Korn and Bash Shell Programming

(Course code AL32)

Student Exercises

ERC 1.0

IBM Certified Course Material

Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX®	AIX 5L™	Language Environment®
OS/2®	POWER™	RISC System/6000®
RS/6000®		

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

October 2007 edition

Contents

Trademarks v

Exercises description vii

Exercise 1. Using Shell Basics 1-1

Exercise 2. Variables 2-1

Exercise 3. Testing 3-1

Exercise 4. Shell Programming Constructs 4-1

Exercise 5. Shell Commands and Features 5-1

Exercise 6. Shell Arithmetic 6-1

Exercise 7. Typeset and Functions 7-1

Exercise 8. More Shell Variables 8-1

Exercise 9. Regular Expressions & Data Selection 9-1

Exercise 10. The sed utility 10-1

Exercise 11. Using awk 11-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX®	AIX 5L™	Language Environment®
OS/2®	POWER™	RISC System/6000®
RS/6000®		

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Exercises description

None of the exercises are dependent on the preceding exercise being successfully completed. However, it is assumed that you understand the concepts and commands from each topic, as these will often be used in further exercises.

Exercise Instructions - Each exercise is divided into steps. There are not detailed instructions on how to complete each step. You are given the opportunity to work through each step of the exercise, using what you have learned in the unit presentation. Use your class notes as a reference manual.

Lab Exercise Hints and Answers — This is a separate section in your class notes, which contains the original exercises, with hints, and most, if not all of the answers to each exercise. Several exercises, particularly the later ones, do not have just one solution. The answer given in this section would then be only one of several.

All scripts are available in /**home**/**workshop**.

Each exercise in this course is divided into sections as described below. Select the section that best fits your method of performing labs. You may elect to use a combination of these sections as appropriate.

Exercise Instructions - This section contains what it is you are to accomplish. There are no definitive details on how to perform the tasks. You are given the opportunity to work through the exercise using what you learned in the unit presentation, utilizing the Student Notebook, your past experience and maybe a little intuition.

Exercise Instructions With Hints - This section is an exact duplicate of the Exercise Instructions section except that in addition, specific details and/or hints are provided to help step you through the exercise. A combination of using the Instructions section along with the Instructions With Hints section can make for a rewarding combination providing you with no hints when you don't want them and hints when you need them.

Optional Exercises - This section gives you additional exercises to perform relating to the unit of discussion. It is strictly optional and should be performed when you have completed the required exercises. The required exercises apply to the most important or useful information provided in the unit. This section may help round out the hands-on experience for a related unit.

Exercise 1. Using Shell Basics

What This Exercise is About

The purpose of the exercise is to review basic shell concepts and practice shell scripting.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Recall and edit a previous command line
- Use I/O descriptors and redirection
- Manipulate files using metacharacters and quoting

Introduction

After successfully logging in to the AIX system, the students will explore their environment and review their shell basics.

Exercise Instructions

Logging in

__ 1. Log in to the system with a user name and password provided by your instructor. It should be of the format **teamXX** where the **XX** is a number from 01 through 05 for Korn Shell 88 users. The login for ksh93 is **ksh93XX**, where **XX** is a number from 01 through 05. The login for a bash shell user is **bashXX**, where **XX** is a number from 01 to 05. The first time you log in to the system, you will be prompted to change your password. You may keep the same password or create a new one.

Basic File Manipulation

__ 2. After logging in, create a directory structure that you will use for the rest of the labs. Create four directories in your **$HOME** with the names **awk, tmp, ksh**, and **functions**.

__ 3. There are some setup files in /**home/workshop** for you. They are **.profile**, **.kshrc**, and **.exrc**. Copy them into your **$HOME** using one command line taking advantage of some metacharacters we learned. (If you are using bash, you must copy over **.bash_profile, .bashrc**, and **.exrc**).

__ 4. Examine the three files. You will notice there are a few errors. Make any necessary corrections and/or additions. Log out/in and check to see if the corrections took effect. If not, ask the instructor for help.

Basic Shells

__ 5. Show the value of the variable **$$**. **$$** is the variable that represents the PID of your shell. Run **ps -ef** and find your PID there, too. Record and remember this value. (bash users must use echo instead of print)

__ 6. Try the following commands - press an extra <**ENTER**> after each one completes. Notice the differences in the values of $$ and quotes and explain the differences of the PIDs that are returned.

(instead of ksh, use **ksh93 -c "print $$" &** or **bash -c "print $$" &** if using these shells)

```
$ ksh -c "print $$" &
$ ksh -c 'print $$' &
$ ksh -c 'print $$'
$ _
```

__ 7. Use vi to create a shell script in your **$HOME** that will print **$$**, and then pause for 5 seconds. Run the script four different ways (you will need to change permissions) and note any differences. Which output matches the PID of your current shell?

Command Line Recall and Editing

__ 8. Use the set command to see if a command line editor is on. If it is not on, turn it on. (To turn it on automatically every time you log in, edit your ENV file.) Take the next five to ten minutes or so to play with the **command line recall and editing** features of the shell.

Job Control

__ 9. Start a few background processes, such as **sleep 999&**, and so forth. Bring the **sleep 999&** job to the foreground. Suspend the same job, then place it in the background. Kill all background jobs. Use the jobs command to report the status between your steps.

Metacharacters and Quoting

__ 10. Use metacharacters to pick out file names in the /**home**/**workshop** directory that start with "**b**" then file names that end with "**h**" (**Hint:** use **ls** to display).

__ 11. Now show the files called "**cars1**" and "**cars2**" but no others. Finally, **ls** any files that have only two characters following a "**.**" at the end of their names. Be creative and use some of the features shown in your student guide.

File Descriptors and Re-direction

__ 12. Copy the /**etc**/**motd** to a new file called **letter** under **$HOME** using redirection.

__ 13. Send mail to another user on your system using input redirection rather than command line entry.

__ 14. List a non-existent file called **lmnop** in your **$HOME** directory, then do it again and eliminate any error messages.

__ 15. Create a new document from the command line using **HERE document** syntax. Then display the file.

__ 16. Set file descriptor 4 so that the output is redirected to the file /**tmp**/**yy** where **yy** are your initials. List all files in your directory, using file descriptor 4, so the output goes indirectly to /**tmp**/**yy**. Now associate file descriptor 5 with 4, so that they both output to the same file. Repeat the listing above or execute the date command two more times, using file descriptor 4 and then file descriptor 5 and verify the output goes to where you think.

__ 17. Use **sort** and **more** to display /**home**/**workshop** in reverse sorted order and “paged”. Using a similar pipeline, save the listing into a file called **ls.sort**. Now do the sorted listing again (but without the reverse) and append to **ls.sort**.

END OF LAB

Exercise Instructions With Hints

Logging in

__ 1. Log in to the system with a user name and password provided by your instructor. It should be of the format **teamXX** where the **XX** is a number from 01 through 05 for Korn Shell 88 users. The login for ksh93 is **ksh93XX**, where **XX** is a number from 01 through 05. The login for a bash shell user is **bashXX**, where **XX** is a number from 01 to 05. The first time you log in to the system, you will be prompted to change your password. You may keep the same password or create a new one.

```
$ login: teamXX
password: teamXX
teamXX's new password: teamXX
teamXX's new password again: teamXX
$ exit
```

Basic File Manipulation

__ 2. After logging in, create a directory structure that you will use for the rest of the labs. Create four directories in your **$HOME** with the names **awk, tmp, ksh**, and **functions**.

```
$ login: teamXX
password: teamXX
$ mkdir tmp ksh functions awk
$ _
```

__ 3. There are some setup files in /**home**/**workshop** for you. They are **.profile**, **.kshrc**, and **.exrc**. Copy them into your **$HOME** using one command line taking advantage of some metacharacters we learned. (If you are using bash, you must copy over **.bash_profile**, **.bashrc**, and **.exrc**)

```
$ ls -a
ksh/ksh93 $ cp /home/workshop/.[ekp]* $HOME
bash $ cp /home/workshop/.[eb]* $HOME
$ _
```

__ 4. Examine the three files. You will notice there are a few errors. Make any necessary corrections and/or additions. Log out/in and check to see if the corrections took effect. If not, ask the instructor for help.

HINT: The **.exrc** file has unnecessary colons at the beginning of each line. Remove them. The **.profile**/**.bash_profile** files need to “export” the ENV variable. The **.kshrc**/**.basrc** files contains a misspelling -- fix it.

ALSO: You may copy all or some of the rest of the files from /**home**/**workshop** to your **$HOME** directory structure. Notice they are all owned by root, while in /**home**/**workshop** but ownership changes after you make the copy. You may want to add execute permission to those files now or you’ll need to later.

Basic Shells

__ 5. Show the value of the variable **$$**. **$$** is the variable that represents the PID of your shell. Run **ps -ef** and find your PID there also. Record and remember this value. *(bash users must use **echo** instead of **print**)*

```
$ print $$      (bash users must use echo instead of print))
$ ps -ef
$ _
```

__ 6. Try the following commands - press an extra <**ENTER**> after each one completes. Notice the differences in the values of **$$** and quotes and explain the differences of the PIDs that are returned. *(bash users must use **echo** instead of **print**))*

```
$ ksh -c "print $$" &
$ ksh -c 'print $$' &
$ ksh -c 'print $$'
$ _
```

__ 7. Use vi to create a shell script in your **$HOME** that will print **$$**, and then pause for 5 seconds. Run the script four different ways (you will need to change permissions) and note any differences. Which output matches the PID of your current shell?

```
$ vi prog.ksh (of course, if it's a bash, you may want to name it prog.bash)
   print $$      (bash users must use echo instead of print))
   sleep 5
   <ESC>:wq
$ chmod 744 prog.ksh
$ prog.ksh
$ ksh prog.ksh   (or bash prog.bash)
$ . prog.ksh
$ exec prog.ksh
login:
```

Command Line Recall and Editing

__ 8. Use the set command to see if a command line editor is on. If it is not on, turn it on. To turn it on automatically every time you log in, edit your **ENV** file. Take the next five to ten minutes or so to play with the **command line recall and editing** features of the shell.

```
$ set -o
$ set -o vi     (or set -o emacs in bash)
$ cd /home/workshop
$ <ESC> k ( j, h, l, +, -, /, \, =, *)     (or arrow keys if in bash)
Note: refer to student guide for more information.
$ cd
$ _
```

Job Control

__ 9. Start a few background processes, such as **sleep 999&**, etc. Bring the **sleep 999&** job to the foreground. Suspend the same job, then place it in the background. Kill all background jobs. Use the jobs command to report the status between your steps.

```
$ sleep 9 &
$ sleep 99 &
$ sleep 999 &
$ jobs
$ fg %(Job#)
$ <CTRL>-z
$ jobs
$ kill %(Job#) Note: Do this for each remaining job.
$ jobs
$ _
```

Metacharacters and Quoting

__ 10. Use metacharacters to pick out file names in the /**home**/**workshop** directory that start with "**b**", then file names that end with "**h**" (hint: use **ls** to display).

```
$ cd /home/workshop
$ ls b*
$ ls *h
$ _
```

__ 11. Now show the files called "**cars1**" and "**cars2**" but no others. Finally, **ls** any files that have only two characters following a "**.**" at the end of their names. Be creative and use some of the features shown in your student guide.

```
$ ls cars[12]
$ ls *.??
$ _
```

File Descriptors and Re-direction

__ 12. Copy the /etc/motd to a new file called **letter** under **$HOME** using redirection.

```
$ cd
$ cat /etc/motd > letter
$ _
```

__ 13. Send mail to another user on your system using input redirection rather than command line entry.

```
$ mail teamYY < letter
$ _
```

__ 14. List a non-existent file called **lmnop** in your **$HOME** directory then do it again and eliminate any error messages.

```
$ cd
$ ls lmnop
$ ls lmnop 2> /dev/null
$ _
```

__ 15. Create a new document from the command line using **HERE document** syntax. Then display the file.

```
$ cat  > new.doc << END
> This is the first line.
> This is the second.
> END
$ cat new.doc
$ _
```

__ 16. Set file descriptor 4 so that the output is redirected to the file /**tmp**/**yy** where **yy** are your initials. List all files in your directory, using file descriptor 4, so the output goes indirectly to /**tmp**/**yy**. Now associate file descriptor 5 with 4, so that they both output to the same file. Repeat the listing above or execute the date command two more times, using file descriptor 4 and then file descriptor 5 and verify the output goes to where you think.

```
$ exec 4> /tmp/yy
$ ls * >&4
$ exec 5>&4
$ ls * >&4
$ cat /tmp/yy
$ date >&5
$ cat /tmp/yy
$ _
```

__ 17. Use **sort** and **more** to display /**home**/**workshop** in reverse sorted order and “paged”. Using a similar pipeline, save the listing into a file called **ls.sort**. Now do the sorted listing again (but without the reverse) and append to **ls.sort**.

```
$ ls /home/workshop ¦ sort -r ¦ more
$ ls /home/workshop ¦ sort -r ¦tee ls.sort ; more ls.sort
$ ls /home/workshop ¦ sort ¦ tee -a ls.sort ; more ls.sort
$ _
```

END OF LAB

Solutions

Basic File Manipulation

4. The three files have some (hopefully) obvious errors. The **.exrc** file still has the colons in the first column, the **.kshrc** file has “alias” spelled incorrectly, and **.profile** has no export command,

Basic Shells

6. The command using the single quote prints the **$$** value of the sub-shell. Within double quotes, **$$** prints the value of the current shell.

