

Shell Programming for System Administrators

Student Guide

D61776GC21

Edition 2.1

April 2010

D66885

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

About This Course	Preface-i
Course Goal	Preface-i
Course Overview	Preface-ii
Course Map	Preface-iii
Module-by-Module Overview	Preface-iv
Course Objectives	Preface-vii
Topics Not Covered	Preface-viii
How Prepared Are You?	Preface-ix
Introductions	Preface-x
How to Use Course Materials	Preface-xi
Conventions	Preface-xii
Icons	Preface-xii
Typographical Conventions	Preface-xiii
UNIX® Shells and Shell Scripts	1-1
Objectives	1-1
What Is a Shell?	1-2
What Are Shell Functions?	1-3
Command-Line Interpreter	1-3
Programming Language	1-3
User Environment	1-4
Solaris OS Shells	1-5
The Bourne Shell	1-5
The C Shell	1-5
The Korn Shell	1-6
Additional Solaris OS Shells	1-7
The GNU Bourne-Again Shell	1-7
The Desktop Korn Shell	1-7
The Job Control Shell	1-8
The Restricted Shell Command Interpreter	1-8
The Enhanced C Shell	1-9
Z Shell	1-9

Shells Discussed in This Course	1-10
Subshells – Child Processes.....	1-11
What Is a Shell Script?.....	1-13
Developing a Script	1-14
Programming Terminology.....	1-15
Logic-Flow Design	1-16
The echoscript1.sh Example Bourne Script.....	1-17
The echoscript2.ksh Example Korn Script	1-18
The /etc/init.d/volmgt Example Boot Script	1-19
Exercise: Reviewing UNIX Shells and Shell Scripts	1-20
Preparation	1-20
Tasks	1-20
Exercise Summary	1-22
Task Solutions.....	1-23
Writing and Debugging Scripts	2-1
Objectives	2-1
Creating Shell Scripts	2-2
Executing a Shell Script.....	2-3
Executing the firstscript.sh Script	2-4
Starting a Script With the #! Characters	2-6
Putting Comments in a Script	2-7
Adding the Debugging Statement.....	2-9
Debug Mode Controls.....	2-10
Example: Debug Mode Specified on the #! Line	2-11
Example: Debug Mode With the set -x Option	2-13
Example: Debug Mode With the set -v Option	2-14
Syntax Comparison for Bourne and Korn Shell Options.....	2-15
Example: Debug Mode With the set -o noglob Option.....	2-16
Exercise: Writing Shell Scripts.....	2-17
Preparation	2-17
Tasks	2-17
Exercise Summary	2-20
Task Solutions.....	2-21
The Shell Environment.....	3-1
Objectives	3-1
User Startup Scripts	3-2
The /etc/profile Script	3-2
The \$HOME/.profile Script.....	3-2
The \$HOME/.kshrc Script	3-3
Modifying a Configuration File	3-4
Shell Variables	3-5
Creating Variables in the Shell	3-7
Exporting Variables to Subshells.....	3-9
Reserved Variables	3-11
Special Shell Variables	3-12

Process Identification.....	3-12
Exit Status.....	3-12
Background Process Identification	3-13
Quoting Characters	3-14
A Pair of Single Quotes	3-14
A Pair of Double Quotes.....	3-15
Backslash	3-15
The eval Command.....	3-16
Shell Command Substitution	3-17
Korn Shell Tilde Expansion.....	3-19
Examples.....	3-19
Arithmetic Operations on Bourne Shell Variables	3-20
Arithmetic Operations on Korn Shell Variables.....	3-23
Arithmetic Precedence	3-24
The Korn Shell let Statement	3-27
Script Math	3-28
Korn Shell Aliases	3-29
Removing Aliases	3-29
Alias Inheritance	3-29
Automatic (Tracked) Aliases	3-30
Built-in Aliases	3-31
Shell Parse Order	3-32
Special Characters Precedence	3-33
Exercise: Using the Shell Environment.....	3-34
Preparation	3-34
Tasks	3-34
Exercise Summary	3-37
Task Solutions.....	3-38
Regular Expressions and the grep Command	4-1
Objectives	4-1
The grep Command.....	4-2
The grep Options.....	4-3
Regular Expression Metacharacters.....	4-5
Regular Expressions	4-6
Escaping a Regular Expression	4-7
Line Anchors.....	4-9
Word Anchors.....	4-10
Character Classes.....	4-11
Character Match.....	4-12
Closure (*)	4-13
The egrep Command.....	4-14
Exercise: Using Regular Expressions and the grep Command	4-15
Preparation	4-15
Tasks	4-15
Exercise Summary	4-17
Task Solutions.....	4-18

Using the sed Editor	5-1
Objectives	5-1
Introduction to the sed Editor	5-2
Command Format	5-3
Editing Commands	5-4
Addressing	5-5
Using sed to Print Text	5-6
Using sed to Substitute Text	5-8
Reading From a File for New Text	5-10
Using sed to Delete Text	5-11
Reading sed Commands From a File	5-12
Using sed to Write Output Files	5-13
Exercise: Using the sed Editor	5-14
Preparation	5-14
Tasks	5-14
Exercise Summary	5-16
Task Solutions.....	5-17
The nawk Programming Language	6-1
Objectives	6-1
Introduction to the nawk Programming Language	6-2
Format of the nawk Command	6-3
Executing nawk Scripts	6-3
Using nawk to Print Selected Fields	6-4
Formatting With print Statements	6-5
Using Regular Expressions	6-7
The BEGIN and END Special Patterns	6-9
Using nawk Scripts	6-11
Using Built-in Variables	6-12
Working With Variables	6-13
Input Field Separator	6-13
Output Field Separator	6-15
Number of Records	6-16
Exercise: Using nawk and Regular Expressions	6-17
Preparation	6-17
Tasks	6-17
Exercise Solution	6-18
User-Defined Variables	6-19
Adding a Field of Numbers	6-19
Variable Examples	6-21
Writing Output to Files	6-23
The printf () Statement	6-24
Exercise: Using nawk to Create a Report	6-26
Tasks	6-26
Exercise: Using nawk Scripts to process Text Files	6-27
Preparation	6-27

Tasks	6-27
Exercise Summary	6-29
Task Solutions.....	6-30
Conditionals.....	7-1
Objectives	7-1
The <i>if</i> Statement.....	7-2
Parts of the <i>if</i> Statement.....	7-4
Command.....	7-4
Block of Statements.....	7-4
End of the <i>if</i> Statement	7-4
Exit Status	7-5
Numeric and String Comparison	7-6
Syntax for <i>if/then/else</i> Statements	7-10
Syntax for <i>if/then/elif/else</i> Statements	7-11
Positional Parameters.....	7-14
Using <i>if</i> to Check Command-Line Arguments	7-15
Creating the USAGE Message.....	7-16
Using <i>if</i> to Check Leap Years.....	7-18
Nested <i>if</i> Statements	7-19
Testing File Objects	7-21
Boolean AND, OR, and NOT Operators	7-23
The <i>case</i> Statement	7-26
Example of Using the <i>case</i> Statement.....	7-27
Replacing Complex <i>if</i> Statements With a <i>case</i> Statement	7-29
The <i>exit</i> Statement	7-31
Exercise: Using Conditionals.....	7-33
Preparation	7-33
Tasks	7-33
Exercise Summary	7-36
Task Solutions.....	7-37
Interactive Scripts	8-1
Objectives	8-1
Input and Output in a Script.....	8-2
The Korn Shell <i>print</i> Statement.....	8-3
Examples of Using the <i>print</i> Statement.....	8-4
Examples of Using the <i>echo</i> Statement	8-5
The <i>read</i> Statement	8-6
Examples of Using the <i>read</i> Statement	8-7
Capturing a Command Result.....	8-8
Printing a Prompt.....	8-9
Prompting for Input – Korn Shell Shortcut	8-11
File Input and Output.....	8-13
User-Defined File Descriptors	8-14
File Descriptors in the Bourne Shell.....	8-15
Korn Shell File Descriptors	8-18

The “here” Document	8-20
Exercise: Using Interactive Scripts	8-22
Preparation	8-22
Tasks	8-23
Exercise Summary	8-25
Task Solutions.....	8-26
Loops	9-1
Objectives	9-1
Shell Loops	9-2
The <code>for</code> Loop Syntax	9-3
The <code>for</code> Loop Argument List.....	9-4
Using an Explicit List to Specify Arguments	9-4
Using Variable Contents to Specify Arguments.....	9-5
Using Command-Line Arguments to Specify Arguments.....	9-6
Using Command Substitution to Specify Arguments.....	9-6
Using File Names in Command Substitution to Specify Arguments	
9-9	
Using File-Name Substitution to Specify Arguments	9-10
Exercise: Using <code>for</code> Loops	9-12
Preparation	9-12
Tasks	9-12
Task Solutions.....	9-13
The <code>while</code> Loop.....	9-15
The <code>while</code> Loop Syntax	9-15
Example of Using a <code>while</code> Loop	9-17
Keyboard Input	9-19
Redirecting Input for a <code>while</code> Loop	9-20
The <code>until</code> Loop.....	9-21
The <code>break</code> Statement	9-22
The <code>continue</code> Statement.....	9-23
Example of Using the <code>continue</code> Statement.....	9-23
The Korn Shell <code>select</code> Loop	9-26
The PS3 Reserved Variable	9-26
Example of Using a <code>select</code> Loop	9-27
Exiting the <code>select</code> Loop	9-29
Submenus	9-31
Example of Using Submenus.....	9-31
The <code>for</code> and <code>select</code> Statements Revisited	9-35
The <code>shift</code> Statement	9-36
Example of Using the <code>shift</code> Statement	9-37
Exercise: Using Loops and Menus	9-39
Preparation	9-39
Tasks	9-39
Exercise Summary	9-41
Task Solutions.....	9-42

