Introduction to Writing and Using Shell Scripts

Neale Ferguson

SINE NOMINE ASSOCIATES

Course Outline

- **Take a "real life" situation**
- **Create a shell script to implement it**
- **Incremental approach**
  - Work through a topic
  - Apply it to the example
  - Move on to the next topic
  - Repeat
- **Investigate**
  - Shells
  - Environment variables
  - File manipulation
  - Scripting language constructs

The Application

- **Simple Report Program**
  - Read 3 files according to day of the week (M-F)
  - Concatenate data and write to output file
  - Optional parameter to act as report header
  - Optional parameter to identify job run
- **Job Control**
  - Specify day of week for which report is to be run
  - Allow run for entire week
  - Choose between “production” and “QA” runs
  - Write log messages to terminal or to a file
  - Write output to a file in a directory named after user
  - Debug option to show “JCL” preparation
  - Handle abnormal termination

How do we do this in Linux?


```
//REPORT    JOB       51315,
                      NEALE,
                      MSGLEVEL=(1,1)
//RPT       EXEC      PGM=REPORT,PARM='Report Title'
//SYSLIB    DD        DSN=HOME.NEALE,DISP=SHR
//SYSPRINT  DD        SYSOUT=*
//IN1       DD        DSN=TMP.PROD.MON.IN001,DISP=SHR
//IN2       DD        DSN=TMP.PROD.MON.IN002,DISP=SHR
//IN3       DD        DSN=TMP.PROD.MON.IN003,DISP=SHR
//OUT       DD        DSN=TMP.PROD.MON.NEALE(OUT),DISP=SHR
/*
```

```
//REPORT    JOB       51315,
                      NEALE,
                      MSGLEVEL=(1,1)
//RPTPROC   PROC      RUN=,DAY=,TITLE=
//RPT       EXEC      PGM=REPORT,PARM='&TITLE.'
//SYSLIB    DD        DSN=HOME.NEALE,DISP=SHR
//SYSPRINT  DD        SYSOUT=*
//IN1       DD        DSN=TMP.&RUN..&DAY..IN001,DISP=SHR
//IN2       DD        DSN=TMP.&RUN..&DAY..IN002,DISP=SHR
//IN3       DD        DSN=TMP.&RUN..&DAY..IN003,DISP=SHR
//OUT       DD        DSN=TMP.&RUN..&DAY..NEALE(OUT),DISP=SHR
//          PEND
/*
//MONRPT    EXEC      PROC=RPTPROC,RUN=PROD,DAY=MON,TITLE='Report Title'
//TUERPT    EXEC      PROC=RPTPROC,RUN=PROD,DAY=TUE,TITLE='Report Title'
/*
```

Running the Application on Linux…

- `report` Program takes several parameters:

```
report -e <var> -t <title>

where:
    -e  - Passes the name of an environment variable to
          program
    -t  - Passes a string to be used as the report title
```

…Running the Application on Linux

- Without a script…

```
export SYSIN_1=$HOME/tmp/Testing/Monday/Input.001
export SYSIN_2=$HOME/tmp/Testing/Monday/Input.002
export SYSIN_3=$HOME/tmp/Testing/Monday/Input.003
export SYSOUT=$HOME/tmp/Testing/Monday/neale/Output
export REPORT=MON
export PATH=$PATH:.
report -e REPORT -t “Monday Report”
```

```
SYSIN_1=$HOME/tmp/Testing/Monday/Input.001 \
SYSIN_2=$HOME/tmp/Testing/Monday/Input.002 \
SYSIN_3=$HOME/tmp/Testing/Monday/Input.003 \
SYSOUT=$HOME/tmp/Testing/Monday/neale/Output \
REPORT=MON \
PATH=$PATH:. \
report -e REPORT -t “Monday Report”
```

- What do all these statements mean?

Lab Setup

- Click on the “PuTTY” icon
- Select the “Linux Lab” menu item
- Click on “Load” and then “Open” buttons
- Logon as `student`*`nn`* with password `linx101`

Lab – Getting a feel for things…

- Try running the program and see what happens:

```
➢report -e REPORT -t “Monday Report”
```

```
➢PATH=$PATH:. \
report -e REPORT -t “Monday Report”
```

```
➢SYSIN_1=$HOME/tmp/Testing/Monday/Input.001 \
➢REPORT=MON \
➢PATH=$PATH:. \
➢report -e REPORT -t “Monday Report”
```

```
➢SYSIN_1=$HOME/tmp/Testing/Monday/Input.001 \
SYSIN_2=$HOME/tmp/Testing/Monday/Input.002 \
SYSIN_3=$HOME/tmp/Testing/Monday/Input.003 \
SYSOUT=Output \
REPORT=MON \
PATH=$PATH:. \
report -e REPORT -t “Monday Report”
```