Command Line Recall and Editing

8. If **vi** does not appear in the **set -o** list, check your **.profile** to see if it matches what is in /**home**/**workshop**/**.profile**. Recopy it to **$HOME** if it is different. (for bash, you are looking for emacs to be on)

If you are using CDE, edit **$HOME**/**.dtprofile** and remove the comment mark from the last line. Then log out and log in again.

If using an X station, you may need to make the xterm(s) behave as login shells. Do this by running the **custom** tool and changing the aixterm properties, specifically the login shell behavior to **true**.

Exercise 2. Variables

What This Exercise is About

The purpose of this exercise is to practice shell scripting using variables and parameters.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Set and reference variables
- Export variables

Exercise Instructions

Variables

__ 1. Set the variables **var1** and **var2** to the values of “1” and “2” respectively, then display the values of each. Make **var1** read only. Now change the value of **var1** to 10. Why did it not work? Using **var1** and the zero character, display the whole number between 9 and 11. (bash users: Remember you must use echo, not print with bash!)

__ 2. Write a shell script that prints its eleventh positional parameter (PP) without using a **shift**. Write another that prints its eleventh PP by using a **shift** command, then display the eleventh PP as **$1**. Finally, create another script by adding the command to reset the PPs to “A”, “B”, and “C”, then display them using **$***. (Remember to use echo in bash)

__ 3. Create the program **param11d.ksh** to display the name of the program and the number of PPs.

__ 4. List all your environment variables. Now list the ones that are exported and available to a shell script. Set the **CDPATH** to include / and /**home/teamXX**. Try to **cd** to “etc” (note there is no “/”). Change your primary prompt to display your command number. Customize your **MAILMSG** variable. Send new mail to yourself and verify. Display the values of **$SECONDS** and **$RANDOM**. Now do it again and explain why they are different. Log out.

__ 5. Edit your .profile. Add a line that sets a variable so your primary prompt reflects your present working directory. (Careful which set of quotes you use!)

__ 6. Test if the export command passes variables down to several levels of subshells. What if you change the value along the way?

END OF LAB

Exercise Instructions With Hints

Variables

__ 1. Set the variables **var1** and **var2** to the values of "1" and "2" respectively, then display the values of each. Make **var1** read only. Now change the value of **var1** to 10. Why did it not work? Using **var1** and the zero character, display the whole number between 9 and 11. (bash users: Remember you must use **echo**, not **print** with bash!)

```
$ var1=1 ; var2=2 ; print $var1 $var2
$ readonly var1 ( or typeset -r var1 )
$ var1=10
$ print ${var1}0
$ _
```

__ 2. Write a shell script that prints its eleventh positional parameter (PP) without using a **shift**. Write another that prints its eleventh PP by using a **shift** command, then display the eleventh PP as **$1**. Finally, create another script by adding the command to reset the PPs to "A", "B", and "C", then display them using **$***. (Remember to use echo in bash)

```
$ vi param11a.ksh
   print ${11}
$ param11a.ksh 1 2 3 4 5 6 7 8 9 10 11 12
$ vi param11b.ksh
   print ${11}
   shift 10
   print $1

$ param11b.ksh one two three four five six seven eight nine
ten eleven
$ vi param11c.ksh
   (Note: add these two lines to param11b.ksh)
      set A B C
      print $*
$ param11c.ksh a b c d e f g h i j k l m
$ _
```

__ 3. Create the program param11d.ksh to display the name of the program and the number of PPs.

```
$ cp param11c.ksh param11d.ksh

$ vi param11d.ksh (Note: add this line to the end)
      print $0 $#
$ param11d.ksh a b c d e f g h I j k l m
$ _
```

__ 4. List all your environment variables. Now list the ones that are exported and available to a shell script. Set the **CDPATH** to include / and /**home**/**teamXX**. Try to **cd** to "etc" (note there is no "/"). Change your primary prompt to display your command number. Customize your **MAILMSG** variable. Send new mail to yourself and verify. Display the values of **$SECONDS** and **$RANDOM**. Now do it again and explain why they are different. Log out.

```
$ set
$ env (or $ typeset -x)
$ CDPATH=$HOME:/
$ cd etc
$ PS1="! "
$ MAILMSG="Y'all got mail!"
$ mail teamXX < letter
$ mail
$ print $SECONDS $RANDOM
$ print $SECONDS $RANDOM
$ exit
```

__ 5. Edit your .profile. Add a line that sets a variable so your primary prompt reflects your present working directory. (Careful which set of quotes you use!)

```
$ vi $HOME/.profile ( add the line:)
   export PS1='$PWD  =>'
```

__ 6. Test if the export command passes variables down to several levels of subshells. What if you change the value along the way?

```
$ x=5
$ print $x; print $$
$ export x
$ ksh  (or whatever shell you want to open-i.e. bash)
$ print $x; print $$
$ x=7
$ ksh  (or whatever shell you want to open)
$ print $x; print $$
$ exit
$ exit
$ print $x; print $$
```

END OF LAB

Solutions

Variables

4. Sometimes there is a delay in the mail program. The students may get frustrated.

Bash Hints

Variables

- In the bash shell, you must use echo, not print.
- You may want to name your bash scripts .bash instead of .ksh (this is convention only).
- The bash shell uses .bash_profile instead of .profile and .bashrc instead of .kshrc.
- The PS1="!" is not supported in the bash shell.

Exercise 3. Testing

What This Exercise is About

The purpose of the exercise is to gain familiarity with shell testing, signals and traps.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Understand return codes and exit statuses
- Conditionally execute a command, based on the success or failure of a previous command
- Perform file status tests
- Perform string and numeric tests on variables
- Send signals to processes
- Trap a signal sent to a process and perform alternative processing

Introduction

This exercise is designed to enable you to become familiar with various tests, both on files and variables, and for you to understand the results of the tests. It should also enable you to perform some basic conditional processing.

The final part of the exercise should enable you to become familiar with the means of sending various signals to processes and how to trap a signal sent to a shell script, and then execute further processing.

Exercise Instructions

***Remember, the bash shell does not support the* print *command, you must use* echo.**

Exit Status

__ 1. Start a new shell, exit and check the value of **$?**. What do you think it should be? Use the **grep** utility to see if the words “IBM” or “Welcome” are within the file /**etc**/**motd**. (Discussion item: If they are, should they be?)

__ 2. Try a pipeline starting with **cat** /**etc**/**motd** and use **grep**, **wc**, **tee**, and any other utilities you wish to use. Then check to see what the return code (or exit status) was. Which command does this exit code refer to?

Conditional Execution

__ 3. Type in a command line that would give a long list all files in the directory /**tmp**/**workshop** if it exists. Now try it on /**home**/**workshop**.

__ 4. Use grep with " ¦¦ " to print the message '**This is not OS/2!**' if /**etc**/**motd** does not contain the string “**IBM**”.

Test with [ ] and [[ ]]

__ 5. Test for these conditions:

	operator used	result
/etc/hosts.equiv exists	_____	_____
/bin/passwd has SUID set	_____	_____
/etc/hosts is empty	_____	_____
/usr/sbin/getty is a symbolic link	_____	_____
/dev/rfd0 is a block special file	_____	_____

__ 6. Run a series of tests to determine the file type of /**unix**, and what permissions it has.

__ 7. Set the variable **value** to a random number but don't display it. Then run some tests to determine the value of **$value**. (**Hint:** it is somewhere in the range of 0 - 32767.) Try using the “half again” method of dividing the range in half and search either the lower or higher half. Then divide the range into half again,... (**Hint #2:** Don't forget to check the value of **$?**.)

String Expressions

__ 8. Is the file /**etc**/**motd** more recent than /**unix**? Use a test to see if the "**vi**" option is on or not. Check the file descriptors 2 and 3 to see if any are associated with a terminal device. Finally, compare the value of **$OLDPWD** with ".." - are they effectively the same?

Signals

__ 9. Use the **kill** command to get a list of defined signals for your system. Now set a background **sleep** command and try sending it various signals. Write down the effect, if any, and restart the process if it dies.

Traps

__ 10. Set a trap to ignore the "**INT**" signal in your current shell. Run a foreground **sleep** and try to interrupt it by pressing **<CTRL>-c**. Try some other methods to stop the **sleep** command.

__ 11. Start a subshell. Set another trap for "**INT**" that prints a message. Run a foreground sleep and try to interrupt it by pressing **<CTRL>-c** and explain what happens.

__ 12. Exit back to your main login shell and list all of the traps that are set. Reset the "INT" trap. Verify that **<CTRL>-c** works again. Now write a program that contains an "INT" trap, prints a message, then exits. (Hint: use a **sleep 9999**.)

__ 13. Create a script called trap2.ksh that does the following: **find** / **-name 'm*'**; **ls -R** /; **date**; **sleep 5** AND **prints** out a message and sleeps for 3 seconds whenever someone does a **<CTRL>-c**. Execute the script and try doing a **<CTRL>-c** when it is displaying the recursive listing. Did the script exit after printing the message? Why or why not? Try to do the **<CTRL>-c** during the find, date, sleep, and so forth.

__ 14. Change the trap in trap2.ksh by adding the **exit** command to the trap and call this script trap3.ksh. Run the script again and do a **<CTRL>-c.** Do you get different results?

END OF LAB

Exercise Instructions With Hints

Remember, the bash shell does not support the **print** command, you must use **echo**.

Exit Status

__ 1. Start a new shell, exit and check the value of **$?**. What do you think it should be? Use the **grep** utility to see if the words “**IBM**” or “**Welcome**” are within the file /**etc**/**motd**. (Discussion item: If they are, should they be?)

```
$ ksh      (or bash)
$ <CTRL>-d
$ print $?
$ grep IBM /etc/motd
$ grep Welcome /etc/motd
$ _
```

__ 2. Try a pipeline starting with **cat** /**etc**/**motd** and use **grep**, **wc**, **tee**, and any other utilities you wish to use. Then check to see what the return code (or exit status) was. Which command does this exit code refer to? Also, take a look at the new file, **pipe.line**.

```
$ cat /etc/motd ¦ grep AIX ¦ tee pipe.line ¦ wc; print $?
$ cat pipe.line
$ _
```

Conditional Execution

__ 3. Type in a command line that would give a long list all files in the directory /**tmp**/**workshop** if it exists. Now try it on /**home**/**workshop**.

```
$ cd /tmp/workshop && ls -al
$ cd /home/workshop && ls -al
$ _
```

__ 4. Use grep with " ¦¦ " to print the message '**This is not OS/2!**' if /**etc**/**motd** does not contain the string “**IBM**”.

```
$ grep IBM /etc/motd ¦¦ print This is not OS/2!
$ _
```


Test with [ ] and [[ ]]

__ 5. Test for these conditions:

	operator used	result
/etc/hosts.equiv exists	_____	_____
/bin/passwd has SUID set	_____	_____
/etc/hosts is empty	_____	_____
/usr/sbin/getty is a symbolic link	_____	_____
/dev/rfd0 is a block special file	_____	_____

```
$ test -e /etc/hosts.equiv ; print $?      ________
   or $ test -f                            ________
$ test -u /bin/passwd ; print $?           ________
$ test -s /etc/hosts  ; print $?           ________
$ test -L /usr/sbin/getty ; print $?       ________
$ test -b /dev/rfd0  ; print $?            ________
$ _
```

__ 6. Run a series of tests to determine the file type of /**unix**, and what permissions it has.

```
$ [[ -e /unix ]] ; print $?      ________________
$ [[ -f /unix ]] ; print $?      ________________
$ [[ -s /unix ]] ; print $?      ________________
$ [[ -w /unix ]] ; print $?      ________________
$ [[ -r /unix ]] ; print $?      ________________
$ [[ -x /unix ]] ; print $?      ________________
$ [[ -u /unix ]] ; print $?      ________________
$ [[ -g /unix ]] ; print $?      ________________
$ [[ -k /unix ]] ; print $?      ________________
$ [[ -L /unix ]] ; print $?      ________________
$ _
```

__ 7. Set the variable **value** to a random number but don't display it. Then run some tests to determine the value of **$value**. (Hint: it is somewhere in the range of 0 - 32767.) Try using the “half again” method of dividing the range in half and search either the lower or higher half. Then divide the range into half again,... (Hint #2: Don't forget to check the value of **$?**.)

```
$ value=$RANDOM
$ [ $value -lt 16383 ] ; print $?      ________
  (repeat 2nd step as necessary )      ________
                                       ________
                                       ________
                                       ________
```