The getopt Statement.....	10-1
Objective	10-1
Processing Script Options With the getopt Statement.....	10-2
Using the getopt Statement.....	10-3
Handling Invalid Options.....	10-4
Specifying Arguments to Options.....	10-7
Example of Using the getopt Statement	10-8
Forgetting an Argument to an Option.....	10-10
Exercise: Using the getopt Statement.....	10-13
Preparation.....	10-13
Tasks	10-13
Exercise Summary	10-14
Task Solutions.....	10-15
Advanced Variables, Parameters, and Argument Lists	11-1
Objectives	11-1
Variable Types	11-2
Assessing Variable Values.....	11-2
The Korn Shell typeset Statement	11-3
Example of Using String Manipulations	11-3
Declaring an Integer Variable.....	11-6
Creating Bourne Shell Constants	11-7
Creating Korn Shell Constants	11-8
Removing Portions of a String.....	11-9
Examples of Removing Portions of a String	11-9
Korn Shell Arrays	11-11
Examples of Using Arrays	11-12
Using the shift Statement With Positional Parameters	11-14
The Values of the "\$@" and "\$*" Positional Parameters.....	11-17
Exercise: Using Advanced Variables, Parameters, and Argument Lists	11-25
Preparation.....	11-22
Tasks	11-22
Exercise Summary	11-24
Exercise Solutions.....	11-25
Functions	12-1
Objectives	12-1
Functions in the Shell	12-2
Syntax	12-2
Function Execution	12-2
Positional Parameters and Functions	12-3
Return Values	12-5
The typeset and unset Statements.....	12-6
Function Files	12-7
Autoloading Korn Shell Functions With the FPATH Variable.....	12-8
Exercise: Using Functions	12-10

Preparation	12-10
Tasks	12-10
Exercise Summary	12-12
Task Solutions.....	12-13
Traps	13-1
Objectives	13-1
Shell Signal Values	13-2
Catching Signals With the <code>trap</code> Statement.....	13-5
Example of Using the <code>trap</code> Statement.....	13-6
Catching User Errors With the <code>trap</code> Statement.....	13-9
Example of Using the <code>trap</code> Statement With the ERR Signal	13-11
When to Declare a <code>trap</code> Statement.....	13-13
Exercise: Using Traps.....	13-15
Preparation	13-15
Tasks	13-15
Exercise Summary	13-16
Task Solutions.....	13-17
Advanced <code>nawk</code> Programming	A-1
Programming Concepts.....	A-2
The <code>if</code> Statement.....	A-3
Conditional Printing With the <code>nawk</code> Language	A-4
String Comparisons and Relational and Logical Operators.....	A-5
Logical Operators	A-6
The <code>while</code> Loop in the <code>nawk</code> Language	A-7
The <code>do while</code> Loop	A-7
The <code>for</code> Loop in the <code>nawk</code> Language.....	A-9
Using Loops With Arrays	A-10
Nonnumeric Array Indices.....	A-11
The <code>break</code> and <code>continue</code> Statements	A-13
The <code>continue</code> Statement.....	A-13
The <code>next</code> and <code>exit</code> Statements	A-15
User-Defined Functions	A-16
Command-Line Arguments	A-18
Using Built-in Variables	A-19
Built-in Arithmetic Functions.....	A-20
Built-in String Functions	A-21
Built-in I/O Processing Functions.....	A-23
The <code>printf()</code> Statement.....	A-24
Built-in Operators	A-25
Additional <code>grep</code> Functionality	B-1
Regular Expression Metacharacters.....	B-2
Character Classes	B-3
Tagged Regular Expressions	B-5

Additional sed Functionality.....	C-1
Editing Commands	C-2
Placing Multiple Edits in a Single sed Command	C-2
Using sed to Append, Insert, or Change Text.....	C-3
Shell Metacharacters for Pattern Matching for File Names.....	D-1
Metacharacter Examples.....	D-3
UNIX Commands and Utilities	E-1
Status Commands	E-2
The date Command.....	E-2
The ps Command	E-2
The who Command	E-4
The rusers Command	E-4
The finger Command	E-5
The uptime Command	E-6
The rup Command.....	E-6
The w Command	E-6
File Access Commands.....	E-8
The find Command.....	E-8
The sort Command	E-9
Reading Part of a File	E-12
The tr Command	E-12
The cut Command.....	E-13
The paste Command.....	E-15
The cmp, diff, and sdiff Commands.....	E-16
Combining Commands	E-17
Pipes.....	E-17
Redirection.....	E-17
Shell Comparison	F-1
Shell Comparison.....	F-2

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Preface

About This Course

Course Goal

The goal of the *Shell Programming for System Administrators* course is to provide you with the knowledge and skills necessary for shell script programming. In this course, you learn shell script syntax, as well as analysis and design procedures, to successfully create shell scripts. The course takes prerequisite knowledge and allows you to automate tasks that you frequently perform on the job.

Upon completion of this course, course should be able to:

- Describe the role of shells in the UNIX environment
- Describe the standard shells
- Write shell scripts
- Use and describe regular expressions
- Use the `grep` command to find patterns in a file
- Use regular expression characters with the `grep` command
- Use the `sed` editor to perform noninteractive editing tasks
- Use regular expression characters with the `sed` command
- Use `nawk` commands from the command line
- Write simple `nawk` programs
- Use conditionals in shell programs
- Create user-defined functions in shell scripts
- Write interactive scripts using the `print`, `echo`, and `read` statements

Course Overview

Course Overview

The course first briefly describes the different shells available in the operating system. Then the course covers the basics in analyzing a problem, designing a series of steps to solve the problem, and then organizing a series of operating system commands and shell built-in commands to perform the necessary steps to solve the problem. Good programming techniques and script debugging are also described.

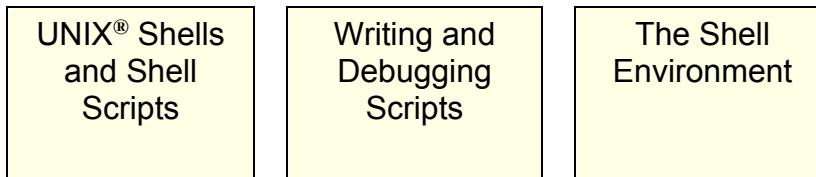
The audience for this course is systems administrators who are familiar with basic administrative concepts and tasks, such as adding users and software and, using the system boot process, daemons, and device configurations. This is a follow-up course to SA-200: *System Administration for the Solaris 10 Operating System, Part 1*.

The course material shows example syntax of both the Bourne and Korn shell, and the module examples and exercise solutions were executed in the Solaris™ 10 OS, Update 8. Most examples and solutions work in either the Bourne or Korn shell, although some topics are denoted as Korn shell only.

Course Map

The following course map enables you to see what you have accomplished and where you are going in reference to the course goals

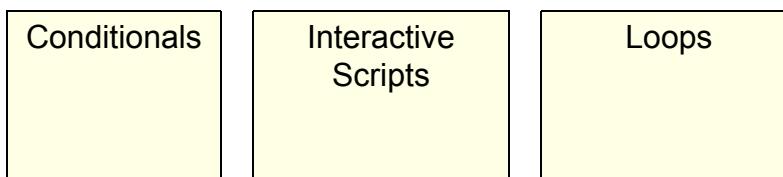
Shell Basics and Script Analysis and Design



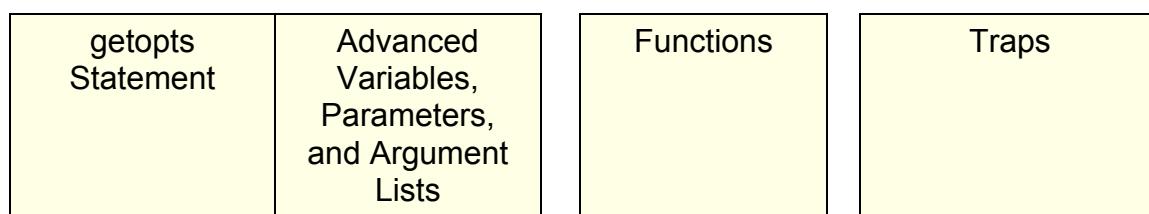
Useful Utilities and Regular Expression Characters



Programming Constructs



Advanced Shell Programming



Module-by-Module Overview

This course contains the following modules:

- Module 1 – “UNIX® Shells and Shell Scripts”

This module provides a review of the roles of a shell, an overview of the three standard operating system shells, and a brief description of all interactive shells. The module also introduces the concept and basic layout of a shell script.