…Lab – Getting a feel for things

- Place the following lines in a file called "`monday.sh`"

```
#!/bin/bash
SYSIN_1=$HOME/tmp/Testing/Monday/Input.001 \
SYSIN_2=$HOME/tmp/Testing/Monday/Input.002 \
SYSIN_3=$HOME/tmp/Testing/Monday/Input.003 \
SYSOUT=Output \
REPORT=MON \
PATH=$PATH:. \
report -e REPORT -t "Monday Report"
```

- Run the program: `sh monday.sh`
- What happens if you put a space after any of those trailing '\' characters?

Our Objective

- `report.sh` script that takes several parameters and invokes `report` program

```
report -d -h -e -t <title> -l <log> -x <err> -q days…

where:
    -d  - Turns on debug mode
    -h  - Prints this message
    -e  - Passes the name of an environment variable to
          program
    -t  - Passes a string to be used as the report title
    -l  - Specifies a log file for messages
    -x  - Specifies a log file for error messages
    -q  - Specifies this is a QA (testing) run
    days    - The names of the days of the week for the report
        Any or all of the following (case insensitive) -
        MONday, TUEsday, WEDnesday, THUrsday, FRIday, ALL
```

Shells

- An interface between the Linux system and the user
- Used to call commands and programs
- An interpreter
- Powerful programming language
  - "Shell scripts" = .bat .cmd EXEC REXX

Shells

- sh — Bourne shell – the original
- csh — C shell – compatible with Bourne shell
- bash — Bourne again shell – most common on Linux
- tcsh — The enhanced C shell
- zsh — Z shell – new, compatible with Bourne shell
- ksh — Korn shell – most popular UNIX shell

Why Do I Care About The Shell?

- **Shell is Not an Integral Part of O/S**
  - UNIX Among First to Separate
  - Compare to MS-DOS, Mac, Win95, VM/CMS
  - GUI is NOT Required
  - Shell is just a command (usually living in **`/bin`**)
  - Default Shell Can Be Configured
    - **`chsh -s /bin/bash`**
    - **`/etc/passwd`**
  - Can swap between at will by invoking the name of the shell
  - Helps To Customize Environment

Shell Scripts


```
#!/bin/bash
while
true
do
   cat somefile > /dev/null
   echo .
done
```

```
/* */
do forever
   'PIPE < SOME FILE | hole'
   say '.'
end
```

Environment Variables

- Environment variables are global settings that control the function of the shell and other Linux programs. They are sometimes referred to global shell variables.
- Each process has access to its own set of environment variables
- Variables may be made available between parent and child processes by “exporting” them
- Setting:
  - `VAR=/home/fred/doc`
  - `export TERM=ansi`
  - `SYSTEMNAME=`uname -n``

Environment Variables

- **Using Environment Variables:**
  - `echo $VAR`
  - `cd $VAR`
  - `cd $HOME`
  - `echo "You are running on $SYSTEMNAME"`
- **Displaying – use the following commands:**
  - `set` (displays local & environment variables)
  - `export`
  - `env`
- **Variables can be retrieved by a script or a program**

Some Important Environment Variables

- **HOME**
  - Your home directory (often be abbreviated as “~”)
- **TERM**
  - The type of terminal you are running (for example vt100, xterm, and ansi)
- **PWD**
  - Current working directory
- **PATH**
  - List of directories to search for commands

PATH Environment Variable

SINE NOMINE ASSOCIATES

- **Controls where commands are found**
  - PATH is a list of directory pathnames separated by colons. For example:

    `PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin`
  - If a command does not contain a slash, the shell tries finding the command in each directory in PATH. The first match is the command that will run
- **Usually set in `/etc/profile`**
- **Often modified in `~/.profile` or `~/.bashrc` or `~/.login`**