String Expressions

__ 8. Is the file /**etc**/**motd** more recent than /**unix**? Use a test to see if the "vi" option is on or not. Check the file descriptors 2 and 3 to see if any are associated with a terminal device. Finally, compare the value of **$OLDPWD** with ".." - are they effectively the same?

```
$ [[ /etc/motd -nt /unix ]] ; print $?          _______
$ [[ -o vi ]] ; print $?                        _______
$ [[ -t 2 ]] ; print $?                         _______
$ [[ -t 3 ]] ; print $?                         _______
$ [[ $OLDPWD = ".." ]] ; print $?               _______
```

Signals

__ 9. Use the **kill** command to get a list of defined signals for your system. Now set a background **sleep** command and try sending it various signals. Write down the effect, if any, and restart the process if it dies.

```
$ kill -l       (lowercase L)
$ sleep 99999 &
$ kill -# PID                              _____________
$ kill -# PID                              _____________
$ kill -# PID                              _____________
$ kill -# PID                              _____________
$ kill -# PID                              _____________
$ kill -# PID                              _____________
$ kill -# PID                              _____________
```

Traps

__ 10. Set a trap to ignore the "**INT**" signal in your current shell. Run a foreground **sleep** and try to interrupt it by pressing **<CTRL>-c**. Try some other methods to stop the **sleep** command.

```
$ trap '' INT
$ sleep 99
$ <CTRL>-c
$ _
```

__ 11. Start a subshell. Set another trap for "**INT**" that prints a message. Run a foreground sleep and try to interrupt it by pressing **<CTRL>-c** and explain what happens.

```
$ ksh (or bash or ksh93)
$ trap "echo Hello world" 2
$ sleep 60
$ <CTRL>-c
$ _
```

__ 12. Exit back to your main login shell and list all of the traps that are set. Reset the "**INT**" trap. Verify that **<CTRL>-c** works again. Now write a program that contains an "**INT**" trap, prints a message, then exits. (Hint: use a **sleep 9999**.)

```
$ exit
$ trap
$ trap - INT
$ vi trap.ksh
   trap "print insert message here; exit" INT
   sleep 9999
$ chmod u+x trap.ksh
$ trap.ksh
<CTRL>-c
$ _
```

__ 13. Create a script called trap2.ksh that does the following: **find** / **-name 'm***'; **ls -R** /; **date**; **sleep 5** AND **prints** out a message and sleeps for 3 seconds whenever someone does a **<CTRL>-c**. Execute the script and try doing a **<CTRL>-c** when it is displaying the recursive listing. Did the script exit after printing the message? Why or why not? Try to do the **<CTRL>-c** during the find, date, sleep, and so forth.

```
$ vi trap2.ksh
   trap 'echo "nice try"; sleep 3' 2
   find / -name "m*"
   ls -R /
   date
   sleep 5
$ chmod u+x trap2.ksh
$ trap2.ksh
   <CTRL>-c
```

__ 14. Change the trap in trap2.ksh by adding the exit command to the trap and call this script trap3.ksh. Run the script again and do a **<CTRL>-c**. Do you get different results?

```
$ cp trap2.ksh trap3.ksh
$ vi trap3.ksh
   Change the trap line to read:
   trap 'echo "nice try"; sleep 3; exit' 2
$ trap3.ksh
    <CTRL>-c
```

END OF LAB

Solutions

Exit Status

1."IBM" should not be in the /etc/motd - if it is, oh well. The word 'Welcome' was the basis of a court case in California whereby a hacker got off because the systems "welcomed" him in but the administrator "failed to provide an id and password" so he broke in. It makes you want to remove your welcome mat from your front door! Also, note that if the hacker sees the word welcome in the motd file, it's too late - he's already in!

Test with [ ] and [[ ]]

5.

```
$ test -e /etc/hosts.equiv ; print $?              0
or  -d          or  -f
$ test -u /bin/passwd ; print $?                   0
$ test -s /etc/hosts  ; print $?                   0
$ test -L /usr/sbin/getty ; print $?               1
$ test -b /dev/rfd0  ; print $?                    1
```

6. Run a series of tests to determine the file type of "/unix", and what permissions it has.

```
$ [[ -e  /unix ]] ; print $?                       0
$ [[ -f  /unix ]] ; print $?                       0
$ [[ -s  /unix ]] ; print $?                       0
$ [[ -w  /unix ]] ; print $?                       1
$ [[ -r  /unix ]] ; print $?                       0
$ [[ -x  /unix ]] ; print $?                       0
$ [[ -u  /unix ]] ; print $?                       1
$ [[ -g  /unix ]] ; print $?                       1
$ [[ -k  /unix ]] ; print $?                       1
$ [[ -L  /unix ]] ; print $?                       0
```

6. and 7. These exercises may be easier for the students if they use some vi command line recall and editing features. For instance, instead of <ESC>-k, lllllllllllllR12345 (l = ell), the user, after guessing 16000, can change the guess to 24000 by typing r 16=24.

```
8. $ [[ /etc/motd -nt /unix ]] ; print $?          0
   $ [[ -o vi ]] ; print $?                        0
   (may get 1 if emacs is on in bash instead)
   $ [[ -t 2 ]] ; print $?                         0
   $ [[ -t 3 ]] ; print $?                         1
   $ [[ $OLDPWD = ".." ]] ; print $?               1
```

11. The trap from the parent shell overrides the trap in the child shell.

Exercise 4. Shell Programming Constructs

What This Exercise is About

The purpose of the exercise is to provide an opportunity for the students to review basic testing concepts and practice shell scripting using return codes, signals, and traps.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Use the if - then - else construct for conditional processing
- Write a loop, using either while - do - done or until - do - done constructs
- Write a finite iteration loop using the for - do - done constructs
- Perform processing dependent on specific values using the case construct
- Create simple menus using the select - do - done constructs
- Terminate an iteration of a loop, and start the next using the continue statement
- Terminate an entire loop using the break and/or exit statements

Introduction

This exercise is designed to familiarize you with the various methods of performing conditional processing. It includes the constructs covered in the lecture.

Exercise Instructions

if - then - elif - else - fi

__ 1. Write a script that reports only the nonexistence of the /**etc**/**host** file. Now perform the same task, but from the command line.

__ 2. Write a script that checks the number of positional parameters entered. Report an error if not at least three and include an exit status **1**, otherwise, display the name of the script ($0) and exit normally.

__ 3. Continue your script to include an **else** section that displays the number of positional parameters (after the **if** and the **then** fail). Now add an **elif** branch to test whether **set -o vi** was issued - and if false, exit with a **2**. Finally, add one more **elif** to test the first positional parameter. If the string is not **loop** or **value** you should exit with a code of **3**. Test all three conditions.

until & while Looping

__ 4. Begin a new subshell. Create a file with the name /**tmp**/**$LOGNAME** using the **touch** command. List it with ls.

__ 5. Create a script with a simple **until** loop that keeps running until a /**tmp**/**$LOGNAME** file contains some data. Run it in the background. In effect you are writing a monitor daemon. Run this script in the background and send some data to /**tmp**/**$LOGNAME** to wake the daemon up.

__ 6. Change your **until** script to a **while** loop that tests to make sure you have positional parameters and if so, the loop will print each of the positional parameters in turn and append each to the /**tmp**/**$LOGNAME** file. (Hint: remember “shift” ?)

__ 7. At the command line, enter a one-line, endless loop.

for - in Loops

__ 8. From the command line, create a **for** loop that will print out odd numbers less than ten. (Hint: you may use the actual numbers in the list.)

__ 9. Write a script to do a long listing of only the names of the directories in /.

case Statements

__ 10. Create a script using **case** that checks the first positional parameter to see if it is a filename in the current directory. If so and it ends with **.tmp** remove it. If it begins with an **f**, then copy it to /**tmp**. If it contains the letter **x**, use the **chmod** command to add execution permission. Finally, add a means that if the previous tests all fail, it prints a message saying so. (You will need to create files in your current directory to match the requirements.)

__ 11. Write a script, using **select** and **case** that will allow you to choose your terminal type. Does this permanently change your **TERM** variable? Why or why not?

__ 12. Edit the **select.ksh** script and add the **break** statement. Then display **"TERM is now $TERM"** where appropriate.

__ 13. Write a short script that will read a list of user names from a file called **mail.list** and send a mail message to everyone except **team01**. (You will need to create mail.list and the message file).

END OF LAB

Exercise Instructions With Hints

if - then - elif - else - fi

__ 1. Write a script that reports only the nonexistence of the /**etc**/**host** file. Now perform the same task, but from the command line.

```
$ vi if-then.ksh
   if [[ ! -f /etc/host ]]
   then
      print no host file
   fi

$ chmod u+x if-then.ksh
$ if-then.ksh
$ if [[ ! -f /etc/host ]] ; then print no host file ; fi
$ _
```

__ 2. Write a script that checks the number of positional parameters entered. Report an error if not at least three and include an exit status **1**, otherwise, display the name of the script (**$0**) and exit normally.

```
$ vi if-then-else.ksh
   if [[ $# -lt 3 ]]
   then
      print "Not enough PPs entered"
      exit 1
   else
      print "There are $# PPs entered"
   fi
   print "This script is named $0"

$ chmod u+x if-then-else.ksh
$ if-then-else.ksh
$ if-then-else.ksh a b c d
$ _
```


__ 3. Continue your script to include an **else** section that displays the number of positional parameters (after the **if** and the **then** fail). Now add an **elif** branch to test whether **set -o vi** was issued - and if false, exit with a **2**. Finally, add one more **elif** to test the first positional parameter. If the string is not "**loop**" or "**value**" you should exit with a code of **3**. Test all three conditions.

```
$ cp if-then-else.ksh if-then-elif.ksh
$ vi if-then-elif.ksh
   if [[ $# -lt 3 ]]
   then
      print "Not enough PPs entered"
      exit 1
   elif [[ !-o vi ]]
   then
      print "Your vi option is not on"
      exit 2
   elif [[ $1 != "loop"  &&  $1 != "value" ]]
   then
      print "The first PP is neither 'loop' nor 'value'"
      exit 3
   else
      print "There are $# PPs entered"
   fi
   print "This script is called $0"

$ chmod u+x if-then-elif.ksh
$ if-then-elif.ksh
$ ksh                    (or bash )
$ set +o vi
$ .  if-then-elif.ksh a b c
$ ksh                    (or bash)
$ set -o vi
$ .  if-then-elif.ksh loop b c
$ _
```

until & while Looping

__ 4. Begin a new subshell. Create a file with the name /**tmp**/**$LOGNAME** using the **touch** command. List it with ls.

```
$ ksh                (or bash)
$ touch /tmp/$LOGNAME
$ ls /tmp
$ _
```

__ 5. Create a script with a simple **until** loop that keeps running until a /**tmp**/**$LOGNAME** file contains some data. Run it in the background. In effect you are writing a monitor daemon. Run this script in the background and send some data to /**tmp**/**$LOGNAME** to wake the daemon up.

```
$ vi until.ksh
   until [[ -s /tmp/$LOGNAME ]]
   do
      sleep 10
   done
   print "Got it!"

$ chmod u+x until.ksh
$ until.ksh  &
$ print "Some data" >> /tmp/$LOGNAME
$ _
```

__ 6. Change your **until** script to a **while** loop that tests to make sure you have positional parameters and if so, the loop will print each of the positional parameters in turn and append each to the /**tmp**/**$LOGNAME** file. (Hint: remember “shift” ?)

```
$ vi while.ksh
   Note: change the program to look like this:
      while [[ $# -gt 0 ]]
      do
         print $1
         sleep 5
         shift
      done >> /tmp/$LOGNAME

$ chmod u+x while.ksh
$ while.ksh 1 2
$ cat /tmp/$LOGNAME
$ _
```

__ 7. At the command line, enter a one-line, endless loop.

```
$ while true ; do sleep 10 ; done
$ (use CTRL-c to stop)
```


for - in Loops

__ 8. From the command line, create a **for** loop that will print out odd numbers less than ten. (Hint: you may use the actual numbers in the list.)

```
$ for oddnumber in 1 3 5 7 9 ; do print $oddnumber ; done
$ _
```

__ 9. Write a script to do a long listing of only the names of the directories in /.

```
$ vi for-in.ksh
   for filename in /*
   do
      if [[ -d "$filename" ]]
         then
         ls -ld $filename
      fi
   done

$ chmod u+x for-in.ksh
$ for-in.ksh
$ _
```

case Statements

__ 10. Create a script using **case** that checks the first positional parameter to see if it is a filename in the current directory. If so and it ends with **.tmp** remove it. If it begins with an **f**, then copy it to /**tmp**. If it contains the letter **x**, use the **chmod** command to add execution permission. Finally, add a means that if the previous tests all fail, it prints a message saying so. (You will need to create files in your current directory to match the requirements.)

```
$ touch pop.tmp fpop.cp pxp.ksh pop.file
$ vi case.ksh
   for filename
   do
      case $filename in
         (*.tmp) rm $filename ;;
         (f*)    cp $filename /tmp ;;
         (*x*)   chmod +x $filename ;;
         (*)     print "File name $filename not processed". ;;
      esac
   done
$ chmod u+x case.ksh
$ case.ksh pop.tmp fpop.cp pxp.ksh pop.file
$ ls -l pxp.ksh; ls pop.tmp
```