Lab exercise – Answer brief questions about the shells available with the operating system, and write a brief shell script on paper.

- Module 2 – “Writing and Debugging Scripts”

This module explains the necessary components of a shell script, as well as good and poor practices. The processes of writing, executing, and debugging scripts are described.

Lab exercise – Add comments, white space, and an explicit call to a shell interpreter to the script written on paper in Lab 1. Type in and correctly execute the script. Follow directions to plant *bugs* in the scripts to practice debugging techniques.

- Module 3 – “The Shell Environment”

This module provides a review of shell variables, aliases, and initialization files. This module defines many standard shell variables and aliases, and shows user-defined variables and aliases. Then the module describes how and where these may be placed in shell initialization files.

Lab exercise – Create new users and modify the system-wide and user-specific initialization files with shell variables and aliases.

- Module 4 – “Regular Expressions and the grep Command”

This module defines and shows the use of specific regular expression characters utilized by the grep utility. This module explains and illustrates the benefit of the grep utility.

Lab exercise – Use the grep command to search for patterns in text files, and then print out the specified records from the file.

- Module 5 – “The sed Editor”

This module describe and illustrates the use of `sed`, the stream editor. The module shows how to perform non-interactive editing on a text file including adding, removing, and substituting text.

Lab exercise – Use the `sed` command to search for patterns in text files, and then modify the specified records from the file before printing the results to standard output.

- Module 6 – “The nawk Programming Language”

This module defines variables and parameters in `nawk`. This module describes and utilizes the extended regular expression characters used by `nawk`.

Lab exercise – Use the `nawk` command to process fields within records in a text data file. The `nawk` scripts processes both numeric and text variables.

- Module 7 – “Conditionals”

This module describes and illustrates shell programming constructs that perform conditional testing using the many forms of the `if` construct. This module also explains positional parameters, case constructs, Boolean logic, and the proper way to programmatically exit a script.

Lab exercise – Use the programming constructs to write scripts that test and validate input from the command line through the positional parameters. Scripts also validate path names to determine if the path is a directory or some type of file.

- Module 8 – “Interactive Scripts”

This module illustrates how to use the `read` command to obtain input from standard input, which, by default, is a user’s keyboard device. The module also shows the assignment of file descriptor numbers to text files to simplify file input and output operations.

Lab exercise – Use the `read` command in a script to obtain input from a user, have that input stored and verified before continuing, and ask the user for other information.

- Module 9 – “Loops”

This module describes and illustrates using shell programming constructs to perform looping within a shell script. This module describes the `for`, `while`, and `until` loop constructs. Additionally the module describes the menu capability of the Korn shell.

Lab exercise – Write and modify scripts to implement loops so the same operations can be applied to a number of files or directories. Modify an existing lab to include a menu of choices for the user login shell.

Module-by-Module Overview

- Module 10 – “The getopt Statement”

This module describes processing script options using the `getopts` statement.

Lab exercise – Modify a shell script to process a command-line argument with the `getopts` statement.

- Module 11 – “Advanced Variables, Parameters, and Argument Lists”

This module describes manipulation of string variables, numeric variables, and array variables. This module also covers resetting the values of the positional parameters using the `set` command.

Lab exercise – Write a script that uses metacharacters to extract a substring from a string variable. The value in the string variable is a long path name.

- Module 12 – “Functions”

This module describes the purpose, benefits, and syntax of shell functions.

Lab exercise – Write a script that defines and calls a function. Additionally, write a script that calls a function that is not contained within the script, but the function’s location is referenced within the script using a special shell variable.

- Module 13 – “Traps”

This module describes traps and signals, and shows how traps can be used within a script to selectively stop a user from interrupting the script.

Lab exercise – Write and modify scripts to create traps to catch the signal generated when the user attempts to interrupt the script.

Course Objectives

Upon completion of this course, you should be able to:

- Develop and debug scripts
- Use local and environmental variables and shell metacharacters in scripts
- Customize system-wide shell initialization files
- Use regular expression characters with the grep, sed, and nawk utilities
- Write sed scripts to perform non-interactive editing tasks
- Write nawk scripts to manipulate individual fields within a record
- Write nawk scripts to write reports based upon an input file
- Use the exit status of a command to determine if the command succeeded or failed
- Access and process command-line arguments passed into a script
- Develop a USAGE message to display when a script is invoked incorrectly
- Use flow control constructs, such as branching and looping
- Develop interactive scripts
- Perform string manipulation and integer arithmetic on shell variables
- Write a script that uses the getopt statement
- Write a script that uses functions
- Write a script that uses a trap to catch a signal

Topics Not Covered

Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Educational Services:

- Basic UNIX® commands
- System startup and shutdown procedures
- Adding a user to the system
- Logical device names for disks
- Mounting and unmounting file systems
- Adding and removing software on the system
- The JumpStart™ software

Refer to the Sun Learning Services catalog for information about available courses and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Are you an experienced UNIX user who is familiar with basic commands, such as `rm`, `cp`, `man`, `more`, `mkdir`, `ps`, and `chmod`?
- Can you create and edit text files using `vi` or a text editor?
- Do you understand the system boot process and proper shutdown procedures?
- Can you create users and passwords and set file permissions?
- Do you understand device-naming conventions to mount and unmount file systems?
- Do you know user software package commands, such as `pkgadd`, `pkgrm`, and `pkginfo`?

This course assumes that you have these prerequisite skills and knowledge:

- This is not an introductory course, so you must be comfortable with the operating system commands and must be able to read the online manual pages.
- To write and save scripts, you must be comfortable with some editor.
- You must be familiar with several boot scripts and system shutdown in regard to system backups.
- User creation is a typical system administration task, and it is used in the course as a scripting exercise.
- References to disk devices for the purposes of monitoring and reporting are typical system administration issues and are used in the course as a scripting exercise.
- Software installation of unbundled or third-party software is a typical system administration task, and it is used in the course as a scripting exercise.
- If you learn best from code examples and technical explanations, you will be able to program the lab exercises based on the explanations and examples presented in the lecture.

Introductions

Introductions

Now that you have been introduced to the course, introduce yourself to each other and the instructor, addressing the following items.

- Name
- Company affiliation
- Title, function, and job responsibility
- System administration experience
- Experience modifying or writing scripts
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials use a learning model that comprises the following components:

- **Goals** – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- **Course map** – An overview of the course content appears in the “About This Course” module so that you can see how each module fits into the overall course goal.
- **Objectives** - What you should be able to accomplish after completing this module is listed here.
- **Lecture** – The instructor will present information specific to the topic of the module. This information helps you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** – Lab exercises give you the opportunity to practice your skills and apply the concepts presented in the lecture.

Conventions

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons



Additional resources – Indicates other references that provide additional information on the topics described in the module.



Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.



Note – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.



Caution – Indicates that there is a risk of personal injury from a nonelectrical hazard, or risk of irreversible damage to data, software, or the operating system. A caution indicates that the possibility of a hazard (as opposed to certainty) might happen, depending on the action of the user.



Caution – Indicates that either personal injury or irreversible damage of data, software, or the operating system will occur if the user performs this action. A warning does not indicate potential events; if the action is performed, catastrophic events will occur.

Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

Use `ls -al` to list all files.
system% You have mail.

Courier is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

The `getServletInfo` method is used to get author information.
The `java.awt.Dialog` class contains `Dialog` constructor.

Courier bold is used for characters and numbers that you type; for example:

To list the files in this directory, type:
`# ls`

Courier bold is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;  
2 import javax.servlet.*;  
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life cycle methods (Line 2).

Courier italics is used for variables and command-line placeholders that are replaced with a real name or value; for example:

To delete a file, use the `rm filename` command.

Courier italic bold is used to represent variables whose values are to be entered by the student as part of an activity; for example:

Type `chmod a+rwx filename` to grant read, write, and execute rights for `filename` to world, group, and users.

Palatino italics is used for book titles, new words or terms, or words that you want to emphasize; for example:

Read Chapter 6 in the *User's Guide*.
These are called *class* options.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 1

UNIX® Shells and Shell Scripts

Objectives

Upon completion of this module, you should be able to:

- Describe the role of shells in the UNIX environment
- Describe the standard shells
- Define shell script components
- Write a simple shell script

What Is a Shell?

What Is a Shell?

A shell is a program that provides an interface between a user and an operating system (OS) kernel. An OS starts a shell for each user when the user logs in or opens a terminal or console window.

A kernel is a program that:

- Controls all computer operations.
- Coordinates all executing utilities
- Ensures that executing utilities do not interfere with each other or consume all system resources.
- Schedules and manages all system processes.

By interfacing with a kernel, a shell provides a way for a user to execute utilities and programs.

What Are Shell Functions?

The following sections describe the main functions of a shell.

Command-Line Interpreter

When you type a command on the command line, the shell:

1. Interprets the command by parsing the line and handling metacharacters, redirection, and control.
2. Searches for and executes the command.
3. Analyzes each line and initiates execution of the requested program.

