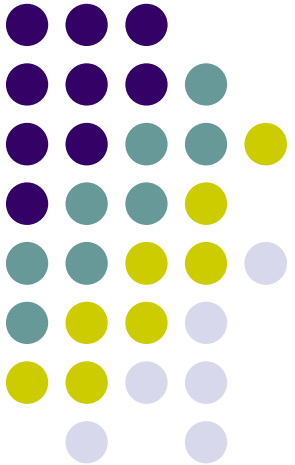
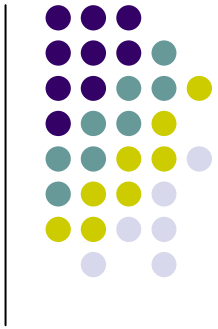
Introduction to Shell Programming

EE1356 Introduction to Information Systems



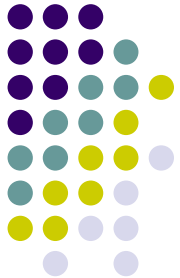
A Bourne Shell Programming Tutorial for learning about using the unix shell.
http://steve-parker.org/sh/sh.shtml

Outline



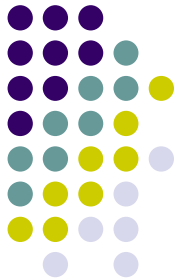
- Introduction
- Philosophy
- A First Script
- Variables - Part I
- Loops
- Test
- Case
- Predefined Shell Variables
- External Programs
- Functions

Introduction



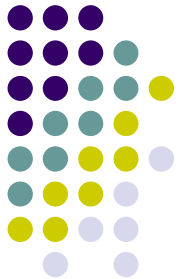
- Introduce some simple but powerful programming available under the bourne shell.
  - Why csh programming is considered harmful.
- This tutorial assumes some prior experience; namely:
  - Use of an interactive Unix shell
  - Minimal programming knowledge - use of variables, functions, etc.
  - Understanding of how Unix commands are structured, and competence in using some of the more common ones.

Philosophy



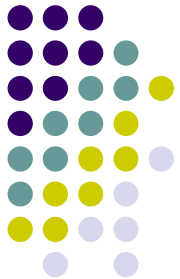
- Shell script programming has a bit of a bad press, because
  - Run slower as compared to a C program, or even an interpreted PERL program.
  - Since it is easy to write a simple shell script, there are a lot of poor quality shell scripts around.
- Good shell scripts.
  - a clear, readable layout.
  - avoiding unnecessary commands.

What Not To DO



- The Award For The Most Gratuitous Use Of The Word Cat In A Serious Shell Script
  - cat /tmp/myfile | grep "mystring"
  - grep "mystring" /tmp/myfile (faster version)
- comp.os.unix.shell newsgroup is a good read
- Unix Power Tools
  - Chapter 44--47

A First Script

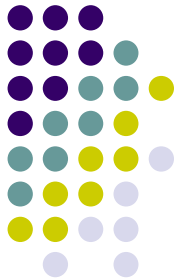


- first.sh

```
#!/bin/sh
# This is a comment!
echo Hello World # This is a comment, too!
```

- The first line tells Unix that the file is to be executed by /bin/sh.
  - #!/usr/bin/perl
- The second line begins with a #. This marks the line as a comment
  - it is ignored completely by the shell.
- The third line runs a command: echo, with two parameters
  - the first is "Hello"; the second is "World".
  - # symbol still marks a comment;

To Run the Script

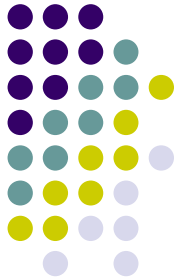


```
$ chmod 755 first.sh
$ ./first.sh
Hello World
$
```

- You will probably have expected that! It's just like running the following command

```
$ echo Hello World
Hello World
$
```