__ 11. Write a script, using **select** and **case** that will allow you to choose your terminal type. Does this permanently change your **TERM** variable? Why or why not?

```
$ vi select.ksh
   print 'Select your terminal type:'
   PS3='terminal? '
   select term in 'IBM 3151'  'WYSE 60'  'DEC vt220'  'xterm'
   do
      case $REPLY in
      1 ) TERM=ibm3151 ;;
      2 ) TERM=wyse60 ;;
      3 ) TERM=vt220 ;;
      4 ) TERM=xterm ;;
      * ) print 'invalid entry' ;;
      esac
   done

$ chmod u+x select.ksh
$ select.ksh
$ _
```

__ 12. Edit the **select.ksh** script and add the **break** statement. Then display "**TERM is now $TERM**" where appropriate.

```
$ vi select.ksh
   PS3='Select your terminal type'
   select term in    \
      'IBM 3151'   \
      'WYSE 60'    \
      'DEC vt220'  \
      'xterm'
   do
      case $REPLY in
      1 ) TERM=ibm3151 ; break  ;;
      2 ) TERM=wyse60 ; break ;;
      3 ) TERM=vt220 ; break ;;
      4 ) TERM=xterm ; break ;;
      * ) print 'invalid entry' ;;
      esac
   done
   print TERM now is $TERM

$ select.ksh
$ _
```

__ 13. Write a short script that will read a list of user names from a file called **mail.list** and send a mail message to everyone except **team01**. (You will need to create **mail.list** and the message file).

```
$ vi party.ksh
   for name in $(cat  mail.list)
   do
   if [[  "$name" = "team01" ]]
   then
      continue
   else
      mail $name < mail.message
   fi
   done

$ chmod u+x party.ksh
$ party.ksh
$ _
```

END OF LAB

Solutions

The scripts in these exercises are available in /home/workshop.

3.

```
$ set +o vi
$ ksh if-then-elif.ksh a b c
$ set -o vi
$ ksh if-then-elif.ksh loop b c
```

11.No, your TERM variable is not permanently changed because the script ran in a subshell. If you want the variable to change your current environment, you must source the script (run it with the dot command).

OPTIONAL EXERCISE:

1) Write an until loop that notifies you every 10 seconds if team20 is not logged in. When team20 does log in, have the script notify you and end. Run the script in the background and log team20 in from another telnet session.

Hint:

```
$ vi until2.ksh
    until who| grep team20 > /dev/null
    do
        echo "team20 is not logged on"
        sleep 10
    done
    echo "team20 has now logged on"
```

2) The lscfg command is used to display configuration, diagnostic and vital product data information about a system. The data contains information such as part numbers, serial numbers, and engineering change levels from the Customized ODM database. Not all devices support this data. The lsattr command lists out attributes. The lscfg command provides vital data on each device. Create a shell script that performs all 3 commands and produces a report which can be printed.

Hint:

```
$ vi vpdreport.ksh
for DEV in $(lsdev -CF name)
      do
          echo $(lsdev -Cl $DEV -F "name location")
          lsattr -EHl $DEV
      done >> $HOME/d.log
      lscfg -v >> $HOME/d.log
  $ more $HOME/d.log
```


3) Create a script that asks the user if they would like to see customized, predefined, defined, or available devices (use PS3) and then allows them to either enter the menu number or the choice name (i.e. 1 or customized). Use the lsdev command to display the devices.

Hint:

- **vi devices.ksh**

```
PS3="What devices would you like a listing of?"
select DEV in customized predefined defined available quit
do
    case $REPLY in
        customized|1)      lsdev -CH ;;
        predefined|2)      lsdev -PH ;;
        defined|3)         lsdev -CH | grep Defined ;;
        available|4)       lsdev -CH | grep Available ;;
        quit|5)            exit ;;
        *)                 echo "Not a choice" ;;
    esac
done
```

Bash Hints

- The bash shell does not support the print command-be sure to use the echo command in your scripts.
- You may want to name your scripts .bash instead of .ksh - this is convention only.

Exercise 5. Shell Commands and Features

What This Exercise is About

The purpose of the exercise is to provide the students with a basic review of shell commands and practice shell scripting using print, read, eval, and fc.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Understand the options of the print and read commands
- Recognize the function of eval
- Control shell environment with set options

Introduction

This exercise is designed to enable the student to use the Korn shell built-in commands. Use of the read and print statements is very important, as are the options to set.

Exercise Instructions

Remember: bash does not support the **print** command, you must use **echo**.

__ 1. Set the value of the variable **x** to negative 5. Using the **print** command, display the value in two different ways.

__ 2. Write a script that looks like a rolling counter that will increment from 1 to 9.

__ 3. Print a comment to your shell history file. Check your history file to verify your results. (Korn shell only)

__ 4. Set file descriptor 3 to output to the file /**tmp**/**yy** using the **exec** command. Verify your results.

Read

__ 5. Use **exec** to set file descriptor 4 to **read** input from /**etc**/**passwd**. Then **read** a line from **&4**. and **print** it to STDOUT. (Hint: don't forget to use quotes!). Create a **while** loop to **read** and **print** the remaining lines from **-u4**. Finally, **read** and **print** all the lines from /**etc**/**passwd** again by applying re-direction at the end of the **while** loop. (Korn shell only)

GETOPTS

__ 6. Type in the **GETOPTS** example from The Getopts Command visual found in the Shell Commands unit and then run the program with both valid and invalid options. Verify the program works.

FC

__ 7. Set your editor for the **fc** command to /**usr**/**bin**/**vi** - or your favorite editor. Display the last 16 commands on your screen. Create some new command lines using **ls**, **cd**, **echo**, or any other commands you like. These will be used in the next few steps. Finally, display the last 16 again.

__ 8. Now, run the command **ls -al** /**home**/**team01**. Edit and execute that command again but change it from **team01** to **team02** using **fc -e -**, or its alias, **r**. Continue using r a few more times.

__ 9. Now edit and execute some of these command lines. Remember, by using **fc**, the commands will automatically reexecute.

Set

__ 10. List the active options for your shell. Make sure you understand what these are. List the currently exported variables (use **env**) then set the option for "allexport". Assign a value to **var5** and run **env** again. Any difference?

__ 11. Set the following options, then run the commands that follow:

```
$ set -u
$ print $variable_that_has_no_value

$ set -C
$ print Hello World >text
$ print Goodbye Planet >text
$ print Goodbye Planet >| text

$ set -o ignoreeof
$ <Ctrl>-d

$ set -f
$ ls *

$ set -e
$ trap "print ERRtrap" ERR
$ cd /home/nodir
$ exit (or reset by set +u, etc)
```

__ 12. If you have extra time, try some of the extra set commands as seen in Unit 5.

END OF LAB

Exercise Instructions With Hints

Remember: bash does not support the **print** command, you must use **echo**.

__ 1. Set the value of the variable **x** to negative 5. Using the **print** command, display the value in two different ways.

```
$ x=-5 ; print - $x
$ print -  -5
$ _
```

__ 2. Write a script that looks like a rolling counter that will increment from 1 to 9.

```
$ vi rolling.counter.ksh
   for counter in 1 2 3 4 5 6 7 8 9
   do
      print "$counter \r\c"
      sleep 1
   done
$ chmod u+x rolling.counter.ksh
$ rolling.counter.ksh
$ _
```

__ 3. Print a comment to your shell history file. Check your history file to verify your results. (Korn shell only)

```
$ print -s This is a comment
$ more $HOME/.sh_history
   Note: Choose one: cat, more, pg, or tail.
$ _
```

__ 4. Set file descriptor 3 to output to the file /tmp/yy using the **exec** command. Verify your results.

```
$ exec 3> /tmp/yy
$ print Hello world >&3 ; print -u3 Goodbye Planet
$ cat /tmp/yy
$ print -u3 Hello world, again
$ cat /tmp/yy
```

Read

__ 5. Use **exec** to set file descriptor 4 to **read** input from /**etc**/**passwd**. Then **read** a line from **&4**. and **print** it to STDOUT. (Hint: don't forget to use quotes!). Create a **while** loop to **read** and **print** the remaining lines from **-u4**. Finally, **read** and **print** all the lines from /**etc**/**passwd** again by applying re-direction at the end of the **while** loop.

```
$ exec 4< /etc/passwd
$ read -u4
$ print "$REPLY"
$ while read
      > do
      > print "$REPLY"
      > done < /etc/passwd
$ _
```

GETOPTS

__ 6. Type in the GETOPTS example from The Getopts Command visual found in the Shell Commands unit and then run the program with both valid and invalid options. Verify the program works.

```
$ vi getopts.example.ksh
  #!/usr/bin/ksh   (or #!/usr/bin/bash or ksh93)
  # Example of getopts
  USAGE="usage:  getopts.example.ksh [+-c] [+-v] [-a argument]"
  while getopts :a:cv varflag
  do
  case $varflag in
       a)  argument=$OPTARG ;;
       c)  compile=on ;;
      +c)  compile=off;;
       v)  verbose=on ;;
      +v)  verbose=off;;
       :)  print "You forgot an argument for the switch called a." ;;
      \?)  print "$OPTARG is not a valid switch" ; print "$USAGE" ; exit 1 ;;
  esac
  done
  shift $(($OPTIND -1))
  print "compile is $compile; verbose is $verbose; argument is $argument"
  # END
$ chmod u+x getopts.example.ksh
$ getopts.example.ksh -c +v
$ getopts.example.ksh -c +v -a Hello
$ getopts.example.ksh +c -v
$ getopts.example.ksh -cd ROM
$ getopts.example.ksh -cva Hello
$ _
```

(Try more on your own.)

FC

__ 7. Set your editor for the **fc** command to /**usr**/**bin**/**vi** - or your favorite editor. Display the last 16 commands on your screen. Create some new command lines using **ls**, **cd**, **echo**, or any other commands you like. These will be used in the next few steps. Finally, display the last 16 again.

```
$ FCEDIT=vi
$ history
   - or -
$ fc -l -15 -1
$ cd
$ ls -al
$ print $MAILMSG
$ ls /home/teamYY  (Use a directory different than yours.)
$ print Hello world
$ ps -ef ¦ grep teamXX
   Do as many more as you like.
$ history
$ _
```

__ 8. Now, run the command **ls -al** /**home**/**team01**. Edit and execute that command again but change it from **team01** to **team02** using **fc -e -**, or its alias, **r**. Continue using r a few more times.

```
$ ls -al /home/team01
$ fc -e - 1=2
$ r 2=3
$ r 3=\?
$ _
   Do as many more as you like.
```

__ 9. Now edit and execute some of these command lines. Remember, by using **fc**, the commands will automatically reexecute.

Set

__ 10. List the active options for your shell. Make sure you understand what these are. List the currently exported variables (use **env**) then set the option for "allexport". Assign a value to **var5** and run **env** again. Any difference?

```
$ set -o
$ env  | more
$ set -a
$ var5=5
$ env  | more
$ _
```

__ 11. Set the following options, then run the commands that follow:

```
$ set -u
$ print $variable_that_has_no_value

$ set -C
$ print Hello World >text
$ print Goodbye Planet >text
$ print Goodbye Planet >| text

$ set -o ignoreeof
$ <Ctrl>-d

$ set -f
$ ls *

$ set -e
$ trap "print ERRtrap" ERR
$ cd /home/nodir
$ exit (or reset by set +u, etc)
```

__ 12. If you have extra time, try some of the extra set commands as seen in Unit 5.

End of Lab

OPTIONAL EXERCISES:

__ 1. Create a script that prompts a user for filename and if the file is not an ordinary file, the script ends. If the file is an ordinary file, give the user a menu that prompts them to list the permissions, view, modify, or save the file. The menu should also offer the user an “exit”. (remember bash uses echo!)