Lab

- Use `set`/`export`/`env` to display current variables
- Set your own variables

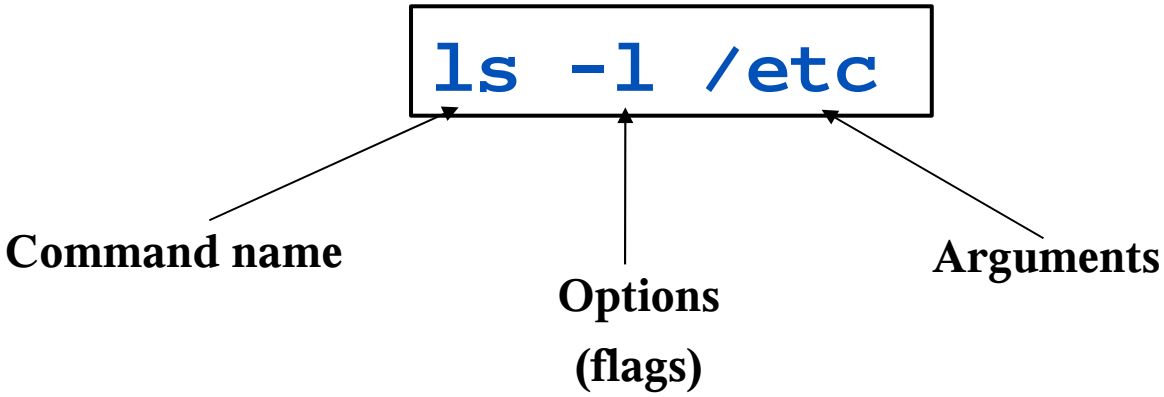
```
ENVVAR="MYVAR"; echo $ENVVAR
echo $ENVVAR
export ENVVAR="MYVAR"
echo $ENVVAR
export ENVVAR=""
```

- Examine effect of PATH

```
date
PATH=/tmp date
```

Linux Command Basics

- To execute a command, type its name and arguments at the command line



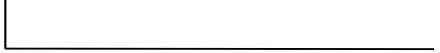
Standard Files

- **UNIX concept of "standard files"**
  - standard input (where a command gets its input) – default is the terminal. Represented by file descriptor 0.
  - standard output (where a command writes it output) – default is the terminal. Represented by file descriptor 1.
  - standard error (where a command writes error messages) – default is the terminal. Represented by file descriptor 2.

Redirecting Output

- The output of a command may be sent to a file:

```
ls -l >output
```



"**>**" is used to specify the output file

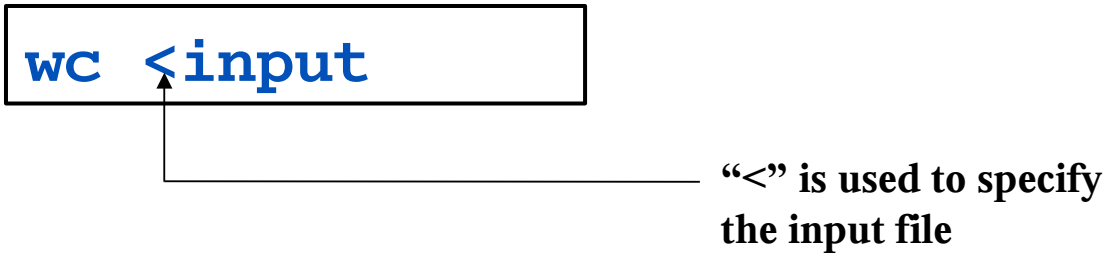
- To redirect the output of standard error use `2>`
- To append to an existing file use `>>`

How our Script uses it

```
if [ $xflag -eq 0 ]; then
    if [ $lflag -eq 0 ]; then
        report "$VSTR" "$TSTR"
    else
        report "$VSTR" "$TSTR" >>$LOGFILE 2>&1
    fi
else
    if [ $lflag -eq 0 ]; then
        report "$VSTR" "$TSTR" 2>>$ERRFILE
    else
        if [ $LOGFILE = $ERRFILE ]; then
            report "$VSTR" "$TSTR" >>$LOGFILE 2>&1
        else
            report "$VSTR" "$TSTR" >>$LOGFILE 2>>$ERRFILE
        fi
    fi
fi
```