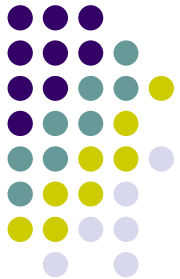
A Modified Version of first.sh



- What about inserting spaces between `Hello` and `World`?
- What about the following script?

```
#!/bin/sh
# This is a comment!
echo "Hello      World"          #This
  is a comment, too!
```

Variables - Part I



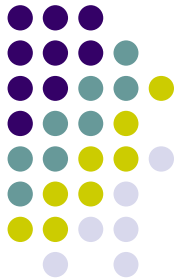
- *Variables*
  - a symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents.
- var.sh

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

Quotes of Variable Assignments

- the quotes of `"Hello World"`
  - `echo Hello World` works because echo will take any number of parameters
  - A variable can only hold one value, so a string with spaces must be quoted to that the shell knows to treat it all as one.
    - Otherwise, the shell will try to execute the command `World` after assigning `MY_MESSAGE=Hello`

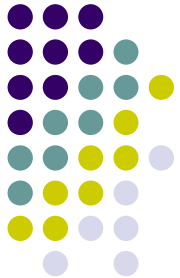
Variables are Strings



- The shell does not care about types of variables
  - Everything are stored as strings.
  - Routines which expect a number can treat them as such.
    - Example:

```
$ x="hello"
$ y=`expr $x + 1`
expr: non-numeric argument
$ x="1"
$ y= `expr $x + 1`
$ echo $y
2
```

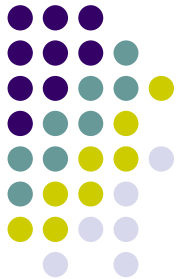
Read a Variable From Users



- Interactively set variable names using the read command
- Example: var2.sh

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're
  well."
```

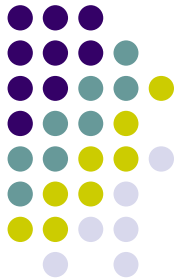
Undefined Variables



- Note that to read an undeclared variable, the result is the empty string.
  - You get no warnings or errors. This can cause some subtle bugs
- Example

```
$MY_OBFUSCATED_VARIABLE=Hello
$echo $MY_OSFUCATED_VARIABLE
#you will get nothing (as the second
  OBFUSCATED is mis-spelled).
```

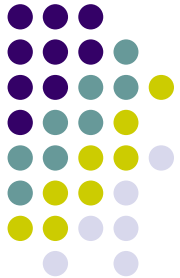
Scope of A Variable



- A variable only lives as long as the current shell is available.
  - Unless declared otherwise by "`export`"
- Example `myvar2.sh`

```
#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

Variables in the Shell Script



- A Variable is not Passed to the Spawned Shell by Default

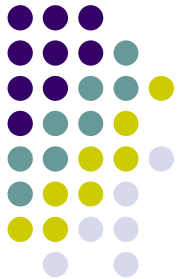
```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

- MYVAR hasn't been set to any value, so it's blank. Then we give it a value, and it has the expected result.

```
$ MYVAR=hello
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

- When you call myvar2.sh from your interactive shell, a new shell is spawned to run the script.

Variables in the Shell Script (Cont.)



- When you call myvar2.sh from your interactive shell, a new shell is spawned to run the script.
  - MYVAR is not passed into the shell script
  - Use `export` for passing variables into sub-shells.

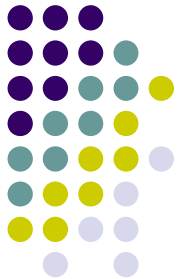
```
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

- Try reading the value of MYVAR:

```
$ echo $MYVAR
hello
$
```

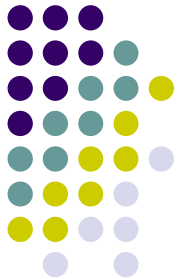
- `MYVAR` set in the shell is not passed back to the calling shell

Setting Variables for the Current Shell



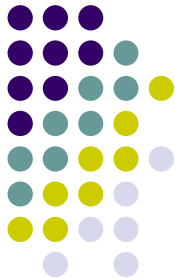
- In order to receive environment changes back from the script, we must *source* the script
  - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.
  - We can source a script via the "." command

Source Examples