Hint:

```
print "Enter a file name: \c"
read filename junk
if [ -f $filename ]
then
        :
else
        echo "$filename is not an ordinary file"
        exit 25
fi
PS3="Please enter a number:  "
select var1 in "list permissions" "view" "modify" "save to diskette"
"exit"
do
    case $var1 in
      "list permissions")   if [ -r $filename ]
                                 then echo "$filename has read permission"
                            fi
                            if [ -w $filename ]
                                 then echo "$filename has write permission"
                            fi
                            if [ -x $filename ]
                                 then echo "$filename has execute permission"
                            fi ;;
      view)                 more $filename ;;
      modify)               vi $filename ;;
      "save to diskette")   tar -cvf /dev/fd0 $filename ;;
      exit)                 exit ;;
      *)                    echo "not a choice" ;;
        esac
done
```

__ 2. Create a script that prompts a user to type in a file they wish to find. Use the find command to see if the file is located on the system somewhere.

```
clear
USAGE="locate.file filename"

[ -d  /tmp/locatefile.dir ] || mkdir /tmp/locatefile.dir
```

```
if [ -d /tmp/locatefile.dir ]
then
    :
else
    print "Unable to create a temporary directory under /tmp"
    exit 4
fi

results=/tmp/locatefile.dir/$$

print "\n\n****************************************\n\n"
print "WELCOME TO THE FILE LOCATING SCRIPT\n\n"
print "****************************************\n\n"

print "Enter the file you wish to locate:  \c"
read file1 junk

if [ -z "$junk" ]
then
    if [ -n "$file1" ]
    then
        :
    else
        print "\n\n******************************"
        print "You did not enter a filename."
        print "Correct usage is $USAGE"
        print "******************************\n\n"
        exit 5
fi
else
    print "\n\n************************************"
    print "You entered too many files"
    print "Correct usage is $USAGE"
    print "************************************\n\n"
    exit 6
fi

print "\n\n**********************************"
print "Please be patient, this may take a while."
print "**********************************\n\n"

find / -name $file1 > $results 2> /dev/null
```

```
if [ -s $results ]
then
    print "\n\n****************************************"
    print "Here are the results, use the space bar to browse."
    print "****************************************\n\n"
    sleep 6
    more $results
else
    print "\n\n********************************"
    print "I'm sorry, unable to locate $file1"
    print "********************************\n\n"
fi

rm $results
```

END OF LAB

Solutions

The scripts in these exercises are available in /**home**/**workshop**.

Set

11.

After each of the next steps.	The display shows:
`print variable_that_has_no_value`	`parameter not set`
`print "Hello World" > text`	
`print "Goodbye Planet" > text`	`ksh: text: file already exists`
`print "Goodbye Planet" >| text`	
`<Ctrl>-d`	`you must type "exit"`
`ls *`	`* not found`
`cd /home/nodir`	`ksh:/home/nodir not found`

BASH HINTS:

- Bash does not support the print command, you must use echo.
- When using echo's special characters (such as \r\c in the rolling counter script) use the -e option with echo.
- The -u and the -s option are not supported in the bash shell.
- Remember, bash does not support "+" options in getopts!

Exercise 6. Shell Arithmetic

What This Exercise is About

This exercise is concerned with performing arithmetic calculations, using expr, let, and the bc utility.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Use expr for basic arithmetic and for logic evaluations
- Use let statements to perform arithmetic
- Use the bc utility for both simple arithmetic and complex mathematical evaluations

Introduction

This exercise is intended to give the student some practice with arithmetic techniques and to learn how to perform some logical tests that are not shell based.

Exercise Instructions

Expr.

__ 1. Use **expr** to evaluate the sum "10 - 5 * (4 / 2)". Display your answer and the return code.

__ 2. Evaluate " 10 % 3 - 3 ". Display your answer and the return code.

__ 3. Set some variables with values. Try some calculations with these.

Logical Evaluations and Let Operations

__ 4. Guess the results of the examples below, then perform the evaluations:

evaluate	result	$?
$ expr a = b	_____	_____
$ expr 1 \& 2	_____	_____
$ expr 0 \& 0	_____	_____
$ expr 0 \¦ 2	_____	_____
$ expr a \¦ b	_____	_____

__ 5. Try some simple let operations using the three variables set in the expr exercise.

evaluate	result	$?
$ ((var1 + var2))	_____	_____
$ let "var1 - var1"	_____	_____
$ ((var4=var2+var2))	_____	_____
$ let "var6 = var3 * (var3-var1)" ; print $var6	_____	_____
$ let var7=4+3 var8=var7+1 ; print $var7 $var8	_____	_____

__ 6. Use the following if construct to test some logical lets:

```
$ if  substitute expression here
> then
> print True
> else
> print False
> fi
$ _
```

Use these expressions:	prediction	result
((1 != var1))	_____	_____
let "var1 -= 1|| var1 >= 2"	_____	_____
let "var1==1 && var2==2"	_____	_____
((var1 != 1 ¦¦ var2 == 2))	_____	_____

__ 7. Declare another integer, w, and use it to display the sum of x, y, and z without using a let or (( )) instruction.

Integers and Bases

__ 8. Declare some integers variables, and define some using base 8 and 16. Use the examples:

```
$ integer l=10 m=13
$ typeset -i8 n=8 o=12
$ typeset -i16 p=16 q=10
```

__ 9. Print each variable and explain why your answer is what it is.

__ 10. Finally, convert the value of m into base 16 by assigning it into the base 16 variable declared above. Convert the value of q into base 8 and the value of n into base 10.

bc Utility

__ 11. Try some simple operations with bc:

```
$ print "scale = 3; 10/6" ¦ bc
$ print "scale = 4 ; (10/3) * 3" ¦ bc
$_
```

__ 12. Write a script to calculate pi to a specified, by **$1**, number of decimal places. (Hint: pi is 4 times the arc tangent of 1.) (Better hint: Use **4 * a(1)** in your formula.)

This program will only accept a scale value as high as ____ .

Find the limit using the trial and error method.

END OF LAB

Exercise Instructions With Hints

Expr.

__ 1. Use **expr** to evaluate the sum "10 - 5 * (4 / 2)". Display your answer and the return code.

```
$ expr 10 - 5 \* \( 4 / 2 \)
$ print $?
$_
```

__ 2. Evaluate " 10 % 3 - 3 ". Display your answer and the return code.

```
$ expr 10 % 3 - 3
$ print $?
$_
```

__ 3. Set some variables with values. Try some calculations with these.

```
$ var1=1 ; var2=2 ; var3=3
$ expr $var1 - $var2 + $var3
$ print $?
$_
```

Logical Evaluations and Let Operations

__ 4. Guess the results of the examples below, then perform the evaluations:

evaluate	**result**	**$?**
`$ expr a = b`	_____	_____
`$ expr 1 \& 2`	_____	_____
`$ expr 0 \& 0`	_____	_____
`$ expr 0 \¦ 2`	_____	_____
`$ expr a \¦ b`	_____	_____

```
$ _
```

__ 5. Try some simple let operations using the three variables set in the expr exercise.

evaluate	**result**	**$?**
`$ (( var1 + var2 ))`	_____	_____
`$ let "var1 - var1"`	_____	_____
`$ (( var4=var2+var2 ))`	_____	_____
`$ let "var6 = var3 * (var3-var1)" ; print $var6`	_____	_____
`$ let var7=4+3 var8=var7+1 ; print $var7 $var8`	_____	_____

```
$ _
```

__ 6. Use the following if construct to test some logical lets:

```
$ if  substitute expression here
> then
> print True
> else
> print False
> fi
$ _
```

Use these expressions:	prediction	result
((1 != var1))	_____	_____
let "var1 -= 1||var1 >= 2"	_____	_____
let "var1==1 && var2==2"	_____	_____
((var1 != 1 ¦¦ var2 == 2))	_____	_____

__ 7. Declare another integer, w, and use it to display the sum of x, y, and z without using a let or (( )) instruction.

```
$ x=3 ; y=6 ; z=9
$ integer w
$ w=x+y+z
$ print $w
$_
```

Integers and Bases

__ 8. Declare some integers variables, and define some using base 8 and 16. Use the examples:

```
$ integer l=10 m=13
$ typeset -i8 n=8 o=12
$ typeset -i16 p=16 q=10
$_
```

__ 9. Print each variable and explain why your answer is what it is.

```
$ print $l $m
$ print $n $o
$ print $p $q
$_
```

__ 10. Finally, convert the value of m to base 16 by assigning it into the base 16 variable declared above. Convert the value of q into base 8, and the value of n into base10.

```
$ l=$n ; print $l
$ o=$q ; print $o
$ p=$m ; print $p
$_
```

bc Utility

__ 11. Try some simple operations with bc:

```
$ print "scale = 3; 10/6" ¦ bc
$ print "scale = 4 ; (10/3) * 3" ¦ bc
$_
```

__ 12. Write a script to calculate pi to a specified, by **$1**, number of decimal places. (Hint: pi is 4 times the arc tangent of 1.) (Better hint: Use **4 * a(1)** in your formula.)

```
$ vi picalc.ksh
      print "scale=$1 ; 4*a(1)" ¦ bc -l

$ chmod u+x picalc.ksh
$ picalc.ksh x
$_
```

This program will only accept a scale value as high as ____ .

```
$ picalc.ksh 1
```

Find the limit using the trial and error method.

```
$ picalc.ksh 2
$ picalc.ksh 3
$ picalc.ksh 4
$ picalc.ksh 5
$ picalc.ksh 6
$ picalc.ksh 7
$ _
```

END OF LAB

Solutions

The scripts in these exercises are available in /**home/workshop**.

Expr

```
1. $ expr 10 - 5 \* \( 4 / 2 \)            0
   $ print $?                               1

2. $ expr 10 % 3 - 3 - 2                   -2
   $ print $?                               0

3. $ var1=1 ; var2=2; var3=3
   $ expr $var1 - $var2 + $var3             2
   $ print $?                               0
```

Logical Evaluations and Let Operations

```
4.    evaluate                                        result      $?
$ expr a = b                                          0           1
$ expr 1 \& 2                                         1           0
$ expr 0 \& 0                                         0           1
$ expr 0 \| 2                                         2           0
$ expr a \| b                                         a           0

5.    evaluate                                        result      $?
$ (( var1 + var2 ))                                               0
$ let "var1 - var1"                                               1
$ (( var4=var2+var2 ))                                0
$ let "var6 = var3 * (var3-var1)"    ; print $var6    6           0
$ let var7=4+3 var8=var7+1 ; print $var7 $var8        7,8         0

6. Use these expressions:                             prediction  result
(( 1 != var1 ))                                       F
let "var1 -= 1"||"var1 >= 2"                          F
let "var1==1 && var2==2"                              F
(( var1 != 1 || var2 == 2 ))                          T

7. $ print $w                                18
```

Integers and Bases

```
9. $ print $l $m                              10          13
   $ print $n $o                             8#10        8#14
```

```
   $ print $p $q                          16#10       16#A

10.$ l=$n; print $l                          8
   $ o=$q; print $o                        8#12
   $ p=$m; print $p                        16#d
```

bc Utility

```
11.$ print "scale = 3 ;   10/6" | bc                1.666
   $ print "scale = 4 ; ( 10/3 ) * 3"  | bc          9.9999
```

12.This program will only accept a scale value as high as 98.

BASH HINTS

- The bash shell does not support the print command-you must use the echo command.

Exercise 7. Typeset and Functions

What This Exercise is About

The purpose of this exercise is to familiarize the student with arrays, command substitution, function definition, and usage and the creation of aliases.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Understand how to define and access arrays and array elements
- Perform command substitution
- Define and use functions
- Understand the behavior of variables in functions
- Use traps in functions to control signals
- Create libraries of functions and understand how to load them
- Create new commands using aliases

Introduction

This exercise is intended to give you experience in command substitution, functions and aliases.

Exercise Instructions

Arrays

__ 1. Define an array “colors” to hold red, yellow, green, blue, black, orange, and white. Display the entire array, then print only the colors of your flag.

__ 2. Add purple as element number 8 and display all. Is it as you expected? Change element 5 to brown and display again. Finally, eliminate yellow from the array and display what is left.

__ 3.

Command Substitutions

__ 4. Store the /**etc**/**passwd** file into a variable in two different ways.

__ 5. Print out how many users are logged on in sentence form using command substitution.

__ 6. **Use expr** to calculate the sum "10 + 3 / (4 - 2) ", nesting will be required, and assign it to the variable called **sum**. Then, use a **while** loop to print out the numbers 0 to 9 with the print control "\c" character, command substitute the output so the variable called **string** holds "0123456789". You can use this format:

string=$(_____ ; while _____ ; do _____ ; done)

(Hint: You may want to use set -x to watch what happens.)

__ 7. What is the difference between these two?

```
PS1="$PWD $ "
PS1='$PWD $ '
```

Functions

__ 8. Create a script with a function called add that adds up two numbers passed to it as positional parameters.

__ 9. Make these additions to the script and predict the outcomes after each line:

Add a statement inside the function to print the value of **$0**.

Set a variable, **testvar**, in your main program giving it the value of 1. Print it inside and outside your function.

Following that **testvar** print line, set the variable to 2 inside your function. Print it inside and outside your function.

Insert the command **typeset testvar** in front of the second assignment. What will this do?

__ 10. Still using the same script: (predict the outcomes after each line)

Place **return 2** inside your function definition. Print **$?** upon return from the function call.

Replace **return 2** with **exit 2**. What happens now?

Remove the **exit 2** line. After the first call to your function, add the line:

```
unset -f function (where function is the name)
```

Try calling the function again, and explain what happens.

__ 11. Continuing with the same script: (predict the outcomes after each line)

Set a trap for INT in the main body of your program (before the function call).

Set another trap at the top of your function definition.

Change the trap in the main body to trap "" INT . What happens now?

Typeset and Variables

__ 12. Define this function at the command line level and invoke it to ensure it works:

```
$ function exp
    {
    print "In function"
    }
$ exp
$_
```

__ 13. From the command line, list all exported variables without their values. Then list the **readonly** ones and explain why they might be protected. Finally, list the integer variables with their associated values.

