liberally
  - extra quotes likely to cause a consistent error
  - missing quotes are likely to cause inconsistent behavior
- Safe Exceptions
  - within if [[ ]]
  - Integer variables (define -i)
  - within if (( ))

Handling undefined variables

```
function PrintVars {
   echo -n “Var1=${Var1:-notset}”
   echo -n “Var2=${Var2:-notset}”
   echo -n “Var3=${Var3:-notset}”
   echo -n “Var4=${Var4:-notset}”
   echo -n “Var5=${Var5:-notset}"
}
```

unset variables

- `${parameter -word}`
  - returns word
- `${parameter +word}`
  - returns empty (returns word if set)
- `${parameter =word}`
  - sets parameter to word, returns word
- `${parameter ?message}`
  - echos message and exits

unset variables

- `${parameter-word}`
- `${parameter+word}`
- `${parameter=word}`
- `${parameter?message}`

default variables

```
function MyDate
{
   declare -i Year=${1:?"$0 Year is required"}
   declare -i Month=${2:-1}
   declare -i Day=${3:-1}

   if (( $Month > 12 )); then
      echo "Error Month > 12" >&2
      exit 1
   fi
   if (( $Day > 31 )); then
      echo "Error Day > 31" >&2
      exit 1
   fi

   echo "$Year-$Month-$Day"
}
```

sub strings

```
declare MyStr="The quick brown fox"

echo "${MyStr:0:3}"   # The
echo "${MyStr:4:5}"   # quick
echo "${MyStr: -9:5}" # brown
echo "${MyStr: -3:3}" # fox
echo "${MyStr: -9}"   # brown fox
```

substr by pattern

- `${Var#pattern}`
- `${Var%pattern}`
- `${Var##pattern}`
- `${Var%%pattern}`

a jingle

We are #1 because we give 110%.

Also, note the position on the keyboard.

substr by pattern

```
declare MyStr="/home/pottmi/my.sample.sh"

echo "${MyStr##*/}" # my.sample.sh
echo "${MyStr%.*}"  # /home/pottmi/my.sample
echo "${MyStr%/*}"  # /home/pottmi

echo "${MyStr#*/}” #home/pottmi/my.sample.sh
echo "${MyStr%%.*}" # /home/pottmi/my
```

search and replace

- `${Var/pattern/replace}`

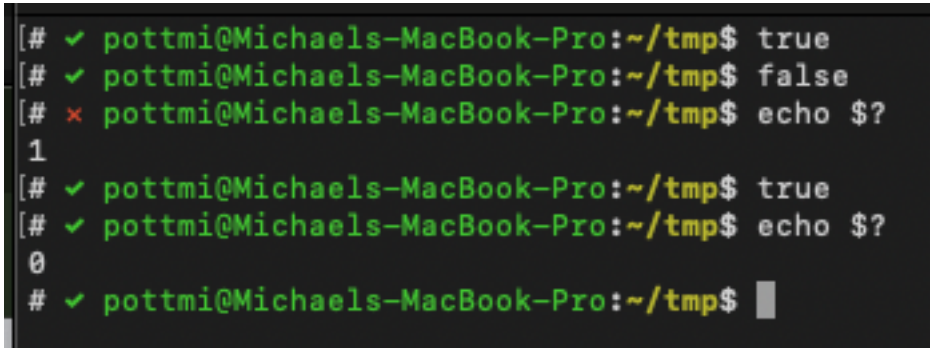
substr by pattern

```
declare MyStr="the fox jumped the dog"

echo "${MyStr/the/a}"
                                  # a fox jumped the dog
echo "${MyStr//the/a}”
                                  # a fox jumped a dog
echo "${MyStr//the }”
                                  # fox jumped dog
```

xargs Ninja

```
grep -r Tapp

                                    grep Tapp *

find . -type f                  |xargs     grep Tapp

find . -type f -print0          |xargs -0 grep Tapp

find . -type f -print0          |xargs -0 grep Tapp /dev/null

cat listOfFiles.txt |tr '\n' '\0' |xargs -0 grep Tapp /dev/null
```

Command Line




```
PS1='# $(if [ $? -eq 0 ]; then printf
"\033[01;32m""\xE2\x9C\x93"; else printf
"\033[01;31m""\xE2\x9C\x95"; fi) \[\e[00;32m\]
\u@\h\[\e[01;37m\]:\[\e[01;33m\]\w\[\e[01;37m\]$\
[\e[00;37m\] '
```

- # keep errant paste from running commands
- Indicate previous command status

unintended subshells

```
declare -i Count=0
declare Lines

cat /etc/passwd | while read Lines
do
   echo -n "."
  ((Count++))
done

echo " final count=$Count"
```

..... final count=0

unintended subshells

```
declare -i Count=0
declare Lines

                    while read Lines
do
   echo -n "."
  ((Count++))
done </etc/passwd

echo " final count=$Count"
```

..... final count=38

unintended subshells

```
declare -i Count=0
declare Lines

                    while read Lines
do
   echo -n "."
  ((Count++))
done < <(cat /etc/passwd)

echo " final count=$Count"
```

..... final count=38

unintended subshells

```
declare -i Count=0
declare Lines

                    while read Lines
do
   echo -n "."
  ((Count++))
done < <(grep “false$” /etc/passwd)

echo " final count=$Count"
```

..... final count=20

Running vi in a loop

```
while read FileName 0<&3
do
   if ! grep stringent $FileName
   then
      vi $FileName
   fi
done 3< <(ls *.sh)
```

Learn more

- man bash
- O'Reilly - 'Learning the Bash shell'
- http://bashdb.sourceforge.net/bashref.html
- http://www.faqs.org/docs/abs/HTML/
- Ask me to help!

Copyright Notice

stringent.sh available for download from
https://github.com/pottmi/stringent.sh