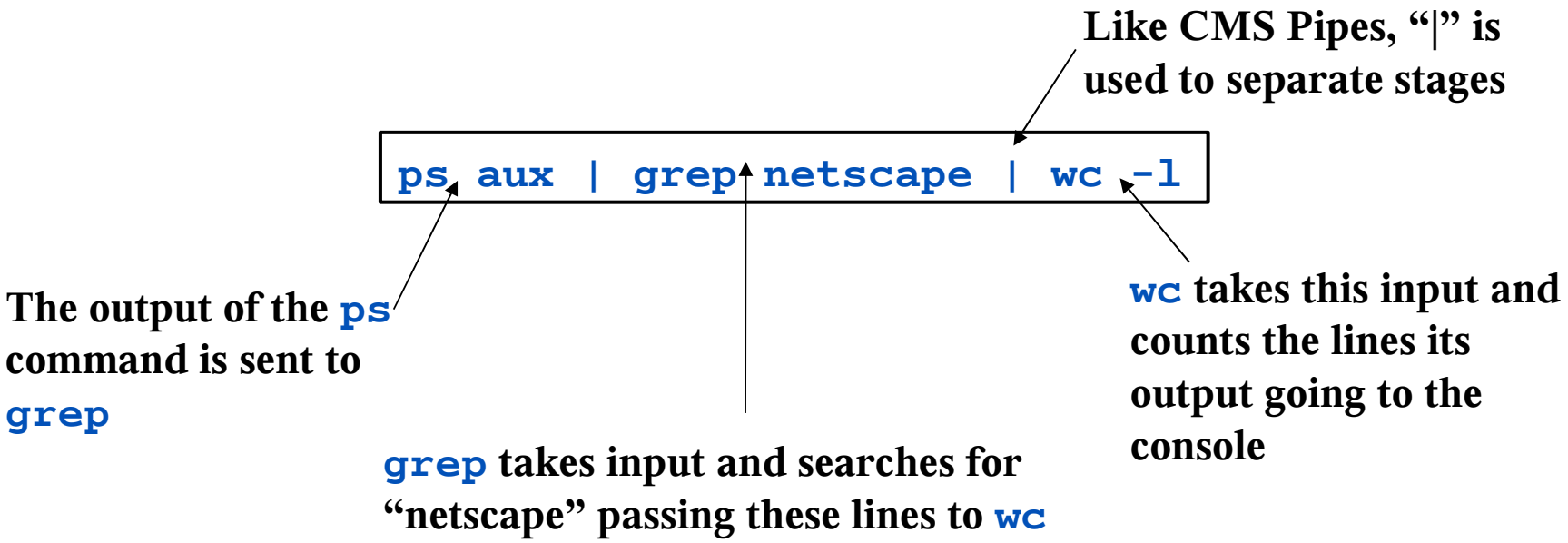
Redirecting Input

- The input of a command may come from a file:



Connecting commands with Pipes

- Not as powerful as CMS/TSO Pipes but the same principle
- The output of one command can become the input of another:



How our Script uses it

```
DAYS=`echo $* | tr '[:lower:]' '[:upper:]'`
```

Command Options

- **Command options allow you to control a command to a certain degree**
- **Conventions:**
  - Usually/Historically: a single dash and are a single letter ("`-l`")
  - POSIX standards complying: double dashes followed by a keyword ("`--help`")
  - Sometimes follow no pattern at all

Language Structures - Agenda

- Terms and concepts
- Statement types
- Invoking a shell program
- System commands
- Logic constructs
- Arithmetic and logic operators
- Functions and subroutines
- Debugging

Terms and Concepts

- BASH = "Bourne Again SHell"
- A shell script is an ordinary text file containing commands that will eventually be read by the shell
- Generally used to startup, control and/or terminate application programs and system daemons
- An interpreted language
- The first line of the program identifies the interpreter: Using `#!/bin/<shell>` ("sh-bang") –
  - `#!/bin/bash2`
  - `#!/bin/sh`
  - If file does not have "**x**" privileges then: `sh <pathname>`

Shell

- **Most Linux commands are files**
  - e.g. **`ls`** is found in **`/bin/ls`**
- **Shell also has built-in commands**
  - **`export`**
  - **`cd`**
- **Needed –**
  - As a programming language construct
  - To be able to operate if PATH setting is invalid
- **Is it a command or is it a built-in?**
  - **`which <command>`**

Lab…

- Create a simple script “hw.sh”

```
echo “Hello World”
```

- Run the script:
  - `hw.sh`
  - `./hw.sh`
  - `bash hw.sh`
  - `csh hw.sh`
- Make the file executable:
  - `chmod +x hw.sh`
  - `./hw.sh`

…Lab

- Update script to look like:

```
#!/bin/csh
echo “Hello World”
set I=0
switch ($I)
   case 0:
      echo “Zero”
      breaksw
endsw
```

- Run again:
  - `./hw.sh`
  - `bash hw.sh`
  - `csh hw.sh`

Comments

- A comment begins with the string `#` and ends with the end of the line
- A comment cannot span multiple lines
- It can appear on the same line as an executable statement

```
J=$(($J+1))    # Increment secondary counter
```

- It cannot be embedded in the middle of an executable statement

Simple Variables

- **Symbols when first defined must begin with an alphabetic or special character "_"**
  - Symbols may contain alphabetic, special, and numeric
- **Symbols referred to by $<symbol name>:**

```
X=1
echo $X
```