For example, when you issue the command `ps -ef | sort +1 | more`, the shell does the following:

1. Breaks the command line into pieces called *words*:
`ps`, `-ef`, `|`, `sort`, `+1`, `|`, and `more`.
2. Determines the *word* purpose:
 - *ps, sort, and more are commands*
 - *-ef and +1 are arguments*
 - *| is an input/output (I/O) operation.*
3. Based upon the shell-parse order, the shell sets up the `ps` output to be the input to `sort`, and the `sort` output to be the input to `more`.
4. Locates the `ps`, `sort`, and `more` commands and then executes them in order, applying the arguments as specified on the command line.

Programming Language

You can type the commands directly into the shell at the prompt, or the shell can read commands from a file. A file containing shell commands is called a *shell program* or a *shell script*. As you perform a task repetitively, you might type the same commands over and over again. You can automate this task by placing these commands in a file and executing them as a shell script.

A shell script can contain commands and programming constructs. In this way, the script can execute based on conditions, loop through a series of commands, and test files.

What Are Shell Functions?

Shell programs are text files that are interpreted, not compiled. The shell reads a line and processes all statements found on that line before reading the next line. Compiled programs written in programming languages such as C and C++ are text files that are preprocessed by a compiler to generate a standalone binary executable. Binary executables are loaded entirely into memory at the time of execution and their standalone nature means they do not require an initial load of an interpreter to process their statements. This is the reason that even the fastest shell programs always run slower than an equivalent program written in a compiled language.

Note – A command placed in a shell script is a *statement*. For example, cat on the command line is a command, and cat in a script is a *statement*. In this course, you will see references command names (for example, cat) both as a command and as a statement in a shell script.

User Environment

The shell also provides a user environment that you can customize using initialization files. These files contain settings for user environment characteristics, such as:

- Search paths for finding commands.
- Default permissions on new files.
- Values for variables that other programs use.
- Values that you can customize.

The following sections describe Solaris OS shells. The shell script examples in this course cover Bourne and Korn shells. Shell features and their default prompts are also described.

Note – If the user entry in the /etc/passwd file doesn't have an entry in the shell field, the user gets a Bourne shell.

Solaris OS Shells

The following sections describe Solaris OS shells.

The Bourne Shell

The Bourne shell (`sh`), written by Steve Bourne at AT&T Bell Labs, is the original UNIX shell. It is the preferred shell for shell programming because of its compactness and speed. A Bourne shell drawback is that it lacks features for interactive use, such as the ability to recall previous commands (history). The Bourne shell also lacks built-in arithmetic and logical expression handling.

The Bourne shell is the Solaris OS default shell. It is the standard shell for Solaris system administration scripts. For the Bourne shell the:

- Command full-path name is `/bin/sh` and `/sbin/sh`.
- Non-root user default prompt is `$`.
- Root user default prompt is `#`.

The C Shell

The C shell (`csh`):

- Is a UNIX enhancement written by Bill Joy at the University of California at Berkeley.
- Incorporated features for interactive use, such as *aliases* and *command history*.
- Includes convenient programming features, such as built-in arithmetic and a C-like expression syntax.

The syntax of C shell programs is similar to the C programming language, hence the name. These features make the C shell:

- Preferable for interactive use.
- Larger and slower than the Bourne shell.

Solaris OS Shells

For the C shell the:

- Command full-path name is /bin/csh.
- Non-root user default prompt is *hostname* %.
- Root user default prompt is *hostname* #.

The Korn Shell

The Korn shell (ksh):

- Was written by David Korn at AT&T Bell Labs
- Is a superset of the Bourne shell.
- Supports everything in the Bourne shell.
- Has interactive features comparable to those in the C shell.
- Includes convenient programming features like built-in arithmetic and C-like arrays, functions, and string-manipulation facilities.
- Is faster than the C shell.
- Runs scripts written for the Bourne shell.

For the Korn shell the:

- Command full-path name is /bin/ksh.
- Non-root user default prompt is \$.
- Root user default prompt is #.

Note – The Bourne, C, and Korn shells were always part of the Solaris OS.

Additional Solaris OS Shells

This section introduces additional Solaris OS shells and their features.

The GNU Bourne-Again Shell

The GNU Bourne-Again shell (bash):

- Is compatible to the Bourne shell.
- Incorporates useful features from the Korn and C shells.
- Has arrow keys that are automatically mapped for command recall and editing.

For the GNU Bourne-Again shell the:

- Command full-path name is /bin/bash.
- Default prompt for a non-root user is bash-x.xx\$.
Where x.xx indicates the shell version number. For example,
bash-3.50\$
- Root user default prompt is bash-x.xx#.
Where x.xx indicates the shell version number. For example,
bash-3.50\$#

The Desktop Korn Shell

The Desktop Korn shell (dtksh) is a version of the Korn shell extended to support X, Xt, and Motif facilities from within a shell script. Therefore, you can open windows with text fields, buttons, labels, scroll bars, menus, and so on from within a Desktop Korn shell script. These windows are limited to a Common Desktop Environment (CDE) windowing environment.

For the Desktop Korn shell the:

- Command full-path name is /usr/dt/bin/dtksh.
- Non-root user default prompt is \$.
- Root user default prompt is #.

The Job Control Shell

The Job Control shell (`jsh`) is an interface to the Bourne shell that provides all of the functionality of `sh` and enables job control. The Job Control shell executables are linked to the corresponding Bourne shell executables. Use the following commands to see how it works:

```
# ls -l /bin/jsh /bin/sh  
  
# ls -l /sbin/jsh /sbin/sh
```

Although the executables for the Bourne and Job Control shells are identical, invoking the shell with the `jsh` command allows you to not only put jobs in the background with an `&`, but also to list the background processes with `jobs`, and remove them with the `kill %job#` command.

For the Job Control shell the:

- Command full-path names are `/bin/jsh` and `/sbin/jsh`.
- Non-root user default prompt is `$`.
- Root user default prompt is `#`.

The Restricted Shell Command Interpreter

The Restricted shell (`rsh`) is a version of the Bourne shell that restricts logins to user environments in which capabilities are controlled. For example, in a Restricted shell, the user may not:

- Change the directory from the present one.
- Change the value of `$PATH`.
- Specify path or command names containing `/`.
- Redirect output.

For more information, see the man page:

```
# man -s 1m rsh
```

For the Restricted shell the:

- Command full-path name is /usr/lib/rsh.
- Non-root user default prompt is \$.
- Root user default prompt is #.

The Enhanced C Shell

The Enhanced C shell (tcsh):

- Is compatible with the C shell.
- Has command-line editor, programmable word completion, and spelling correction features.
- Has arrow keys that are automatically mapped for command recall and editing.

For the Enhanced C shell the:

- Command full-path name is /bin/tcsh.
- Non-root user default prompt is >.
- Root user default prompt is #.

Z Shell

The Z shell (zsh):

- Closely resembles the Korn shell.
- Includes built-in spelling correction and programmable command completion.

For the Z shell the:

- Command full-path name is /bin/zsh.
- Non-root user default prompt is *hostname*%.
- Root default prompt is *hostname*#.

Additional Solaris OS Shells

To change your current shell, type in an alternate shell (command) at the prompt. For example, if you are in the Bourne shell, type `csh` at the prompt to change your shell to the C shell.

To determine your current shell, execute the `ps` command. If you have opened a series of shells, use the `ps | sort` command to sort the shells in the order in which you invoked them.

Shells Discussed in This Course

This course limits the discussion of shells to the Bourne and Korn shells. The Korn shell is a superset of the Bourne shell, so all descriptions apply to both shells, unless otherwise noted. When a feature applies only to the Korn shell, it is noted.

Subshells – Child Processes

The shell determines the type of the program that was executed. It is either a compiled (executable) program or a script file for one of the available shells. If it is an executable program, a new process is created for the program by means of a *fork*, which makes a copy of the shell process. The new process is called a *child process* or *subshell*. The kernel loads the program into the child process (called *exec*'ing the program). The program then begins running, and the shell process waits until the child process finishes.

When the program being executed is a shell script, the execution process is a little different. The shell creates a new shell process (also by *forking*). This new process (subshell) reads the lines from the shell script file one at a time. The subshell reads, interprets, and executes each line as if it had come from the keyboard. The entire execution process that is described in this module is repeated for each line of the shell script.

While the subshell processes the lines of the shell script, the parent shell waits for it to finish. When there are no more lines in the shell script file to read, the subshell terminates and the parent shell awakes, displaying a new prompt.

Subshells – Child Processes

Figure 1-1 shows the order of execution of a shell script.