```
$ MYVAR=hello
$ export MYVAR
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

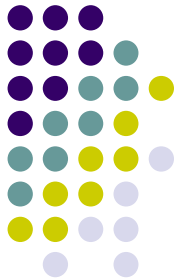
Create New Variables with Defined Variables



- Example script:

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called
  $USER_NAME_file"
touch $USER_NAME_file
#not defined variable $USER_NAME_file
```

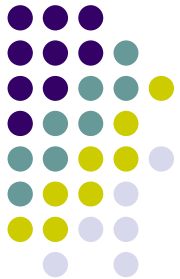
Create New Variables with Defined Variables



- A variable can be identified by ${ }.
- Example: `user.sh`

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called
  ${USER_NAME}_file"
touch "${USER_NAME}_file"
#
```

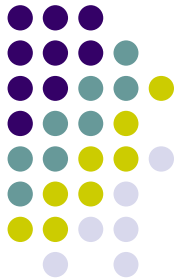
For Loops Examples



- for.sh

```
#!/bin/sh
for i in 1 2 3 4 5
do
   echo "Looping ... number $i"
done
```

For Loops Examples

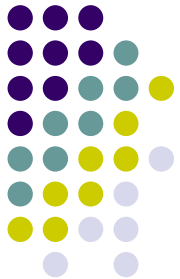


- for2.sh

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done
```

- Note that the * will be interpreted and expanded by sh into file names in the current directory.

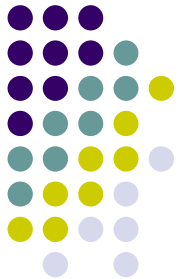
While Loops Examples



- while.sh

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
   echo "Please type something in (bye to
  quit)"
   read INPUT_STRING
   echo "You typed: $INPUT_STRING"
done
```

While Loops Examples

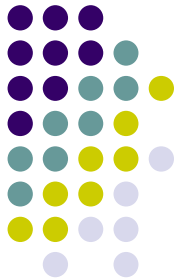


- while2.sh

```
#!/bin/sh
while :
do
   echo "Please type something in (^C to
  quit)"
   read INPUT_STRING
   echo "You typed: $INPUT_STRING"
done
```

- The : will always be evaluated to TRUE.

While Loops Examples

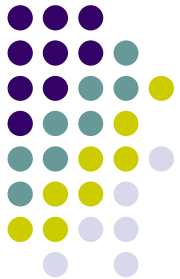


- while3a.sh

```
#!/bin/sh
while read f
do
  case $f in
  hello)          echo English      ;;
  howdy)          echo American     ;;
  gday)           echo Australian   ;;
  bonjour) echo French ;;
  "guten tag")    echo German ;;
  *)        echo Unknown Language: $f          ;;
   esac
done < myfile
```

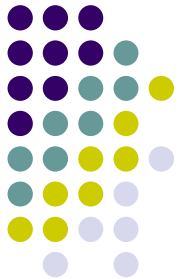
- while will execute command in (do done) loop if “read f” return 0
- while loop is considered as one command, so it can take redirected input from myfile

Test



- Test is a simple but powerful comparison utility.
  - For full details, run man test.
- Test is often combined with `if` and `while` statements.
- Test is often a shell built-in command named as [

Man Test



NAME

test - check file types and compare values

SYNOPSIS

test EXPRESSION
[EXPRESSION]
test OPTION

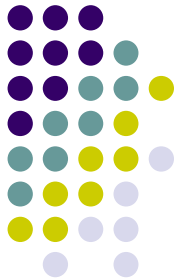
DESCRIPTION

Exit with the status determined by EXPRESSION.