- **Symbols are case-sensitive**
  - **`$fred`** is not the same symbol as **`$Fred`** is not the same symbol as **`$FRED`**
- **Symbols that have never been assigned a value have a default of ""**
- **Variables can be read from standard input using "`read <var>`"**

Set variable to result of command

- **Use the "tick" format of assignment to set a variable to the result of a command:**

```
#!/bin/bash
MACHINE=`uname -m`
echo $MACHINE
```

Yields…

```
s390x
```

Single and Double Quotes

- Without quotes

```
MY_VAR='This is my text'
echo $MY_VAR
This is my text
```

- Using double quotes

```
echo "$MY_VAR"
This is my text
```

- Using single quotes

```
echo '$MY_VAR'
$MY_VAR
```

- Why use double quotes?

```
x="school bag"
if [ $x = "abc"]; then
```
versus
```
if [ "$x" = "abc"]; then
```

Lab

- Examine the difference of using double quotes in the test of $x

```
#!/bin/sh
set -x
x="school bag"
if [ $x == "abc" ]; then
        echo "Strange!"
fi
```

- Examine the difference of using single quotes

```
#!/bin/sh
set -x
x="school bag"
if [ ‘$x’ != “school bag" ]; then
        echo "Stranger!"
fi
```

How our Script uses it


```
ID=`whoami`
vflag=0
tflag=0
lflag=0
xflag=0
```

```
ddName() {
    export $1=$2
    message $INFO "$1 has been assigned to $2"
}
```

```
if [ $TITLE = "@" ]; then
    read USRTITLE
    TSTR="-t$USRTITLE"
    message $INFO "Report title set to $USRTITLE"
else
```

How our Program uses it


```
for (i_fd = 0; i_fd < 3; i_fd++) {
    sprintf (ddName, "SYSIN_%d", i_fd+1);
    in[i_fd] = getenv(ddName);
    if (in[i_fd] != NULL) {
        inFd[i_fd] = open(in[i_fd],O_RDONLY);
        if (inFd[i_fd] < 0) {
            err = errno;
            fprintf(stderr, "Error opening %s - %s\n",
                in[i_fd],strerror(errno));
            return (-err);
        }
    } else {
        fprintf(stderr, "DD name missing for %s\n",ddName);
        return (-1);
    }
}
```

Assignment

- **The equal sign = is used as the assignment operator**

```
i=3
j="A string"
k_q=`expr $i + 2` or k_q=$(($i+2)) or let k_q=$i+2
```

- **It is also used as the comparison operator for numeric equality**

```
if [ $i == 4 ]…
_equal =`expr $i == 4` or _equal=$(($i==4))
```

  - Usage is determined from context
    - The last statement above sets the variable **`_equal`** to 'true' or 'false' (1 or 0) depending on whether **`$i`** equals 4

Array Variables

- Arrays of values are implemented using:

```
#!/bin/bash2
Y=0
X[$Y]="Q"
echo ${X[$Y]}
-----------------------------
Q
```

How our Script uses it


```
INFO=0
WARN=1
ERRA=2
MSGPRI[$INFO]="info"; MSGPRI[$WARN]="warn"; MSGPRI[$ERRA]="err“
MSGIND[$INFO]="I"; MSGIND[$WARN]="W"; MSGIND[$ERRA]="E"
STAT[$INFO]=0; STAT[$WARN]=0; STAT[$ERRA]=0
```

```
message() {
        PRI=$1
        shift
        TOD=`date +"%F %T"`
        echo "$TOD $ID ${MSGIND[$PRI]} $*"
        logger -i -t report -p ${MSGPRI[$PRI]} "$*"
        STAT[$PRI]=$((STAT[$PRI] + 1))
}
```

```
stats() {
        msg="${STAT[$INFO]} informational, "
        msg="$msg ${STAT[$WARN]} warning(s), "
        msg="$msg ${STAT[$ERRA]} error(s)"
        message $INFO "Message statistics: $msg"
}
```

Syntax

- **A script may have parameters and options using the same syntax as normal commands**

```
foo -anycase .therc
```

  - might perform the foo function on file **`.therc`**, ignoring case
- **We must be able to perform the usual functions of a program:**
  - access the parameter string
  - produce output
  - exit the program when done

Accessing Parameters

- Parameters are identified by `$0`, `$1`, `$2`...
- `$0` returns the name of the script
- `$#` returns number of arguments
- `$*` returns all arguments
- The `set` function can assign values to `$0` etc.
- The `shift` function makes `$1=$2`, `$2=$3` etc.