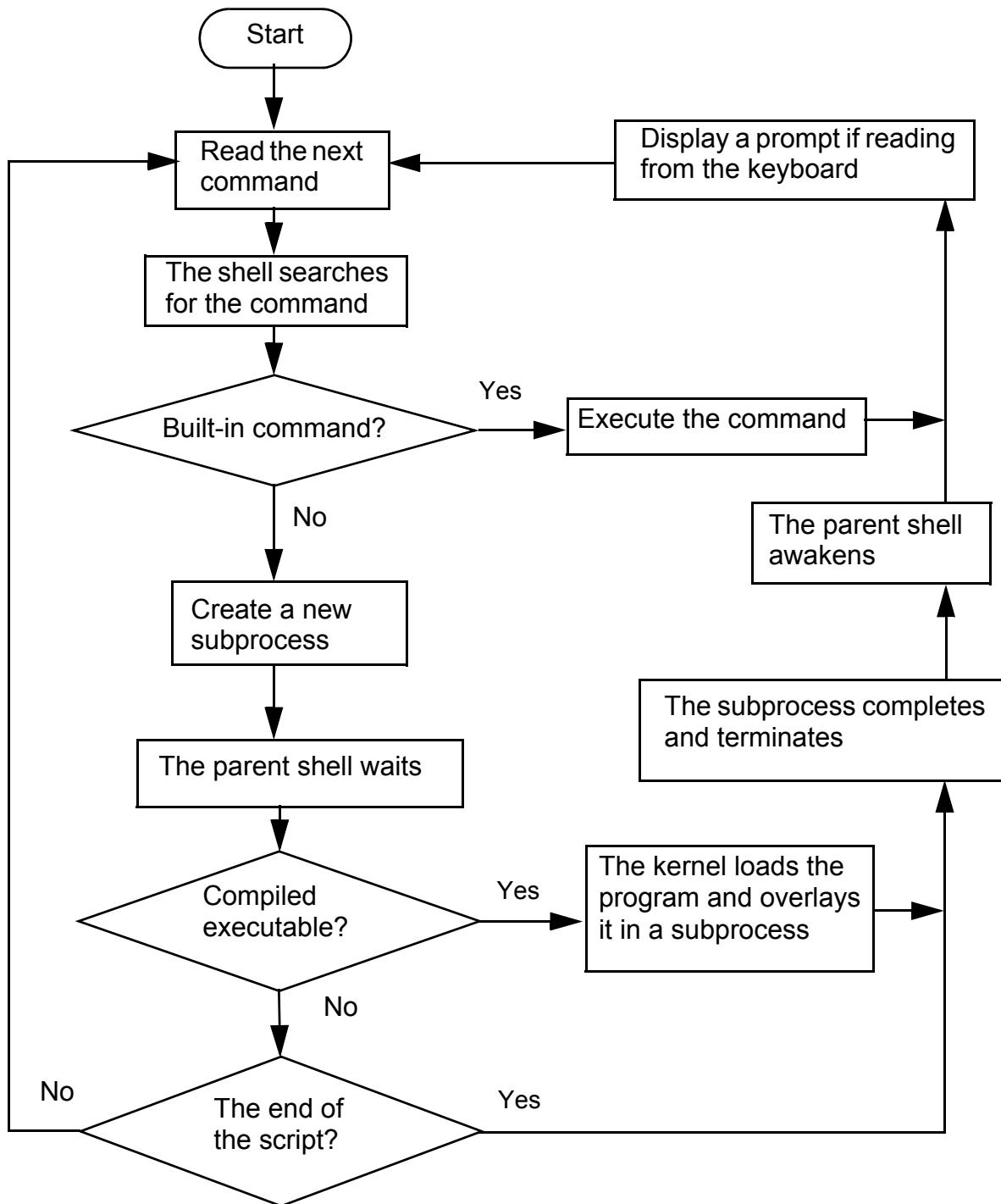


Figure 1-1 Shell Script Order of Execution

What Is a Shell Script?

A shell script is a file that contains shell and UNIX commands. Similar to compiled executable programs, the shell script has a specific purpose and is reusable. When the commands in the shell script are ordered and error-free, you can execute the shell script to carry out the tasks efficiently.

All scripts, programs, and procedures use the following rules:

- They should execute without error.
- They perform the task for which they are intended.
- They ensure that the program logic is clearly defined or apparent.
- They should not incorporate unnecessary work.

Procedures are common in everyday work. For example, procedures are used when:

- A system boots.
- You create a user on a system.
- You log in.

All programs and procedures are a series of tasks that are organized into a logical sequence, sometimes with certain sequences being repeated.

Developing a Script

Developing a Script

The structure of a shell script is flexible. You must ensure correct logic, flow control, and efficiency before tasks can be carried out easily and correctly by the user who executes the script.

When you start to develop your script, determine the types of information needed before writing the script. For example, your script might:

- Prompt the user for the directory name.
- Read and store the directory name in a variable.
- Verify that the directory exists and that the user has permissions for the directory.
- Execute the `tar` command and create the archive file (perhaps in a temporary directory).
- Compress the archive file (into the current directory).
- Notify the user of the name of the compressed file.

Other script components might include:

- Comments
- Information displays
- Conditional testing
- Loops
- Arithmetic
- String manipulation
- Variable manipulation and testing
- Argument and option handling

Programming Terminology

Table 1-1 describes terminology you use when designing and writing programs or procedures.

Table 1-1 Terminology

Term	Meaning
Logic flow	The overall design of the program or procedure. The logic flow determines the logical sequence of the tasks that are involved in the program or procedure so that a successful and controlled outcome is achieved.
Loop	A portion of the procedure that is performed zero or more times.
User input	Information that is provided by an external source, during the running of the program or procedure, that is used within the program or procedure. This information can be stored so that it can be recalled if needed.
Conditional branch	A logical point in the program or procedure when a condition determines the actions that are subsequently performed.
Command control	Testing the exit status of a command to control whether a portion of code should be performed.

Logic-Flow Design

Logic-Flow Design

A program must follow a logical order. When designing a program or procedure, start with the flow of logic for that program.

A number of standard methods are used in computing. One of the simplest methods of logic-flow design is itemized lists. By itemizing the list of processes involved in the program, you can describe each task, or process, and refer to each individual task by its associated number.

If you start by writing the tasks in a descriptive format using the English language, the program is in an understandable form. Later, you can transform the English-language statements into the appropriate computer-language statements.

Here is an example of a logic-flow design. This example procedure describes the process of adding new users to your system.

1. Do you want to add a user?
 - a. If Yes:
 1. Enter the user's name.
 2. Choose a shell for the user.
 3. Determine the user's home directory.
 4. Determine the group to which the user belongs.
 - b. If No, go to Step 3.
2. Do you want to add another user?
 - a. If Yes, go to Step 1.a.
 - b. If No, go to Step 3.
3. Exit.

This example shows a possible repetitive loop, which is controlled by the number of users, between Steps 1.a and 2.a.

The user must be provided with information, such as "These are the available shells (sh, ksh, csh). Which would the user prefer?"

Additionally, input from the user must be obtained and stored; for example, the choice for user name, shell, and group.

The echoscript1.sh Example Bourne Script

The echoscript1.sh script uses some familiar commands to display information about the workstation.

```
# cat echoscript1.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME"
echo

echo "Todays date is: \c"
date '+%m/%d/%y'

echo "and the current time is: \c"
date '+%H:%M:%S%N'

echo "Now a list of the processes in the current shell"
ps

echo "SCRIPT FINISHED!!"
```

Note – The specification of full path names to commands is not necessary if you plan to be logged in when you execute the script and you are sure that your PATH variable uses the correct version of every command (/bin/ps instead of /usr/ucb/ps).

If you are executing this script from cron, you must supply full path names and redirect output and errors.

The echoscript2.ksh Example Korn Script

The echoscript2.ksh Example Korn Script

The echoscript2.ksh script uses some familiar commands to display information about the workstation. The Korn shell version uses the print command rather than the echo command.

```
$ cat echoscript2.ksh
#!/bin/ksh

clear
print "SCRIPT BEGINS"

print "Hello $LOGNAME"
print

print -n "Todays date is: "
date '+%m/%d/%y'

print -n "and the current time is: "
date '+%H:%M:%S%n'

print "Now a list of the processes in the current
shell"
ps

print "SCRIPT FINISHED!!"
```

The /etc/init.d/volmgt Example Boot Script

This system script starts and stops the volume manager daemon (vold). Notice the system commands used. Module 7, “Conditionals,” revisits the script and looks at each line to define what is specifically being performed.

```
$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright 2006 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# ident "@(#)volmgt      1.9      06/01/20 SMI"

case "$1" in
'start')
    if [ -f /etc/vold.conf -a -f /usr/sbin/vold -a \
        "${_INIT_ZONENAME:='/sbin/zonename'}" = "global" ]; then
        echo 'volume management starting.'
        svcadm enable svc:/system/filesystem/volfs:default
    fi
    ;;

'stop')
    svcadm disable svc:/system/filesystem/volfs:default
    ;;

*)
    echo "Usage: $0 { start | stop }"
    exit 1
    ;;

esac
exit 0
```

Note – Full path names are used in the UNIX commands. They are required because they are executed during system boot and no PATH variable is set.

Exercise: Reviewing UNIX Shells and Shell Scripts

Exercise: Reviewing UNIX Shells and Shell Scripts

Exercise objective – Answer questions pertaining to the Solaris OS-supplied shells and write a simple shell script on paper based on the specified requirements.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

When writing the script on paper, you may want to execute commands on the system to ensure you have the correct command and option to obtain the prescribed output.

Tasks

1. Describe the functions of the UNIX shells.

2. Name and describe the three standard shells found in the Solaris OS.

3. Name any three components of a shell script.

4. On paper, write an ordered list of shell commands to yield the following output. (In the following module, you will modify the script, enter the script into the system using an editor, and test the script.)
 - *The system name (hint: use uname)*
 - *The hardware platform (for example, SUNW Ultra-5_10) (hint: use uname)*
 - *Processor-specific information including clock speed (hint: use psrinfo)*
 - *The total number of processes running on the system (hint: use ps and wc)*

Exercise Summary

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Describe the functions of the UNIX shells.