EXPRESSION is true or false and sets exit status. It is one of:

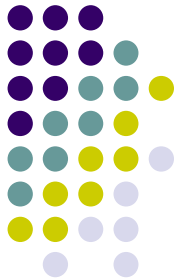
(EXPRESSION)
EXPRESSION is true

Man Test (Expression Logic)



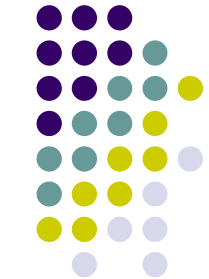
- ! EXPRESSION
  - EXPRESSION is false
- EXPRESSION1 -a EXPRESSION2
  - both EXPRESSION1 and EXPRESSION2 are true
- EXPRESSION1 -o EXPRESSION2
  - either EXPRESSION1 or EXPRESSION2 is true

Man Test (String Comparison)



- [-n] STRING
  - the length of STRING is nonzero
- -z STRING
  - the length of STRING is zero
- STRING1 = STRING2
  - the strings are equal
- STRING1 != STRING2
  - the strings are not equal

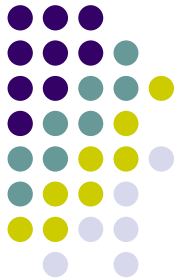
Man Test (Integer Comparison)



- Integer1 <operator> Integer2

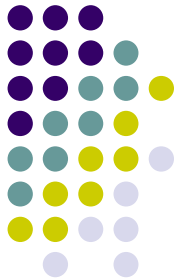
<operator>	meaning
-eq	equal to
-ge	greater than or equal to
-gt	greater than
-le	less than or equal to
-lt	less than
-ne	not equal

Man Test (File Comparison)



- FILE1 -ef FILE2
  - FILE1 and FILE2 have the same device and inode numbers
- FILE1 -nt FILE2
  - FILE1 is newer (modification date) than FILE2
- FILE1 -ot FILE2
  - FILE1 is older than FILE2

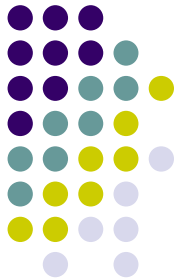
Man Test (File Test)



- <operator> FILE
  - Test the existence and properties of FILE
  - Only frequently used operators are listed

<operator>	meaning
-d	is a directory
-e	FILE exists
-f	is a regular file
-r	is readable
-s	has a size greater than zero
-w	is writable
-x	is executable

If Statement

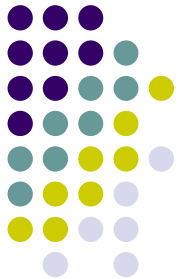


```
if [ ... ]
then
# if-code
else
# else-code
fi
```

- Second form of if-then. Note that you need a ; to separate “test” and then

```
if [ ... ]; then
# do something
fi
```

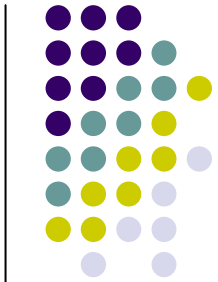
Test with if Examples



- test.sh

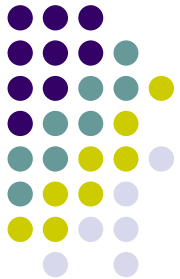
```
#!/bin/sh
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
```

Test with if Examples (Cont.)