Lab

- **Write a script:**
  - Displays the script name
  - Displays the number of parameters
  - Displays the parameters passed
  - Use the shift command to shuffle the parameters down by 3 and display the new 1st parameter

Accessing Parameters

- Use `getopt` function to resolve flags and operands:

```
getopt <flags> <result>
```

```
while getopts put: opt
do
   case "$opt" in
    p) _autoload_dump printable; return 0;;
    u) _autoload_unset=y ;;
    t) _autoload_opt="$OPTARG" ;;
    *) echo "autoload: usage:"
       echo "  autoload [-put<opt>] [function ...]" >&2
      return 1 ;;
   esac
done
shift $(($OPTIND-1))
```

The echo Instruction

- One way to produce output from a program is simply to display it on the terminal or monitor
- The echo instruction is used to do this

  `echo` *expression*
  - evaluates the expression and displays its value
- For example

```
echo “Hello World!”
X=“XYZ”
echo $X
-----------------------------------
Hello World!
XYZ
```

Tracing the Program

- Prior to executing:

```
set -x
```

- Option of `sh` command:

```
sh -x <shellscript>
```

- Within a script:

```
#!/bin/sh
set -x
echo $0
```

How our Script uses it

```
while getopts dehl:t:qx: name
do
    case $name in
        d)  set -x;;
```

Terminating the Program...

- **The exit instruction terminates the program immediately.**
- **It takes an optional parameter of a return code**
  - The return code must be an integer
  - It may be positive, negative, or zero

```
echo "File not found"
exit 28
```

Structure and Logic

- **Several programming constructs are available in the shell language**
  - The loop constructs
    - At least five unique forms exist
    - They can be combined to produce interesting results
  - The **`case`** … **`esac`** construct
    - Used to execute one of a set of mutually exclusive code fragments
  - The **`if`**/**`then`**/**`fi`** and **`if`**/**`then`**/**`else`**/**`fi`** constructs
    - The **`else`** clause is optional
    - The forms may be nested to execute complex logical operations

Tests

- The test may deal with file characteristics or numerical/string comparisons.
- Although the left bracket here appears to be part of the structure, it is actually another name for the Unix test command (located in `/bin/[`).
- Since `[` is the name of a file, there must be spaces before and after it as well as before the closing bracket.

Comparison Functions

- *TEST OPTIONS – FILE TESTS*
  - `-s <file>` Test if file exists and is not empty.
  - `-f <file>` Test if file is an ordinary file, not a directory.
  - `-d <file>` Test if file is a directory.
  - `-w <file>` Test if file has write permission.
  - `-r <file>` Test if file has read permission.
  - `-x <file>` Test if file is executable.
  - `!` "Not" operation for test.

Comparison Functions

- *TEST OPTIONS – STRING COMPARISONS*
  - `$X -eq $Y` $X is equal to $Y.
  - `$X -ne $Y` $X is not equal to $Y.
  - `$X -gt $Y` $X is greater than $Y.
  - `$X -lt $Y` $X is less than $Y.
  - `$X -ge $Y` $X is greater than or equal to $Y.
  - `$X -le $Y` $X is less than or equal to $Y.
  - `"$A" = "$B"` String $A is equal to string $B.

Comparison Functions

- *TEST OPTIONS – NOT (!)*
  - `"$A" != "$B"` String $A is not equal to string $B.
  - `$X ! -gt $Y` $X is not greater than $Y.

The Simple do…done Group

- **A group of statements may be preceded by a `do` statement and followed by an `done` statement**
  - This allows the group of statements to be treated as a unit
  - No change in the execution of the statements is produced
- **The entire set of statements between the `do` and `done` is executed if *condition* is true**

Looping Conditionally

- An `until` loop always executes at least once
- A `while` loop will not execute at all if *condition* is false at initial entry to the `while` statement

```
while condition
do
   statements
done

while condition; do; statements; done

until condition
do
   statements
done

until condition; do; statements; done
```

While 1 -- an Unending Loop

- The `while 1` or `until 0` construct will loop forever
- Used when the termination condition is not known
- The termination condition (if any) is found inside the group

```
while [ 1 ];
do
    ….
    if [ condition ]; then
          break
    fi
done
```

The break Instruction

- The `break` instruction is used to exit an iterative loop
- By default, it exits the innermost loop if it is executed inside nested loops then `break` *n* will exit out of *n* levels of loops
- If *n* is greater than the level of nesting then all levels are exited