They provide:

- *Command-line interpretation*
- *A programming language*
- *Interactive user environment*

2. Name and describe the three standard shells found in the Solaris Operating Environment.

- *Bourne shell – The original UNIX shell written at AT&T Bell Labs, which is the preferred shell for scripting because of its compactness and speed. It lacks features often expected by users for frequent interactive use.*
- *C shell – Written at the University of California at Berkeley. The programming constructs were borrowed from the C language, hence the name of the shell. It includes the history and alias capabilities frequently used by users and programmers.*
- *Korn shell – Written at AT&T Bell Labs. This shell uses the programming constructs of the Bourne shell and the history and alias capabilities of the C shell and then adds features, such as integer arithmetic, arrays, and string manipulation.*

3. Name any three components of a shell script.

- *Comments*
- *Information displays (output)*
- *Conditional testing*
- *Loops*
- *Arithmetic*
- *String manipulation*
- *Variable manipulation and testing*
- *Argument and option handling*

Task Solutions

4. On paper, write an ordered list of shell commands to yield the following output. (In the following module, you will modify the script, enter the script into the system using an editor, and test the script.)

- *The system name (hint: use uname)*
- *The hardware platform (for example, SUNW_Ultra-5_10) (hint: use uname)*
- *Processor-specific information including clock speed (hint: use psrinfo)*
- *The total number of processes running on the system (hint: use ps and wc)*

The commands to produce the output should resemble the following:

```
echo
echo "The name of the machine is \c"
uname -n          # uname -n gives the host name
echo

echo "The machine platform type is \c"
uname -i          # uname -i gives the hardware platform
echo

echo "Specific processor information for this machine is:"
/usr/sbin/psrinfo -v
echo

echo "Currently the total number of processes "
echo " running on this machine is \c"
ps -ef | tail +2 | wc -l      # wc -l gives number of lines
```

Module 2

Writing and Debugging Scripts

Objectives

Upon completion of this module, you should be able to:

- Start a script with #!
- Put comments in a script
- Change permissions on a script
- Execute a script
- Debug a script

Creating Shell Scripts

Creating Shell Scripts

A shell script is a text file that contains a sequence of UNIX commands. The shell script is often a command sequence for which you have a repeated use. Typically, you execute the sequence of commands in the shell script by entering the name of the shell script on the command line.

To create a shell script, create a file, and put UNIX commands into it. UNIX commands include standard utilities, user programs, and the names of other scripts that you need to accomplish your task.

Note – If you put the names of other scripts in your script, be sure they have *read* and *execute* permission so they will run.

A script file can have any name following the conventions of regular file names in the operating-system environment. When naming your shell programs you should avoid using names that conflict with existing UNIX commands or shell functions.

The following example script uses the echo command to inform the user about the output to expect before the actual command that creates the output is called. It is good practice when writing scripts to inform users about what is happening so they have an idea of whether the script is running correctly.

```
$ vi firstscript.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME!"
echo

echo "Todays date and time: \c"
date
echo

mynum=21
myday="Monday"

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

Executing a Shell Script

To execute a shell script as you do other commands on the system, first assign it the execute permission with the chmod command; for example :

```
$ chmod 744 scriptname  
$ scriptname
```

This is the preferred and most common method for executing a shell script. A subshell is created to execute the specified script. Give execute permission only to users who need to execute the script.

When a script is executed in a subshell, the variables, aliases, and functions created and used in the script are known only in the subshell. After the script finishes and control returns to the parent shell, the variables, functions, and other changes to the state of the shell made by the script are no longer known.

Generally, you execute a script using the shell command only to invoke certain options on a script that you do not have permission to execute (*or do not want to execute*).

```
$ ksh scriptname  
$ sh scriptname
```

Again, a subshell is created to execute the specified script. This is used in situations when you want the shell to start with specific options not explicitly specified in the script. The typical situation is when you want to turn on and off debugging for the entire script.

The following example shows executing a script with debugging:

```
$ ksh -x scriptname  
$ sh -x scriptname
```

The following example runs the script in the current shell:

```
$ . ./scriptname
```

To execute the shell script in the current shell, you do not need to execute permissions on the script. You are running the commands in the current shell, so any operations you perform on shell variables or aliases take affect in the current shell and are seen after the shell script terminates.

Executing the firstscript.sh Script

Executing the firstscript.sh Script

The following is the firstscript.sh script.

```
$ cat firstscript.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME !"
echo

echo "Todays date and time: \c"
date
echo

mynum=21
myday="Monday"

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

Variables created by the script are not known to the shell that executed it. The script is executed in a subshell, and changes made there do not affect the parent shell.

```
$ chmod 744 firstscript.sh
$ ./firstscript.sh
SCRIPT BEGINS
Hello root!

Todays date and time: Tue May  5 11:38:13 MDT 2009

The value of mynum is 21
The value of myday is Monday

SCRIPT FINISHED!!
$ echo $mynum
$ echo $myday

$ sh firstscript.sh
SCRIPT BEGINS
Hello root!

Todays date and time: Tue May  5 11:38:57 MDT 2009

The value of mynum is 21
The value of myday is Monday

SCRIPT FINISHED!!
$ echo $mynum
$ echo $myday
```

Starting a Script With the #! Characters

Starting a Script With the #! Characters

When running a script that has *execute* permission by typing the script name on the command line, you should know the subshell (sh, csh, ksh, bash, or other subshell) that will run the script.

The first line of the script determines the shell that is created as the interpreter of the script. The first two characters on the first line in a script are #!. The current shell interprets what follows as the path name for the subshell to execute the script.

Note – “*The first two characters on the first line*” means there must be no blank lines or spaces before these characters.

In the following examples, the specified shell is forked as the subshell to run the script.

```
#!/bin/sh
```

```
#!/bin/csh
```

```
#!/bin/ksh
```

Although the syntax #! *path_for_shell* allows the script writer to specify exactly in what shell the script should be run, not all operating systems recognize this syntax. Therefore, scripts written in this way might not run in exactly the same way on all operating systems.

Base the shell you choose on two rules:

1. For a system boot script, use the Bourne shell (/sbin/sh).

On Solaris 9 and earlier, when it is possible that only the root file system may be mounted, use the only shell that exists in the root file system, /sbin/sh. Remember the system library files are in /usr/lib. Starting Solaris 10, all shells reside on root as well as the libraries they are linked against.

2. For a typical script, use the shell that you are most comfortable with or the one that supports all the features you need.

Hence, if you want to use the alias feature or true integer variables, use the Korn shell. If you know the C language, the C shell might be the best shell for you to use.

Putting Comments in a Script

The persons who write scripts usually are not the only persons who read them. Many persons run scripts written by others. If they view the contents of the script file to see what the script does, it is helpful if the author of the script included some explanations in comments.

It is a good practice to put comments into programs and shell scripts. The comments should explain the purpose of the script and should explain any specific lines that might be especially confusing.

The example script, `scriptwithcomments.sh`, has a comment at the beginning of the script explaining what the script does when it runs. The line with the date statement contains a comment explaining the output produced by the date statement.

```
$ cat scriptwithcomments.sh
#!/bin/sh

# This script clears the window, greets the user,
# and displays the current date and time.

clear                                     # Clear the window
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME!" # Greet the user
echo

echo "Todays date and time: \c"           # Display current date and time
date
echo

mynum=21                                    # Set a local shell variable
myday="Monday"                             # Set a local shell variable

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

Putting Comments in a Script

```
$ ./scriptwithcomments.sh  
SCRIPT BEGINS  
Hello user1!
```

Todays date and time: Tue Nov 24 15:01:22 IST 2009

The value of mynum is 21
The value of myday is Monday

SCRIPT FINISHED!!

The addition of comments does not affect the execution of the script unless a syntactical error is introduced when the comments are added. The comments are there for documentation purposes, so someone reading the script will have an idea of what will occur when the script is executed.

Comments are sometimes the first lines entered in a shell script. The comments can list the step-by-step operations that need to take place. Follow each comment line by the operating system commands to carry out the operation.

Adding the Debugging Statement

When a script does not work properly, determine the location of the problem. The shell provides a debugging mode. Run the entire script in debug mode or just a portion of the script. To run an entire script in debug mode, add `-x` after the `#!/bin/ksh` on the first line:

```
#!/bin/ksh -x
```

To run an entire script in debug mode from the command line, add a `-x` to the `ksh` command used to execute the script:

```
$ ksh -x script_name
```

Run several portions of a script in debug mode by placing the `set -x` option at the point where debugging is to begin and the `set +x` option where you want it to stop. Do this as many times in the script as you want. The debug options are shown in Table 2-1.

Table 2-1 Debug Statement Options

Option	Meaning
<code>set -x</code>	Prints the statements after interpreting metacharacters and variables
<code>set +x</code>	Stops the printing of statements
<code>set -v</code>	Prints the statements before interpreting metacharacters and variables
<code>set -f</code>	Disables file name generation (using metacharacters)

The `set -x` option allows you to see the statement line after the shell interpreted any metacharacters and performed any statement, variable, or file name substitutions. For the portion of the script file running in this debug mode, each statement is output, preceded by a `+`, before the statement is executed.

Debug Mode Controls

Debug Mode Controls

The `set -v` statement is similar to `set -x`, except it shows the statement line *before* the shell performs any interpretation or substitution. When you use these two options, with `set -xv`, the shell displays the statement lines both before and after interpretation and substitution, so that you can see how the shell has used the statement line in the script.

For the following statement in a shell script:

```
echo $HOME      # echo the HOME directory path
```

The statement line, before interpretation (`set -v`) is:

```
echo $HOME      # echo the HOME directory path
```

The statement line, after interpretation (`set -x`) is:

```
+ echo /export/home/user200
```

If both options are turned on, the `-v` line is always displayed before the `-x` line. Also, the `+` character does not precede the `-v` line.

The `set -f` option turns off the shell file name substitution (globbing) capability. This disables the special meaning of characters, such as `*` and `?`, when used as part of a file or directory name.

To turn on the option, you must give the statement `set -option`, where *option* is one, or a combination, of the letters `x`, `v`, and `f`.

To turn off the option, you must give the statement, `set +option`, where *option* is one, or a combination, of the letters `x`, `v` and `f`.

Use the options on the first line of the script by adding them to the end of the shell's path name; for example:

```
#!/bin/ksh -xvf
```

Example: Debug Mode Specified on the #! Line

The following examples show the debug1.sh script being run once with no debugging turned on, and then in debug mode. In debug mode, each command is displayed with a + in front of it, and then it is executed. If you remove all lines with a + at the beginning, the remainder is the output from the normal running of the script.

```
$ cat debug1.sh
#!/bin/sh

echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo

$ ./debug1.sh
Your terminal type is set to: ansi

Your login name is: root

Now we will list the contents of the /etc/security directory
audit          audit_user        dev           policy.conf
audit_class    audit_warn       device_policy  priv_names
audit_control  auth_attr       exec_attr     prof_attr
audit_event    bsmconv         extra_privs   spool
audit_record_attr bsmunconv    kmfpolicy.xml tsol
audit_startup  crypt.conf      lib
```

Example: Debug Mode Specified on the #! Line

```
$ vi debug1.sh
< append a "-x" to the first line >
< write and save the file >
$ 

$ cat debug1.sh
#!/bin/sh -x

echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo

$ ./debug1.sh
+ echo Your terminal type is set to: ansi
Your terminal type is set to: ansi
+ echo

+ echo Your login name is: root
Your login name is: root
+ echo

+ echo Now we will list the contents of the /etc/security directory
Now we will list the contents of the /etc/security directory
+ ls /etc/security
audit           audit_user      dev          policy.conf
audit_class     audit_warn     device_policy priv_names
audit_control   auth_attr      exec_attr    prof_attr
audit_event     bsmconv       extra_privs  spool
audit_record_attr bsmunconv   kmfpolicy.xml tsol
audit_startup   crypt.conf    lib
+ echo
```

Example: Debug Mode With the set -x Option

The following example shows the debug mode turned on for only a portion of the script.

```
$ cat debug2.sh
#!/bin/sh

set -x
echo "Your terminal type is set to: $TERM"
echo
set +x

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo

$ ./debug2.sh
+ echo Your terminal type is set to: ansi
Your terminal type is set to: ansi
+ echo

+ set +x
Your login name is: root

Now we will list the contents of the /etc/security directory
audit          audit_user        dev           policy.conf
audit_class    audit_warn       device_policy  priv_names
audit_control  auth_attr       exec_attr     prof_attr
audit_event    bsmconv         extra_privs   spool
audit_record_attr bsmunconv   kmfpolicy.xml tsol
audit_startup  crypt.conf      lib
```

Example: Debug Mode With the set -v Option

Example: Debug Mode With the set -v Option

The following example shows the -v debug mode turned on for the entire script. This option:

- Prints the statement prior to executing but without a leading + (unlike what the -x option displays).
- Does not translate metacharacters; therefore, it prints the statement exactly as it is in the script.
- Prints comment lines.

```
$ cat debug3.ksh
#!/bin/ksh

set -v
echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

echo "Now we will list the contents of the /etc/security directory"
ls /etc/security
echo

$ ./debug3.ksh
echo "Your terminal type is set to: $TERM"
Your terminal type is set to: ansi
echo

echo "Your login name is: $LOGNAME"
Your login name is: root
echo

echo "Now we will list the contents of the /etc/security directory"
Now we will list the contents of the /etc/security directory
ls /etc/security
audit          audit_user        dev           policy.conf
audit_class    audit_warn       device_policy  priv_names
audit_control  auth_attr       exec_attr     prof_attr
audit_event    bsmconv         extra_privs   spool
audit_record_attr bsmunconv    kmfpolicy.xml tsol
audit_startup  crypt.conf      lib
echo
```

Syntax Comparison for Bourne and Korn Shell Options

Table 2-2 compares the syntax for options in the Bourne and Korn shells.

Table 2-2 Comparison of Syntax

Bourne and Korn Options	Korn Shell-Specific Options
set -x	set -o xtrace
set +x	set +o xtrace
set -v	set -o verbose
set +v	set +o verbose
set -f	set -o noglob
set +f	set +o noglob

Example: Debug Mode With the set -o noglob Option

Example: Debug Mode With the set -o noglob Option

The following example shows the Korn-specific syntax using debug option names. In this example, the debug option is turned on for only a portion of the script by using the `set -o` option. In this example:

- the `noglob` option disables file name substitution.
- the `noglob` option is turned on only for a portion of the script.
- you can compare the results for the same command when the debug option is turned on and subsequently when it is turned off.

```
$ cat debug4.ksh
#!/bin/ksh

echo "Your terminal type is set to: $TERM"
echo

echo "Your login name is: $LOGNAME"
echo

set -o noglob
echo "Now we will list the contents of the /etc/security directory"
ls /etc/secur*
echo
set +o noglob

echo "Now we will list the contents of the /etc/security directory"
ls /etc/secur*
echo

$ ./debug4.ksh
Your terminal type is set to: ansi

Your login name is: root

Now we will list the contents of the /etc/security directory
/etc/secur*: No such file or directory

Now we will list the contents of the /etc/security directory
audit          audit_user      dev           policy.conf
audit_class    audit_warn     device_policy  priv_names
audit_control  auth_attr     exec_attr     prof_attr
audit_event    bsmconv       extra_privs   spool
audit_record_attr bsmunconv  kmfpolicy.xml tsol
audit_startup  crypt.conf    lib
```

Exercise: Writing Shell Scripts

Exercise objective – Type and correctly execute the script you wrote in the previous module exercise. Execute modified versions of the scripts. The modifications are based on errors you are directed to enter. This exercise shows the benefit of the different debug options (-x, -v, and -f).

Preparation

Locate the script you wrote for the exercise in the last module, and refer to the lecture notes as necessary to perform the following tasks.

Change to the mod2/lab directory prior to beginning the exercise.

Tasks

1. Type the script you wrote for the end-of-module exercise in Module 1, “UNIX® Shells and Shell Scripts.” Name the script `echoscript.sh`. Given the topics of the current module, make sure you:
 - Include a first line that calls the Bourne shell as the interpreter
 - Add a comment to state the script’s purpose
 - Add any other comments that you deem necessary
2. Change the permissions on `echoscript.sh` so you can execute the script as a command.
3. Execute the `echoscript.sh` script. If this script does not execute correctly, determine the problem, re-edit, and execute again. If debug options would help in fixing the error, add debugging commands.
4. Copy the `debugscript.sh` script to `debug1.sh`. Read `debug1.sh` and note the different debug options entered in comment statements. This makes it easier to execute the script and test the option that gives you the most useful information.
5. Execute `debug1.sh`. Does it appear to execute correctly?

Exercise: Writing Shell Scripts

6. Misspell one of the echo statements near the middle of the script. Re-execute the script, and record the error message given. Then correct the spelling.

7. Change the statement in debug1.sh that prints out the value of the `logname` variable instead of `LOGNAME`. Therefore, change:

`echo "Your login name is: $LOGNAME"`

to:

`echo "Your login name is: $logname"`

8. Re-execute `debug1.sh`, and write down the output given for the modified line.

9. Now re-execute the `debug1.sh` script after turning on each of the three different debug options one at a time. Record whether each option gives you any help in determining the problem.

`set -v` _____

`set -x` _____

`set -f` _____

10. Which debug options gave you information to help point to the variable name problem?

11. Copy the `debug1.sh` script to `debug2.sh`.

12. Modify the `debug2.sh` script to:

- Run the Korn shell interpreter rather than the Bourne shell
- Turn on the verbose debug mode with the Korn shell-specific syntax (not `set -v`) just before the first `echo` statement

13. Execute the `debug2.sh` script, and compare the output to the output of `debug1.sh` executing with the `-v` option. Are the two script outputs the same?

14. Write a template shell script that contains basic information that should be included for each script that you write. It should provide documentation, in the form of comments, for each script you write in the future. Later when creating a new script, copy the template, fill in the information, and proceed to enter the statements into the script.

The template should include:

- The shell interpreter
- The name of the script
- The author of the script
- The date the script was written
- A brief description of the purpose of the script

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Type the script you wrote for the end-of-module exercise in Module 1, “UNIX® Shells and Shell Scripts. Name the script `echoscript.sh`. Given the topics of the current module, make sure you:
 - Include a first line that calls the Bourne shell as the interpreter
 - Add a comment to state the scripts purpose
 - Add any other comments that you deem necessary

```
$ vi echoscript.sh
#!/bin/sh

# This script will print useful information about the current machine.
# It will print the machine name, model type, specific processor
# information, and the current number of processes running.

echo
echo "The name of the machine is \c"
uname -n          # uname -n gives the hostname
echo

echo "The machine platform type is \c"
uname -i          # uname -i gives the hardware platform
echo

echo "Specific processor information for this machine is:"
psrinfo -v
echo

echo "Currently the total number of processes "
echo " running on this machine is \c"
ps -ef | tail +2 | wc -l      # wc -l gives number of lines
```

2. Change the permissions on `echoscript.sh` so you can execute the script as a command.