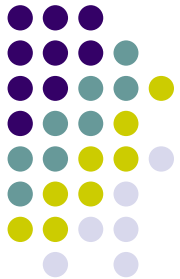
```
[ "$X" -le "0" ] && echo "X is less than or equal
  to  zero"
[ "$X" -ge "0" ] && echo "X is more than or equal
  to zero"
[ "$X" = "0" ] && echo "X is the string or number
  \"0\""
[ "$X" = "hello" ] && echo "X matches the string
  \"hello\""
[ "$X" != "hello" ] && echo "X is not the string
  \"hello\""
[ -n "$X" ] && echo "X is of nonzero length"
[ -f "$X" ] && echo "X is the path of a real file"
  || echo "No such file: $X"
[ -x "$X" ] && echo "X is the path of an
  executable file"
[ "$X" -nt "/etc/passwd" ] && echo "X is a file
  which is newer than /etc/passwd"
```

Additional Notes on &&



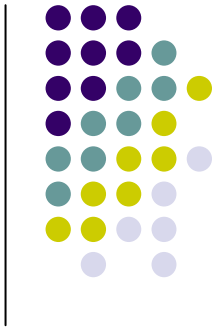
- && and || are command list seperators
- command1 && command2
  - command2 is executed if, and only if, command1 returns an exit status of zero.
- command1 || command2
  - command2 is executed if and only if command1 returns a non-zero exit status.
- The return status of AND and OR lists is the exit status of the last command executed in the list.

Testing test.sh



```
$ X=5
$ export X
$ ./test.sh
... output of test.sh ...
$ X=hello
$ ./test.sh
... output of test.sh ...
$ X=test.sh
$ ./test.sh
... output of test.sh ...
```

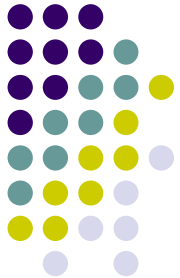
Another Case Example



- talk.sh

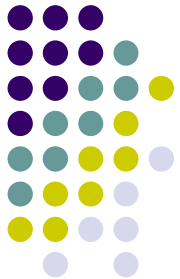
```
#!/bin/sh
echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
   hello)
        echo "Hello yourself!"
        ;;
   bye)
        echo "See you again!"
        break
        ;;
   *)
        echo "Sorry, I don't understand"
        ;;
  esac
done
echo
echo "That's all folks!"
```

Predefined Variable Symbols



- $0 is the *basename* of the program as it was called.
- $1 .. $9 are the first 9 additional parameters the script was called with.
- The variable $@ is all parameters $1 .. whatever.
- $# is the number of parameters the script was called with.

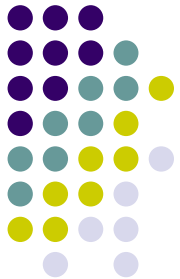
The Script Showing Predefined Variables



- var3.sh

```
#!/bin/sh
echo "I was called with $#
  parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

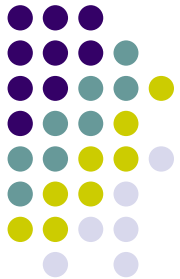
Testing var3.sh



```
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

- echo "My name is `basename $0`"

Shift and $?



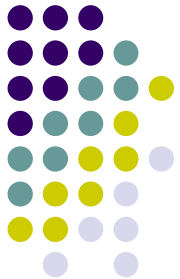
- The following script keeps on using shift until $# is down to zero, at which point the list is empty.
- var4.sh

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
done
```

- $?
  - contains the exit value of the last run command. So the code:

```
#!/bin/sh
/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
  echo "Sorry, we had a problem there!"
fi
```

Examples of Calling External Programs



- To find the full name of the current user

```
$ grep "^${USER}:" /etc/passwd | cut -d: -f5
Steve Parker
```

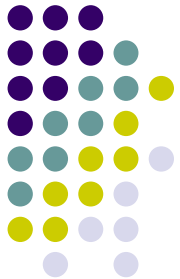
- Assign the output to a variable

```
$ MYNAME=`grep "^${USER}:" /etc/passwd | cut -d: -
  f5`
$ echo $MYNAME
Steve Parker
```

- Saving output for later process