The Case Construct…

- Many programming languages have a construct that allow you to test a series of conditions and execute an expression when a true condition is found

```
case $key in          Match the variable $key.
       pattern1)              Test match to pattern1.
           statement          If $key matches pattern1, then
                                execute statement
       ;;                     Each pattern ends with ;;.
       pattern2)              Test match to pattern2
           statement              If match, then execute
           statement
       ;;
esac           Close the case with esac.
```

The Case Construct

- **The first condition that evaluates as "true" causes its corresponding expression to be executed**
  - Control then transfers to the end of the **`case`** group
  - No other conditions are tested
- **The same rules apply here for expressions as apply with the `if/then/else` construct**

Lab

- **Use the `getopts`/`while`/`case` constructs to parse the options of a script that accepts the following options:**
  - `-v` Verbose (no operands)
  - `-t` Title (next operand is the actual title)
  - `-l` Logfile (next operand in the name of a file)
- **Print a messages that tell the user**
  - Whether verbose option was specified
  - The title (if specified)
  - The name of the log file (if specified)
- **http://aussie-1.osdl.marist.edu/report.file**

How our Script uses it


```
while getopts dehl:t:qx: name
do
    case $name in
        d)  set -x;;
        e)  vflag=1;;
        t)  tflag=1
            TITLE="$OPTARG";;
        l)  lflag=1
            LOGFILE="$OPTARG";;
        x)  xflag=1
            ERRFILE="$OPTARG";;
        q)  qflag=1;;
        h)  usage;;
    esac
done
shift $(($OPTIND - 1))
:
DAYS=`echo $* | tr '[:lower:]' '[:upper:]'`
```

Conditional Execution (if/then/else)

- Uses the traditional form of the conditional execution statements

```
if [ test ]
then      then must appear on new line (or use ';')
          command
else      else is optional also on new line
          command
fi    if always finishes with fi

if [ test ]; then command; else command; fi
```

Tests

- **Examples:**

```
if [ $# -ne 1 ]
then
        echo "This script needs one argument."
        exit -1
fi
input="$1"
if [ ! -f "$input" ]
then
        echo "Input file does not exist."
        exit -1
else
        echo "Running program bigmat with input $input."

        bigmat < $input
fi
```

Lab

- **Use the `if/then/else/fi` and `test` constructs to:**
  - Check for the existence of `/etc/profile` and display a message informing the user
  - Read a variable from `stdin` using the read command and compare it against a string “ABORT” and display a message saying whether the comparison is true
  - Repeat the previous test but make the comparison case insensitive

How our Script uses it


```
if [ x$RUNMODE != xProduction ]; then
    message $WARN "Run mode has forced report processing to Testing"
    qflag=1
fi

if [ $qflag -eq 1 ]; then
    DIR="Testing"
else
    DIR="Production"
fi

if [ $xflag -eq 1 ]; then
    rm -f $ERRFILE
fi

if [ $lflag ]; then
    rm -f $LOGFILE
fi
```

Looping Through a List

- There are several forms of a do loop controlled by a counter

```
for variable in list
do
    statement      Execute statement on each loop.
done       Close the do with done.
```

```
for month in "January" "February" "March"
do
   echo $month
done
```

Lab

- Use the for statement to iterate through a list of vegetables: “`carrot`”, “`potato`”, “`turnip`”, “`bean`”, “`pea`”
- Use the if statement to test for the existence of a file in `/tmp` that has the same name as the vegetable
- Display a message telling the user whether that file exists or not

How our Script uses it

SINE NOMINE ASSOCIATES

```
for REPORT in $DAYS; do
    case $REPORT in
        MON|MOND|MONDA|MONDAY)
            Report="Monday"
            runday
            ;;
        :
        ALL)
            for Report in "Monday" "Tuesday" "Wednesday" \
                          "Thursday" "Friday"; do
                runday
                if [ $RC -ne 0 ]; then
                    abort -3
                fi
            done
            ;;
        *)
            usage;;
    esac
done
```

Arithmetic Functions…

- - + unary minus and plus
- ! ~ logical and bitwise negation
- ** exponentiation
- * / % multiplication, division, remainder
- + - addition, subtraction
- << >> left and right bitwise shifts
- <= >= < > comparison
- == != equality and inequality

Arithmetic Expressions

- **&** bitwise AND
- ^ bitwise exclusive OR
- | bitwise OR
- **&&** logical AND
- || logical OR
- *expr*?*expr*:*expr* conditional evaluation
- = *= /= %= +=
  -= <<= >>= **&=**
  ^= |= assignment