__ 14. Create a new variable with:

```
typeset -i8 a[2]
```

Assign values to both elements, print all array elements, assign a value to element number 4, and reprint all elements. Did this work as expected? (Korn Shell only)

Aliases

__ 15. Define an alias for **ls** that will always list files beginning with "." along with regular files in your **$PWD**. Try it once with double quotes, and once with single quotes. Any difference? Next, clear this new alias and ensure the real **ls** works again.

__ 16. List all the **tracked aliases** for your shell. Set the trackall option then run a number of AIX commands: **ls, cat, cp, touch**, and so forth, then re-examine the “tracked aliases” you have. (For the bash shell use the hash command, see student notebook)

Eval

__ 17. Set four positional parameters to any values. Then display the last argument (Hint: use **$$#**). Is that what you expected? Now run the same command, but place **eval** in front of the first word. Is that better? To verify what happened, set **xtrace** on, and retype the last command.

__ 18. Using **eval** with a **for** loop, set four variables: **var1**, **var2**, **var3**, and **var4** to the numbers 1, 2, 3, and 4. Print out the results and verify.

END OF LAB

Exercise Instructions With Hints

Arrays

__ 1. Define an array “colors” to hold red, yellow, green, blue, black, orange, and white. Display the entire array, then print only the colors of your flag. (Bash users should use ‘**echo**’ instead of ‘**print**’)

```
ksh$ set -A colors red yellow green blue black orange white
bash$ colors=(red yellow green blue black orange white)
$ print ${colors[*]}
   - or -
$ print ${colors[@]}
$ for shade in 0 6 3
   > do
   >  print ${colors[$shade]}
   > done
$ _
```

__ 2. Add purple as element number 8 and display all. Is it as you expected? Change element 5 to brown and display again. Finally, eliminate yellow from the array and display what is left.

```
$ colors[8]=purple
$ print ${colors[*]}
$ colors[5]=brown
$ print ${colors[*]}
$ colors[1]=
$ print ${colors[*]}
$ _
```

Command Substitutions

__ 3. Store the /**etc**/**passwd** file into a variable in two different ways.

```
$ variable=`cat /etc/passwd`
$ print $variable
$ variable=$(< /etc/passwd)
$ print $variable
$ _
```

__ 4. Print out how many users are logged on in sentence form using command substitution.

```
$ print "There are $(who| wc -l) users logged on the system."
```

__ 5. **Use expr** to calculate the sum "10 + 3 / (4 - 2) ", nesting will be required, and assign it to the variable called **sum**. Then, use a **while** loop to print out the numbers 0 to 9 with the print control "\c" character, command substitute the output so the variable called **string** holds "0123456789". You can use this format:

string=$(_____ ; while _____ ; do _____ ; done)

(Hint: You may want to use set -x to watch what happens.)

```
$ sum=$( expr 10 + 3 /  \( 4 - 2 \) ) ; print $sum
$ string=$(s=-1 ; while (( (s+=1) <= 9 )) ; do print "$s\c" ;
done)
$ print $string
$ _
```

__ 6. What is the difference between these two?

```
      PS1="$PWD $ "
      PS1='$PWD $ '
$ PS1="$PWD $ "
   (Change directories many times and watch what happens)
$ PS1='$PWD $ '
   (Change directories many times and watch what happens)
$  _
```

Functions

__ 7. Create a script with a function called add that adds up two numbers passed to it as positional parameters.

```
$ vi add
     function add
     {
        let sum=$1+$2
     }
     add $1 $2
     print $sum
$ chmod u+x add
$ add 3 5
   8

$ _
```

__ 8. Make these additions to the script and predict the outcomes after each line:

Add a statement inside the function to print the value of **$0**.

Set a variable, **testvar**, in your main program giving it the value of 1. Print it inside and outside your function.

Following that **testvar** print line, set the variable to 2 inside your function. Print it inside and outside your function.

Insert the command **typeset testvar** in front of the second assignment. What will this do?

```
$ vi add2  (script should look like this)
      function add2
      {
        let sum=$1+$2
        print function name = $0
        print testvar inside = $testvar
        typeset testvar
        testvar=2
        print testvar inside = $testvar
      }
      testvar=1
      print testvar outside = $testvar
      add2  $1 $2
      print sum outside = $sum
      print testvar outside = $testvar
$ chmod u+x add2
$ add2 3 5
$ _
```

__ 9. Still using the same script: (predict the outcomes after each line)

Place **return 2** inside your function definition. Print **$?** upon return from the function call.

Replace **return 2** with **exit 2**. What happens now?

Remove the **exit 2** line. After the first call to your function, add the line:

```
unset -f function (where function is the name)
```

Try calling the function again, and explain what happens.

```
$ vi add3 (script should look like this)
   function add3
   {
           let sum=$1+$2
           print function name = $0
           print testvar inside = $testvar
           typeset testvar
           testvar=2
```

```
            print testvar inside = $testvar
            return 2 # then exit 2, then remove
      }
      testvar=1
      print testvar outside = $testvar
      add3 $1 $2
      print return code = $?
      print sum outside = $sum
      print testvar outside = $testvar
      unset -f add3   # then remove
      add3 $1 $2   # then remove
$ chmod u+x add3
$ add3 3 5
$ _
```

__ 10. Continuing with the same script: (predict the outcomes after each line)

Set a trap for INT in the main body of your program (before the function call).

Set another trap at the top of your function definition.

Change the trap in the main body to trap "" INT . What happens now?

```
$ vi add4 (script should look like this)
  function add4
  {             # 1) new line here
  trap "print function received INT ; exit" INT
     print function name = $0
     print testvar inside = $testvar
     typeset testvar
     testvar=2
     print testvar inside = $testvar
     let sum=$1+$2
     return 2
  }
  testvar=1
  print testvar outside = $testvar
  trap "print program received INT ; exit" INT
  # trap "" INT
  sleep 9
  add4 $1 $2
  print return code = $?
  print sum outside = $sum
  print testvar outside = $testvar
$ chmod u+x add4
$ add4 3 5
$ _
```

Typeset (or declare) and Variables

__ 11. Define this function at the command line level and invoke it to ensure it works:

```
$ function exp
   {
   print "In function"
   }
$ exp
$_
```

__ 12. From the command line, list all exported variables without their values. Then list the **readonly** ones and explain why they might be protected. Finally, list the integer variables with their associated values. Bash users: use '**declare**'

```
$ typeset +x
$ typeset +r
$ typeset -i
$ _
```

__ 13. Create a new variable with:

```
typeset -i8 a[2]
```

Assign values to both elements, print all array elements, assign a value to element number 4, and reprint all elements. Did this work as expected? (Korn Shell only)

```
$ typeset -i8 a[2]
$ a=1 ; a[1]=2
$ print ${a[@]}
$ a[4]=4 ; print ${a[@]}
$ _
```

Aliases

__ 14. Define an alias for **ls** that will always list files beginning with "." along with regular files in your **$PWD**. Try it once with double quotes, and once with single quotes. Any difference? Next, clear this new alias and ensure the real **ls** works again.

```
$ alias ls="ls -a $PWD"
$ ls
$ cd /
$ ls
$ alias
$ alias ls='ls -a $PWD'
$ ls
$ cd -
$ ls
$ alias
$ unalias ls
```

__ 15. List all the **tracked aliases** for your shell. Set the trackall option then run a number of AIX commands: **ls, cat, cp, touch**, and so forth, then re-examine the "tracked aliases" you have. (For the bash shell use the hash command, see student notebook)

```
$ alias -t
$ set -o trackall
$ ls
$ cat /etc/motd
$ cp /etc/motd my.motd
$ touch your.motd
$ alias -t
$ _
```

Eval

__ 16. Set four positional parameters to any values. Then display the last argument (Hint: use **$$#**). Is that what you expected? Now run the same command, but place **eval** in front of the first word. Is that better? To verify what happened, set **xtrace** on, and retype the last command.

```
$ set a b c d
$ print The last argument is \$$#
$ eval print The last argument is \$$#
$ set -x
$ eval print The last argument is \$$#
   + eval print The last argument is $4
   + print The last argument is d
$ set +x
$ _
```

__ 17. Using **eval** with a **for** loop, set four variables: **var1**, **var2**, **var3**, and **var4** to the numbers 1, 2, 3, and 4. Print out the results and verify.

```
$ vi eval.ksh
   for i in 1 2 3 4
   do
      eval var$i=$i
      eval print var$i = \$var$i
   done

$ chmod u+x eval.ksh
$ eval.ksh
$ _
```

END OF LAB

Optional Exercises

__ 1. Use the system variable **SECONDS** to put the time of day into your prompt. (This file is located in /**home**/**workshop**/**settime.ksh**)

```
$ vi .profile
    export SECONDS="$(date '+3600*%H+60*%M+%S')"
    typeset -Z2 hour minute
    timenow='$(((hour=(SECONDS/3600)%24))).$(((minute=SECONDS/60%60))'
    export PS1="($timenow) > "
$ . .profile
12:34  > _
```

__ 2. Define the array below to hold a sequence of alphabet characters:

```
$ set -A chars z a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Write a function that expects an alphabet character as its only positional parameter. It should work out the position of the character in the array “chars” using a while loop to progress through the array until a match is found. When you have found the array element number of the positional parameter passed to the function, print out the next character in the array. For example, you type **m**, it prints **n**. You need to load the function first by running the script.

```
$ vi bet
  function bet
  {
     integer n=0
     while [[ ${chars[n]} != $1 ]]
     do
          n=n+1
     done
     if [[ n -eq 26 ]]
     then
          print a
     else
          print ${chars[n+1]}
     fi
  }
$ bet m
n
$ _
```

__ 3. Type in the card dealing pickacard.ksh program from Unit 7. Play with it to verify it works, then adapt it so it deals another card by request. Use let for loop control and case to handle the evaluation of card values. Have the program print out the total value of the cards chosen and the name of each of card picked.

Logic hints:

Initialize loop counters, card value total, deck array size (52)
set-up card deck array
while total value < 21 and "yes" to another card
    pick a card - choose a random array element (% deck size)
    print the card chosen
    add its value to the total value so far
    print the total value so far selected
    ask if another card is to be drawn
    put the last card in the deck in place of the chosen card
    reduce the card deck size by 1
repeat the loop

```
$ vi pickacard.ksh
#!/usr/bin/ksh
# This is an example of the complete program
integer number=0 total=0 size=52 element
for suit in CLUBS DIAMONDS HEARTS SPADES
do
  for value in ACE 2 3 4 5 6 7 8 9 10 JACK QUEEN KING
  do
       card[number]="$value of $suit"
       number=number+1
  done
done
another_card=yes
while (( total < 21 )) && [["$another_card" = "yes" ]]
do
  element=RANDOM%size
  choice=${card[element]}
  print $choice
  case $choice in
       (KING*)   total=total+10 ;;
       (QUEEN*)  total=total+10 ;;
       (JACK*)   total=total+10 ;;
       (10*)     total=total+10 ;;
       (9*)      total=total+9 ;;
       (8*)      total=total+8 ;;
       (7*)      total=total+7 ;;
       (6*)      total=total+6 ;;
```

```
        (5*)      total=total+5 ;;
        (4*)      total=total+4 ;;
        (3*)      total=total+3 ;;
        (2*)      total=total+2 ;;
        (ACE*)    total=total+1 ;;
        (*)       print "Do you always cheat?" ;;
  esac
  print That makes $total
  read another_card?"Another card [yes or no]: "
  size=size-1
  card[element]=${card[size]}
done

$ ksh pickacard.ksh
$ _
```

END OF LAB

Solutions

The scripts in these exercises are available in /**home/workshop**.

Command Substitutions

5. Want to see what happened? Type

   ```
   set -x
   ```

   at the prompt and run the command again.

6. What is the difference between these two substitutions?

   $ PS1="$PWD $ "(Change directories many times and watch what happens) WON'T change as you 'cd'

   $ PS1='$PWD $ '(Change directories many times and watch what happens) WILL change as you 'cd'

Typeset and Variables

12. They are protected to keep the system secure from the users!

13. $ print ${a[@]} 1 2 4

BASH HINTS:

- Bash does not support the print command, you must use echo instead.
- Bash does not support the typeset command, you must use declare instead.
- set -A is not supported in the bash shell. Instead you can set up an array on one command line like this “colors=(red yellow green blue black orange white)”

Exercise 8. More Shell Variables

What This Exercise is About

The purpose of the exercise is to familiarize the student with the concepts of handling information contained in strings, specifically those held in variables.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Manipulate character strings
- Understand and use more typeset options

Introduction

This exercise will test the knowledge learned regarding some complex shell syntax for command line expansions as a means of saving execution time.

Exercise Instructions

Variable Replacements

__ 1. Run the **env** command to examine your environment variables with the export attribute. If **TMOUT** is not currently set, use the **${ }** syntax to set it to 60 seconds. Then using the **null** command, how can you produce an error message if the variable "**var**" is not set, but produce no output or side-effects if it is?

__ 2. View /**etc**/**profile** for TERMDEF and explain how it does what it does.

String Manipulations

__ 3. Set a variable to hold an eight character string, such as **var=abcdefgh**. Use the shell ${ % or # } syntax to chop the string into eight one character strings: **var1** to **var8**. (Hint, use your **<ESC>k** or up arrow to lessen the amount of typing.)

__ 4. Use the Korn shell **typeset** or Bash shell **declare** command to re-chop the var string into eight more one character strings, type1 to type8. (Korn Shell only, bash users can try printf if interested)

__ 5. Write a script that asks a user to enter their e-mail address, and then address them by their username only. (That is, user enters mary_smith@us.com, you reply "hello mary_smith!"

Formatting (Korn Shell only)

__ 6. Set the following variables:

num1=000001

num2=000123

num3=012345

Use typeset to clear the zeros and print the numbers.

__ 7. Use typeset with a while loop to print the numbers 1 to 10 in a two-digit format:, 01, 02, 03, ... 10. (Hint: while (((i=i+1)<= 10))

__ 8. Transform an uppercase string into lowercase. What happens if the string used is mixed case?

Challenge

__ 9. In some shells, notably the C Shell, there are functions named pushd and popd that implement a stack of directory names. This allows you to move to another directory temporarily and remember where you were. This is helpful when doing work between two directories. You want to be able to tell the user where they are after the function completes. A stack is a last-in, first-out list. Write two suitable Shell functions.

    Use shell ${# or % } syntax to get the directory name. What do you do if the list is empty?

    Hint: you will need to initialize the stack only on the first use of the function.

What else might you have to do before using either function?

END OF LAB

Exercise Instructions With Hints

Variable Replacements

__ 1. Run the **env** command to examine your environment variables with the export attribute. If **TMOUT** is not currently set, use the **${ }** syntax to set it to 60 seconds. Then using the **null** command, how can you produce an error message if the variable "**var**" is not set, but produce no output or side-effects if it is?