```
#!/bin/sh
HTML_FILES=`find / -name "*.html" -print`
echo $HTML_FILES | grep "/index.html$"
echo $HTML_FILES | grep "/contents.html$"
```

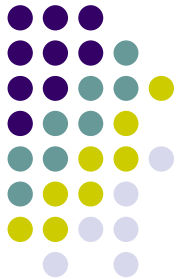
Shell Functions Examples



- function.sh

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{
  USER=$1
  PASSWORD=$2
  shift; shift;
  # Having shifted twice, the rest is now comments ...
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with password
   $PASSWORD"
}
```

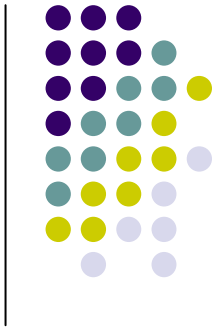
Shell Functions Examples



- function.sh (cont.)

```
###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein Bob Holness the presenter
add_a_user fred badpassword Fred Durst the singer
add_a_user bilko worsepassword Sgt. Bilko the role
  model
echo "End of script..."
```

Variable Scope



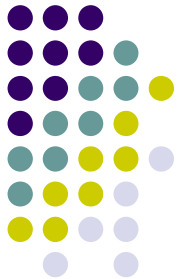
- Every variables are global

```
#!/bin/sh
myfunc()
{
echo "I was called as : $@"
x=2
}
### Main script starts here
echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

- $ scope.sh a b c

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

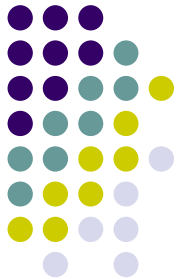
Recursion



- factorial.sh

```
#!/bin/sh
factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}
while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

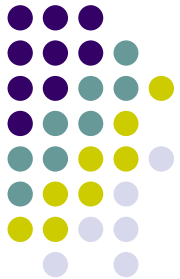
Function Used as Library



- common.lib

```
# common.lib
# Note no #!/bin/sh as this should not spawn an extra shell.
# It's not the end of the world to have one, but clearer not to.
#
STD_MSG="About to rename some files..."

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=`basename $i $FROM`
    mv $i ${j}$TO
  done
}
```

Function Library Examples



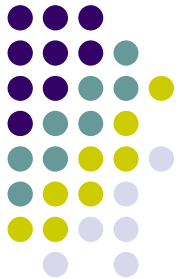
- function2.sh

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

- function3.sh

```
#!/bin/sh
# function3.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

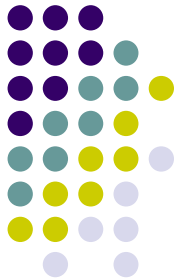
Function with “Return” Codes



- #!/bin/sh

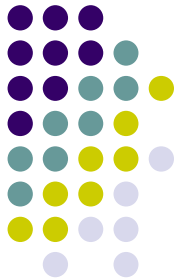
```
adduser()
{
  USER=$1
  PASSWD=$1
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with password $PASSWORD"
}
```

Return Codes Example



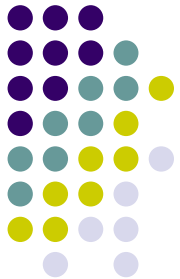
```
## Main script starts here

adduser bob letmein Bob Holness, the famous
  BlockBusters presenter!
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
else if [ "$?" -eq "2" ]; then
  echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

Exit Code Example for Scripts



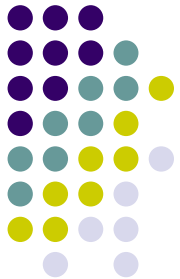
```
#!/bin/sh
# A Tidier approach

check_errs()
{
  # Function. Parameter 1 is the return code
  # Para. 2 is text to display on failure.
  if [ "${1}" -ne "0" ]; then
    echo "ERROR # ${1} : ${2}"
    # as a bonus, make our script exit with the
  right error code.
    exit ${1}
  fi
}
```