More Useful Commands

- **`printf`**
  - Format and print data
- **`sort`**
  - Sort lines of text files (also has a –u for unique sorting)
- **`uniq`**
  - Remove duplicate lines from a sorted file

Subroutines

- Defined before where they are called
- Take parameters $1, $2…
- Can return an integer

```
test() {
    echo "Was passed $1"
    return 0
}

test "First parameter" "Second Parameter"
echo $?
exit
```

Yields…

```
Was passed First parameter
0
```

Lab

- **Create a script which:**
  - Takes a single parameter
  - Based on the value of the parameter call one of 3 subroutines:
    - one – which prints “subroutine one called” and returns 1
    - two – which prints “subroutine two called” and returns 2
    - xxx – which prints “subroutine xxx called with $1” and returns -1
  - The mainline will take the return code from the subroutine and display it and exit with that code

Building the Input Files

SINE NOMINE ASSOCIATES

```
#!/bin/sh
init() {
        i=0
    mkdir -p $HOME/tmp/{Production,Testing}/$1
        for name in $1 $2 $3  # or $*
        do
                i=$(($i+1))
                for dir in "Production" "Testing"
                do
                        echo -n "$name" >$HOME/tmp/$dir/$1/Input.00$i
                done
        done
}

init "Monday" "Montag" "Lundi"
init "Tuesday" "Dienstag" "Mardi"
init "Wednesday" "Mittwoch" "Mercredi"
init "Thursday" "Donnerstag" "Jeudi"
init "Friday" "Freitag" "Vendredi"
exit
```

Debugging Shell Scripts

- **The set instruction is your primary debugging tool**
  - **`set -e`**
    - If a simple command fails the shell shall immediately exit
  - **`set -n`**
    - The shell shall read commands but does not execute them
  - **`set -u`**
    - The shell shall write a message to standard error when it tries to expand a variable that is not set and immediately exit
  - **`set -v`**
    - The shell shall write its input to standard error as it is read
  - **`set -x`**
    - The shell shall write to standard error a trace for each command after it expands the command and before it executes it

Monitoring Jobs…

- **Running jobs in background:**
  - <scriptname> <parameters> &
- **Use `jobs` command to display status**
  - Only on current session
- **Use `ps` command to display process(es)**
  - `ps`
  - `ps -u <user>`
  - `ps -ef`
  - `ps -L`
- **Canceling jobs/processes:**
  - `CTRL-C`
  - `kill [-SIGNAL] %<job number>`
  - `kill [-SIGNAL] <process id>`
  - `killall [-SIGNAL] <process name>`

…Monitoring Jobs

- Redirect script output to file
  - `report.sh … >report.out 2>report.err &`
  - `report.sh … >report.out 2>&1 &`
  - `report.sh … 2>&1 | tee report.out &`
- Monitoring log files:
  - `tail -20f report.out`
  - `tail -20f report.out | grep -i “title”`

Trapping Signals

- Use "`trap`" command to intercept signals
- Used to allow clean-up of job

```
trap "<action>" <signals…>


trap "echo Received a signal; exit -1" TERM
while [ 1 ]
do
    echo -n "."
    sleep 1s
done
```

How our Script uses it


```
abort() {
        rm -f $SYSOUT
        message $ERRA "Job processing terminated abnormally"
        exit $1
}
:
:
trap "abort -4" INT QUIT ABRT TERM
:
# Report processing
:
trap "" INT QUIT ABRT TERM
```

Lab

- **Start the report script using the following:**

```
./report.sh -t "Weekly Report" -l ~/tmp/report.log -e all &
```

- **While the job is running enter:**
  - `jobs` – take note of the job number
  - `kill -ABRT %n` – where *n* is the job number
  - Take note of the termination message from the script and from the shell
- **Start the report script again**
- **While the job is running enter:**
  - `ps -u <user>` – where <user> is your id
  - Take note of the process id (PID)
  - What does the PPID field report
  - Wait a few seconds and enter the above command again
  - What do you notice about the PID/PPID values?
  - What happens if you issue `kill -ABRT <PID>`

Final Grade

- Look at `report.broken`
- Identify and correct all the mistakes:
  - `diff -U5 report.sh report.broken`
- Once fixed rename to `report.new`
- Update this program to process data for Saturday
- Update the appropriate directory structure to support Saturday processing
- Extra credit: Use the mail command to send a note to a user when the job completes
  - Report the completion status

```
mail -t << _EOF
To: user@node.domain
From: BatchSystem

Text
_EOF
```