```
$ env
$ unset TMOUT
$ TMOUT=${TMOUT:=60}
$ print $TMOUT
$ print ${var?this is an error message}
$ _
```

__ 2. View /**etc**/**profile** for TERMDEF and explain how it does what it does.

```
$ more /etc/profile
$ _
```

String Manipulations

__ 3. Set a variable to hold an eight character string, such as **var=abcdefgh**. Use the shell ${ % or # } syntax to chop the string into eight one character strings: **var1** to **var8**. (Hint, use your **<ESC>k** or up arrow to lessen the amount of typing!)

```
$ var=abcdefgh
$ var1=${var%???????}; print $var1
$ tmpvar1=${var%??????}; print $tmpvar1
$ tmpvar2=${var%?????}; print $tmpvar2
$ tmpvar3=${var%????}; print $tmpvar3
$ tmpvar4=${var%???}; print $tmpvar4
$ tmpvar5=${var%??}; print $tmpvar5
$ tmpvar6=${var%?}; print $tmpvar6
$ var2=${tmpvar1#?}; print $var2
$ var3=${tmpvar2#??}; print $var3
$ var4=${tmpvar3#???}; print  $var4
$ var5=${tmpvar4#????}; print $var5
$ var6=${tmpvar5#?????}; print $var6
$ var7=${tmpvar6#??????}; print $var7
$ var8=${var#???????}; print $var8
$ print $var $var1 $var2 $var3 $var4 $var5 $var6 $var7 $var8
   - or -
$ integer count=0
$ var=abcdefgh
```

```
$ mask="???????"
$ while (( ( count += 1) < 9 ))
  > do
    >              eval var$count=${var%$mask}
    >               mask=${mask#?}
    >               var=${var#?}
    > done

$ print $var $var1 $var2 $var3 $var4 $var5 $var6 $var7 $var8
$ _
```

__ 4. Use the Korn shell **typeset** or Bash shell **declare** command to re-chop the var string into eight more one character strings, type1 to type8. (Korn Shell only, bash users can try **printf** if interested)

```
$ var=abcdefgh
$ typeset -L1 type1=$var
$ typeset -L2 tmpvar1=$var
$ typeset -L3 tmpvar2=$var
$ typeset -L4 tmpvar3=$var
$ typeset -L5 tmpvar4=$var
$ typeset -L6 tmpvar5=$var
$ typeset -L7 tmpvar6=$var
$ typeset -R1 type2=$tmpvar1 type3=$tmpvar2 type4=$tmpvar3
$ typeset -R1 type5=$tmpvar4 type6=$tmpvar5 type7=$tmpvar6
$ typeset -R1 type8=$var
$ print $type1 $type2 $type3 $type4 $type5 $type6 $type7
    $type8
$ _
  - or -
$ integer count=0
$ var=abcdefgh
$ while (( ( count += 1) < 9 ))
  > do
    >  eval typeset -L1 type$count=$var
    >  var=${var#?}
    > done

$ print $type1 $type2 $type3 $type4 $type5 $type6 $type7
    $type8
$ _
```

__ 5. Write a script that asks a user to enter their e-mail address, and then address them by their username only. (that is, user enters mary_smith@us.com, you reply "hello mary_smith!"

```
$ vi chop.ex
    print "Please enter your e-mail address:\c"
    read emailvar
    usernamevar=${emailvar%@*}
    print "Hello $usernamevar  !!!"
$ chmod u+x chop.ex
$ chop.ex
$ _
```

Formatting -- (Korn Shell only)

__ 6. Set the following variables:

num1=000001
num2=000123
num3=012345

Use typeset to clear the zeros and print the numbers.

```
$ num1=000001 ; num2=000123 ; num3=012345
$ typeset -LZ6 num1 num2 num3
$ print $num1 $num2 $num3
$ _
```

__ 7. Use typeset with a while loop to print the numbers 1 to 10 in a two-digit format:, 01, 02, 03, ... 10. (Hint: while (((i=i+1)<= 10))

```
$ typeset -RZ2 i=0
$ while (( (i=i+1) <= 10 ))
  > do
  > print $i
  > done
$ _
```

__ 8. Transform an uppercase string into lowercase. What happens if the string used is mixed case?

```
$ typeset -l uc_string=UPPER
$ print $uc_string
$ _
```

Challenge

__ 9. In some shells, notably the C Shell, there are functions named pushd and popd that implement a stack of directory names. This allows you to move to another directory temporarily and remember where you were. This is helpful when doing work between two directories. You want to be able to tell the user where they are after the function completes. A stack is a last-in, first-out list. Write two suitable Shell functions.

Use shell ${# or % } syntax to get the directory name. What do you do if the list is empty?

Hint: you will need to initialize the stack only on the first use of the function.

```
function pushd
{
   dname=$1
   cd ${dname:?"Missing directory name?"}
   DIRLIST="$dname ${DIRLIST:-$PWD}"
   print "$DIRLIST"
}
$ _
function popd
{
   DIRLIST=${DIRLIST#* }
   cd ${DIRLIST%% *}
   print "$PWD"
}
$ _
```

What else might you have to do before using either function?

END OF LAB

Solutions

The scripts in these exercises are available in /**home/workshop**.

String Manipulations

3.

```
$var            $type            $chop                abcdefgh
$var1           $type1           $chop1               a
$var2           $type2           $chop2               b
$var3           $type3           $chop3               c
$var4           $type4           $chop4               d
$var5           $type5           $chop5               e
$var6           $type6           $chop6               f
$var7           $type7           $chop7               g
$var8           $type8           $chop8               h
```

Formatting

```
6. $ print $num1 $num2 $num3
   1     123   12345
   $ print $num1 $num2 $num3 | tr -d " "
   112312345


7. 01
   02
   03
   04
   05
   06
   07
   08
   09
   10

8. The reply is:
   upper
   Any lower case characters remain unaffected.
```

Challenge

9. You would need to initialize the DIRLIST variable somehow.

Bash Hints

- The bash shell does not support the print command, you must use echo.
- When using \c with echo, don't forget to use the -e option.
- Bash does not support many of the typeset options; how else can the exercises be accomplished in bash?

Exercise 9. Regular Expressions & Data Selection

What This Exercise is About

The purpose of the exercise is to use some of the more common data selection tools and techniques. The students will have the opportunity to combine the use of regular expressions with grep and use cut, paste, and other utilities.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Use regular expressions
- Use grep commands
- Use cut and paste

Introduction

The use of regular expressions is very helpful for a number of tools. The most obvious use is with grep but we shall see sed and awk, and also use them. The exercise concentrates on use of tr, cut and paste, and other selections and data manipulation tools.

Exercise Instructions

Regular Expressions Using grep

__ 1. Copy the /**home**/**workshop**/**phone.list** file to your home directory, then find the following lines:

People whose surnames start with **J** to **P**:

People whose first names start with the letters **M** to **R**:

People whose numbers don't end with **1, 3**, or **6**:

People whose surnames start with **F** through **P**, and whose phone numbers end with **3** or **6**:

__ 2. List entries which contain a sequence of two to four occurrences of **2, 3**, or **5**. Then list people with six character surnames and three or four character first names.

__ 3. Find people whose first names contain a sequence of two identical characters. Then list the people whose numbers contain a duplicated digit anywhere in the number.

Translating text

__ 4. Use the **tr** utility to convert all text in a file to uppercase.

Cutting and pasting

__ 5. Set the **var** variable to be abcdefgh. Use the **cut** command to chop up the **var** string into 8 separate parts, **chop1** to **chop8**.

__ 6. Cut the group names and members from the /**etc**/**group** file into a new group file in your home directory. PLEASE take care not to corrupt /**etc**/**group**. Use **tr** to separate the words with a space.

/etc/group before:	$HOME/group after:
system:!:0:root	system root
bin:!2:bin	bin bin
sys:!:3:bin,sys	sys bin sys

__ 7. Use cut and paste to change the order of the data in phone.list. You would like to have phone_numberlastname, firstname

END OF LAB

Exercise Instructions With Hints

Regular Expressions Using grep

__ 1. Copy the **/home/workshop/phone.list** file to your home directory, then find the following lines:

People whose surnames start with **J** to **P**:

```
$ cp /home/workshop/phone.list  $HOME/phone.list
$ grep '^[J-P]' $HOME/phone.list
$ _
```

People whose first names start with the letters **M** to **R**:

```
$ grep ', [M-R]' $HOME/phone.list
$ _
```

People whose numbers don't end with **1, 3**, or **6**:

```
$ grep -v '[136]$' $HOME/phone.list
- or -
$ grep '[^136]$' $HOME/phonelist
$ _
```

People whose surnames start with **F** through **P**, and whose phone numbers end with **3** or **6**:

```
$ grep '^[F-P].*[36]$' $HOME/phone.list
$ _
```

__ 2. List entries which contain a sequence of two to four occurrences of **2, 3**, or **5**. Then list people with six character surnames and three or four character first names.

```
$ grep '[235]\{2,4\}' $HOME/phone.list
$ grep '^.\{6\}, .\{3,4\}     ' $HOME/phone.list
$ _
```

__ 3. Find people whose first names contain a sequence of two identical characters. Then list the people whose numbers contain a duplicated digit anywhere in the number.

```
$ grep ', .*\([A-Z,a-z]\)\1' $HOME/phone.list
$ grep '\([0-9]\).*\1' $HOME/phone.list
$ _
```

Translating text

__ 4. Use the **tr** utility to convert all text in a file to upper case.

```
$ tr "[a-z]" "[A-Z]" < phone.list
$ _
```

Cutting and pasting

__ 5. Set the **var** variable to be **abcdefgh**. Use the **cut** command to chop up the **var** string into 8 separate parts, **chop1** to **chop8**.

```
$ var=abcdefgh
$ chop1=$(print $var ¦ cut -c1)
$ chop2=$(print $var ¦ cut -c2)
$ chop3=$(print $var ¦ cut -c3)
$ chop4=$(print $var ¦ cut -c4)
$ chop5=$(print $var ¦ cut -c5)
$ chop6=$(print $var ¦ cut -c6)
$ chop7=$(print $var ¦ cut -c7)
$ chop8=$(print $var ¦ cut -c8)
$ print $chop1 $chop2 $chop3 $chop4 $chop5 $chop6 $chop7
    $chop8
   - or -
$ integer count=0 (bash: declare -i count=0)
$ while (( (count +=1 ) < 9 ))
   > do
   > eval chop$count=$(print abcdefgh ¦ cut -c $count)
   > done
$ _
$ print $chop1 $chop2 $chop3 $chop4 $chop5 $chop6 $chop7
    $chop8
$ _
```

__ 6. Cut the group names and members from the **/etc/group** file into a new group file in your home directory. PLEASE take care not to corrupt **/etc/group**. Use **tr** to separate the words with a space.

/etc/group before:	**$HOME/group after:**
system:!:0:root	**system root**
bin:!2:bin	**bin bin**
sys:!:3:bin,sys	**sys bin sys**

```
$ cat /etc/group (to become familiar with the file)
$ cut -d: -f1,4 /etc/group | tr ":" " " | tr "," " " >
    $HOME/group.
$ cat $HOME/group
$ _
```

__ 7. Use cut and paste to change the order of the data in phone.list. You would like to have phone_numberlastname, firstname

```
$ cut -f2 phone.list | paste -d"\t\n" - phone.list | cut -f1,2
$ _
```

END OF LAB

OPTIONAL EXERCISES

These exercises ask the student to create scripts. Some example hints have been provided; but there are many solutions; try to do it more then one way.

__ 1. Using grep, create a script to check to see if any users are logged in more than once.