Exit Code Example for Scripts



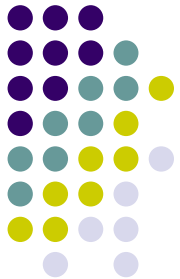
```
### main script starts here ###

grep "^${1}:" /etc/passwd > /dev/null 2>&1
check_errs $? "User ${1} not found in /etc/passwd"
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
check_errs $? "Cut returned an error"
echo "USERNAME: $USERNAME"
check_errs $? "echo returned an error - very
  strange!"

This allows us to test for errors 3 times, with
  customized error messages,
```

Sed and Awk



- Sed: a stream editor
  - To automate editing actions to be performed on one or more files.
  - To simplify the task of performing the same edits on multiple files.
  - To write conversion programs.
- Awk: A pattern-matching programming language
  - View a text file as a textual database made up of records and fields.
  - Use variables to manipulate the database.
  - Use arithmetic and string operators.
  - Use common programming constructs such as loops and conditionals.
  - Generate formatted reports.
  - Define functions.
  - Execute UNIX commands from a script.
  - Process the result of UNIX commands.
  - Process command-line arguments more gracefully.
  - Work more easily with multiple input streams.

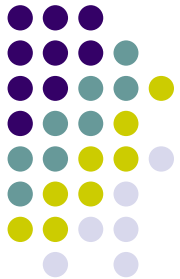
Simple Sed Examples



- sed [-e] '*instruction*' *file*
  - -e: to specify an “instruction” will follow
- $ sed 's/MA/Massachusetts/' list
  - to replace "MA" with "Massachusetts."

```
John Daggett, 341 King Road, Plymouth Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston Massachusetts
```

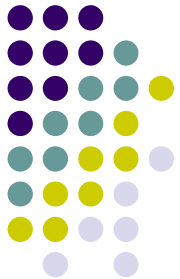
Multiple Sed Commands



- The following two commands are the same
  - `sed 's/ MA/, Massachusetts/; s/ PA/, Pennsylvania/' list`
  - `sed -e 's/ MA/, Massachusetts/' -e 's/ PA/, Pennsylvania/' list`

```
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

Using Sed Script



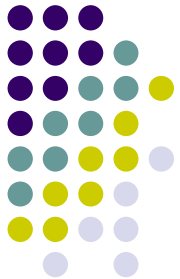
- $ cat sedscr

```
s/ MA/, Massachusetts/
s/ PA/, Pennsylvania/
s/ CA/, California/
s/ VA/, Virginia/
s/ OK/, Oklahoma/
```

- $ sed -f sedscr list

```
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond, Virginia
Orville Thomas, 11345 Oak Bridge Road, Tulsa, Oklahoma
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke, Virginia
Amy Wilde, 334 Bayshore Pkwy, Mountain View, California
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

Simple Awk Examples



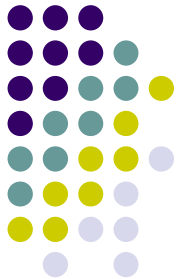
- **awk** '*instructions*' *files*
- **awk -f** *script files*
- $ awk '{ print $1 }' list

```
John
Alice
Orville
Terry
Eric
Hubert
Amy
Sal
```

- $ awk '/MA/' list

```
John Daggett, 341 King Road, Plymouth MA
Eric Adams, 20 Post Road, Sudbury MA
Sal Carpenter, 73 6th Street, Boston MA
```

Awk Examples (Cont.)



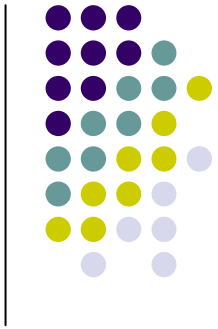
- $ awk '/MA/ { print $1 }' list

```
John
Eric
Sal
```

- $ awk -F, '/MA/ { print $1 }' list

```
John Daggett
Eric Adams
Sal Carpenter
```

An Awk Script



- grades.awk

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

- Example file

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

- $ awk -f grades.awk grades

```
john 87.4
andrea 86
jasper 85.6
```