```
$ vi usercount
for username in $(who | tr -s " " | cut -f1 -d" ")
do varcount=$(who | grep -c  $username)
   if [[  $varcount -gt 1 ]]
   then
      echo $username is logged on $varcount times >
      $HOME/usercountlog
   fi
done
uniq $HOME/usercountlog
$ chmod u+x usercount ; usercount
$ _
```

__ 2. Cat out /etc/passwd and become familiar with it's format. Create a script that checks for "back doors"- that is, any user beside root that has a userid of 0.

```
$ cat /etc/passwd
$ vi backdoorchk.ex
   echo "the following users have a userid of 0"
   grep "^.*:.*:0:.*:.*:.*:.*" /etc/passwd
$ chmod u+x backdoorchk.ex
$ backdoorchk.ex
$ _
```

__ 3. Create a script that asks a user to enter an ip address. After they enter the ip address, check to make sure it is in the correct format. (1-3 numbers followed by a period, followed by 1-3 numbers, followed by a period, and so forth.)

```
print "enter ip address"
read ip_address
print ${ip_address}
if  print ${ip_address}| grep
'^[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}$' >
/dev/null
then    print "correct format"
else    print "incorrect format"
fi
$ _
```

END OF LAB

Solutions

The scripts and files in these exercises are available in /**home**/**workshop**.

BASH HINTS

- The bash shell does not support the print command. Use echo instead.

Exercise 10. The sed utility

What This Exercise is About

The purpose of the exercise is to use some of the more popular sed commands and to use other tools.

What You Should Be Able to Do

At the end of the lab, you should be able to use sed commands.

Introduction

The sed utility is designed to handle data files of simple text.

Exercise Instructions

sed

Substitutions

__ 1. Write a command that removes the first two characters of each phone number and replaces it with **X**.

__ 2. Remove the first names from **phone.list** and store the changed list in another file.

__ 3. Substitute the numbers in front of the dash with **555.**

__ 4. Replace each surname by the first initial and a period.

__ 5. Using a multi-line command, replace the numbers in front of the dash with **555** for phone numbers starting from **1-4** and **666** for numbers starting from **6-9**.

Delete and Print

__ 6. Delete all entries where the surname begins with **L**.

__ 7. What is the difference between these two commands?

```
sed p $HOME/phone.list
sed -n p $HOME/phone.list
```

__ 8. Output the first four lines of the phone.list file.

__ 9. Output lines that contain the numbers **4** to **6**.

Append, Insert and Change

__ 10. Insert 2 blank lines at the end of the phone.list file.

__ 11. Print a title line before the normal sed output:

"Name:<tab><tab><tab>Phone Number:"

__ 12. Change the third and fourth lines of the file into "Line Deleted".

sed with grep

__ 13. There is a command found on systems using the MKS tools that may be useful. It is called gres. It does a find and replace of your RE string as in

```
gres "[ABC]" "xx" example
```

Here [ABC] is the pattern to match and "xx" is the replacement string. Construct a version of this program using the Korn Shell and sed. You can assume for a first version that the input file is supplied as an argument.

END OF LAB

Exercise Instructions With Hints

sed

Substitutions

__ 1. Write a command that removes the first two characters of each phone number and replaces it with **X**.

```
$ sed 's/[0-9]\{2\}/X/' phone.list
```

__ 2. Remove the first names from **phone.list** and store the changed list in another file.

Note: Be careful of whitespace - is it tabs or spaces?

```
$ sed 's/, .*     /   /' $HOME/phone.list > new.phone.list
```

__ 3. Substitute the numbers in front of the dash with **555**.

```
$ sed 's/...-/555-/' $HOME/phone.list
   - or -
$ sed 's/[0-9]\{3\}-/555-/' phone.list
```

__ 4. Replace each surname by the first initial and a period.

```
$ sed 's/^\(.\).*,  \(.*\)/\1. \2/' $HOME/phone.list
```

__ 5. Using a multi-line command, replace the numbers in front of the dash with **555** for phone numbers starting from **1-4** and **666** or numbers starting from **6-9**.

```
$ sed 's/[1-4]..-/555-/
   > s/[6-9]..-/666-/' $HOME/phone.list
$ _
```

Delete and Print

__ 6. Delete all entries where the surname begins with **L**.

```
$ sed '/^L/d' $HOME/phone.list
$ _
```

__ 7. What is the difference between these two commands?

```
$ sed p $HOME/phone.list
$ sed -n p $HOME/phone.list
$ _
```

__ 8. Output the first four lines of the phone.list file.

```
$ sed -n '1,4p' $HOME/phone.list
$ _
```

__ 9. Output lines that contain the numbers **4** to **6**.

```
$ sed -n '/.*[4-6].*/p' $HOME/phone.list
$ _
```

Append, Insert and Change

__ 10. Insert 2 blank lines at the end of the phone.list file.

```
$ sed '$a\
   > \
   > ' $HOME/phone.list
$ _
```

__ 11. Print a title line before the normal sed output:

"Name:<tab><tab><tab>Phone Number:"

```
$ sed '1i\
   > Name:          Phone Number:    ' $HOME/phone.list
$ _
```

__ 12. Change the third and fourth lines of the file into "Line Deleted".

```
$ sed '3,4c\
   > Line Deleted\
   > Line Deleted' $HOME/phone.list
$ _
```

sed with grep

__ 13. There is a command found on systems using the MKS tools that may be useful. It is called **gres**. It does a find and replace of your RE string as in

```
gres "[ABC]" "xx" example
```

Here [ABC] is the pattern to match and "xx" is the replacement string. Construct a version of this program using the Korn shell and sed. You can assume for a first version that the input file is supplied as an argument.

```
$ cat ./gres
#!/bin/ksh
if [[ $# < 3 ]]
then   print Usage: gres pattern replacement file
       exit 1
fi
patt=$1
repl=$2
if [ -f $3 ]
then file=$3
else echo $3 is not a file
   exit 2
fi
SEP="$(print | tr '\012' '\001' )"
sed -e "s$SEP$patt$SEP$repl$SEP" $file
```

END OF LAB

Solutions

The scripts in these exercises are available in /**home**/**workshop**.

13. You should find that the lines are reversed. This has been done by using the Hold Space and deleting lines from the pattern space.

BASH HINTS

- The bash shell does not support the print command. Use echo instead.

Exercise 11. Using awk

What This Exercise is About

The purpose of the exercise is to learn and use awk.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Understand the basic parts of an awk program
- Use awk to manipulate and output data

Introduction

awk is a program that operates on data. It is often used as a report generator or as a means of reformatting data based on input patterns.

Exercise Instructions

awk

Basics

__ 1. Print the Lastname, Math, and English scores for all the students in the /**home**/**workshop**/**results** file. For safety's sake, make a copy of the file in your home directory first.

__ 2. Now use the **BEGIN { getline }** syntax and print the same information. Note any differences.

__ 3. Next, swap the Math and English scores, but keep the headings the same.

__ 4. Create a program structure that calculates the total number of lines and words in the **phone.list** file and also displays the average number of words per line. However there is a coding difference depending on whether you use the **BEGIN** statement or not. Make the program work correctly twice, once with and once without **BEGIN**.

Functions and Files

__ 5. Write an awk script prefix.awk which takes the phone number of the **results** file and prefixes any number that has the first digit less than 4 with **555** and any other number with **666**.

__ 6. Write an awk script which changes any occurrence of the school Georgia and replaces it with Tech. Also, display how many replacements were made.

__ 7. Next, copy your **prefix.awk** program to a file called **prefix2.awk**. Edit the new file so you can use it as a command file with the **-f** option to awk.

for - while - if

__ 8. Create a script with a **for** loop that will print out the frequencies of lines having no words, one word, two words, etc., up to a maximum of ten words.

__ 9. Now make a copy of the last script and use a **while** loop instead of the for loop.

Arrays

__ 10. Create a script using arrays which lists the lastnames of the **results** file next to the Science scores - in reverse order. Then print the sum of the Science scores.

END OF LAB

Exercise Instructions With Hints

awk

Basics

__ 1. Print the Lastname, Math, and English scores for all the students in the **/home/workshop/results** file. For safety's sake, make a copy of the file in your home directory first.

```
$ cp /home/workshop/results $HOME/results
$ awk '{ print $2, $5, $6 }' $HOME/results
$ _
```

__ 2. Now use the **BEGIN { getline }** syntax and print the same information. Note any differences.

```
$ awk 'BEGIN { getline }
   > { print $2, $5, $6 }'  $HOME/results
$ _
```

__ 3. Next, swap the Math and English scores, but keep the headings the same.

```
$ awk 'BEGIN { print "LastName", "Math", "English" ; getline }
   > { print $2, $6, $5 }'  $HOME/results
$ _
```

__ 4. Create a program structure that calculates the total number of lines and words in the **phone.list** file and also displays the average number of words per line. However there is a coding difference depending on whether you use the **BEGIN** statement or not. Make the program work correctly twice, once with and once without **BEGIN**.

```
$ awk ' BEGIN { print "Lines\tWords\tAverage So Far" }
   { wcount += NF }
   END { print NR, "\t", wcount, "\t", wcount / NR }'
   $HOME/phone.list
$ awk '{ wcount += NF }
   END   { print "Lines\tWords\tAverage"
   print NR, "\t", wcount, "\t", wcount / NR }'
   $HOME/phone.list
$ _
```

Functions and Files

__ 5. Write an awk script called prefix.awk which takes the phone number of the **results** file and prefixes any number that has the first digit less than 4 with **555** and any other number with **666**.

```
$ vi prefix.awk
   awk 'BEGIN { getline }
      { x=substr($4,1,1)
        if (x<4) {
        print $2, "555-"$4
            }
        else {
        print $2, "666-"$4
            }
        }' $HOME/results
$ prefix.awk
$ _
```

__ 6. Write an awk script which changes any occurrence of the school Georgia and replaces it with Tech. Also, display how many replacements were made.

```
$ awk '{ print gsub("Georgia","Tech"), "\t", $0 }'
    $HOME/results
$ _
```

__ 7. Next, copy your **prefix.awk** program to a file called **prefix2.awk**. Edit the new file so you can use it as a command file with the **-f** option to awk.

```
$ cp prefix.awk prefix2.awk
$ vi prefix2.awk
   BEGIN {getline}
      { x = substr($4,1,1)
        if (x<4) {
        print $2, "555-"$4
            }
        else {
        print $2, "666-"$4
            }
        }
$ _
$ awk -f prefix2.awk $HOME/results
$ _
```

for - while - if

__ 8. Create a script with a **for** loop that will print out the frequencies of lines having no words, one word, two words, etc., up to a maximum of ten words.

```
$ vi for.awk
   awk  ' { len[NF]++ }
      END { for( i=0 ; i <= NR+1 ; i++ ) {
      if (len[i]){
      print len[i], " lines with ", i, " words"
             }
         }
      } '  $HOME/results
$ for.awk
$ _
```

__ 9. Now make a copy of the last script and use a **while** loop instead of the for loop.

```
$ cp for.awk while.awk
$ vi while.awk
   awk '{ len[NF]++ }
      END { i = 0
      while( i <= NR+1 )      {
      if (len[i])           {
      print len[i], " lines with ", i, " words"
                }
             i++
         }
      } '  $HOME/results
$ while.awk
$ _
```

Arrays

__ 10. Create a script using arrays which lists the lastnames of the **results** file next to the Science scores - in reverse order. Then print the sum of the Science scores.

```
$ vi array.awk
   awk '{ x[NR,1] = $7; x[1,NR] = $2; sum += $7 }
      END   {
      for (i = NR; i > 1; --i)
         { name = NR - i + 2
             print x[1,i], x[i,1] }
             print "Sum of column 7 = "sum
      }' $HOME/results
$ array.awk
$ _
```

END OF LAB

OPTIONAL EXERCISES:

__ 1. Create a script that uses sed and awk to check the output of the df command for the percentage of space used in each filesystem. If the percentage is more then 80% used, send a note to root. One example is shown below, however there are several ways to do this.

```
#!/usr/bin/ksh
star()
{
print "**************************************************\a"
}
MAX=80
df -k | awk '{ print $4, $7 }' |
while read USED FILESYS
do
     if [[ $USED != '%Used' ]]
     then
          USED=`print ${USED} | sed s/%//`
          if [[ ${USED} -ge $MAX ]]
          then
               star
               print "Filesystem ${FILESYS} needs checking!!"
               star
          fi
     fi
done
```

END OF LAB

Solutions

The scripts in these exercises are available in /**home**/**workshop**.