```
$ chmod 744 echoscript.sh
```

3. Execute the `echoscript.sh` script. If this script does not execute correctly, determine the problem, re-edit, and execute again. If debug options would help in fixing the error, add debugging statements.

```
$ ./echoscript.sh
```

Task Solutions

4. Copy the debugscript.sh script to debug1.sh. Read debug1.sh and note the different debug options entered in comment statements. This makes it easier to execute the script and test the option that gives you the most useful information.

```
$ cp debugscript.sh debug1.sh  
$ cat debug1.sh
```

5. Execute debug1.sh. Does it appear to execute correctly?

```
$ ./debug1.sh
```

Yes, it executes correctly.

6. Misspell one of the echo statements near the middle of the script. Re-execute the script, and record the error message given. Then correct the spelling.

```
$ ./debug1.sh  
Your terminal type is set to: dtterm  
  
Your login name is: user1  
. /debug1.sh: ehco: not found           <-- NOTE: ERROR MESSAGE
```

To list the contents in the /etc/security directory:

```
ls /etc/security
```

7. Change the statement in debug1.sh that prints out the value of the logname variable instead of LOGNAME. Therefore, change:

```
echo "Your login name is: $LOGNAME"
```

to:

```
echo "Your login name is: $logname"
```

8. Re-execute debug1.sh, and write down the output given for the modified line.

Your login name is:

9. Now re-execute the debug1.sh script after turning on each of the three different debug options one at a time. Record whether each option gives you any help in determining the problem.

```
set -v
```

Output:

```
echo "Your login name is: $logname"  
Your loging name is:
```

You can see the misspelled variable in the echo statement.

```
set -x
```

Output:

```
+ echo Your login name is:  
Your login name is:  
+ echo
```

You cannot see the name of the variable because it is was interpreted and replaced by its value (the default value is a null string).

```
set -f
```

Output:

```
Your login name is:
```

You cannot see the name of the variable because it is was interpreted and replaced by its value (the default value is a null string).

10. Which debug options gave you information to help point to the variable name problem?

The -v option was best because it showed the misspelling of the variable as part of the debug output.

11. Copy the debug1.sh script to debug2.sh.

```
$ cp debug1.sh debug2.sh
```

12. Modify the debug2.sh script to:

- Run the Korn shell interpreter rather than the Bourne shell

Change #!/bin/sh to #!/bin/ksh

- Turn on the verbose debug mode with the Korn shell-specific syntax (not set -v) just before the first echo statement

Add the statement set -o verbose before to the first echo statement.

Task Solutions

13. Execute the debug2.sh script, and compare the output to the output of debug1.sh executing with the -v option. Are the two script outputs the same?

In one window:

```
$ ./debug1.sh
```

In another window:

```
$ ./debug2.sh
```

The two scripts have the same output.

14. Write a template shell script that contains basic information that should be included for each script that you write. It should provide documentation, in the form of comments, for each script you write in the future. Later when creating a new script, copy the template, fill in the information, and proceed to enter the statements into the script.

The template should include:

- The shell interpreter
- The name of the script
- The author of the script
- The date the script was written
- A brief description of the purpose of the script

```
$ cat template.sh  
#!/bin/sh
```

```
# Script name: template  
# Author: Student name  
# Date written: xx/xx/xx  
#  
# Purpose: This script serves as a documentation template for future  
#           scripts.  
#
```

```
$ cat template.ksh  
#!/bin/ksh
```

```
# Script name: template  
# Author: Student name  
# Date written: xx/xx/xx  
#  
# Purpose: This script serves as a documentation template for future  
#           scripts.  
#
```

Module 3

The Shell Environment

Objectives

Upon completion of this module, you should be able to:

- Use Bourne and Korn shell variables
- Assign values to shell variables
- Display the value of shell variables
- Make variables available to subprocesses using the `export` command
- Display the value of environment variables
- Unset shell and environment variables
- Customize the user environment using the `.profile` file
- Perform arithmetic operations
- Create and use aliases
- Display aliases and the values assigned to them
- Define the built-in aliases
- Customize the Bourne and Korn shell environments
- Use the tilde expansion and command substitution features of the Korn shell

User Startup Scripts

User Startup Scripts

When a user logs in using `sh` or `ksh`, configuration scripts run in the following order:

1. `/etc/profile`
2. `$HOME/.profile`
3. `$HOME/.kshrc` (for Korn shell (`ksh`) users if the `ENV` variable is set)

The `/etc/profile` Script

- You set up `/etc/profile`.
- It's the same file for all users of a system.
- Generally, it contains only a few commands, such as setting a default umask and printing the message of the day.

The `$HOME/.profile` Script

`. $HOME/profile:`

- Is in a user home directory.
- Is run each time a user logs in.
- Doesn't exist by default.
- Generally, contains environment variables, such as a user's preferred editor, the default printer, and the application software location.

`$HOME/.profile` Script Example

The following example `.profile` file sets and then exports variables. The `MANPATH` variable is set to the directories in which the `man` utility should search for the online manual pages. The `ENV` variable is set to the full path name of the file that is read each time a new Korn shell is started. The `EDITOR` variable is set to the `vi` editor. Applications, such as `cron` and `edquota`, use this variable. Exporting the variables makes them part of the environment, which is inherited by all subshells and subprocesses.

Note – Variables are described in the “Shell Variables” on page 3-5.

```
$ cat /.profile
# This file initially did NOT exist for root
MANPATH=$MANPATH:/usr/share/man:/usr/dt/share/man:/usr/java1.2/man
ENV=$HOME/.kshrc
EDITOR=vi
export MANPATH ENV EDITOR
```

The ENV variable has special meaning only to the Korn shell.

Within a user's .profile file, often, there will be variables to support software, such as environment variables required by database software that specify the location of the software or the software libraries.

The following three lines are special because they allow a root user to change the shell from Bourne to Korn after the initial login. This benefits a root user who wants the default to be Bourne in case of a system catastrophe when only the root file system is available. If there is no failure, the shell then changes to Korn, in which the history, alias, or other features of the Korn shell are available. Therefore, refine the .profile file to test for the existence of /usr/bin/ksh, and only then execute these lines if the file exists.

```
SHELL=/usr/bin/ksh      # This variable sets the default shell
                         # for subshells
export SHELL
/usr/bin/ksh      # Invokes a Korn shell as a child of the login shell
```

The \$HOME/.kshrc Script

The Korn shell executes the \$HOME/.kshrc file if the ENV variable is set. This happens each time a Korn shell is spawned, either at the request of a user or when a shell spawns a Korn subshell. This file resides in the user's home directory. The file sets Korn shell specific parameters, variables, and aliases.

\$HOME/.kshrc Script Example

The following .kshrc script first sets the PS1 variable, which is the prompt. The exclamation point within the string expands to the current command number in the history file. Setting the trackall option causes the Korn shell to execute commands faster by creating aliases for commands for which it has to search. Aliases are established to save keystrokes.

Note – The set command and aliases are described in the “Automatic (Tracked) Aliases” on page 3-30.

User Startup Scripts

```
$ cat .kshrc
PS1="'hostname' ! $ "
set -o trackall
alias l='ls -laF'
alias ll='ls -aF'
alias hi='fc -l'
alias c=clear
```

Modifying a Configuration File

If you make changes to the `/etc/profile` or `$HOME/.profile` file, the shell does not read the changes until the next time you log in. To avoid having to re-log in, use the dot command to tell the shell to reread the `.profile` file:

```
$ . $HOME/.profile
```

By invoking a script with the dot command, you tell the shell (parent) to read the commands in the *script_name* file and execute them without spawning a child process. Any definitions in the script then become part of the present environment.

Most shell scripts execute in a private environment. Any variables they set are not available to other scripts. The shell (or script) that invokes a script is called a *parent*, and the script that is being invoked is called the *child*. Variables are not inherited unless they are exported with the `export` command.

Shell Variables

Shell variables are capitalized by convention. The shell maintains two lists of variables:

1. Those local to the current shell
2. Those global to all shells (environment variables).

Use the `set` and `env` commands to display local and environmental variables, respectively. The following is a partial output of the `set` and `env` statements. Many variables appear in both the local and environment variable list.

```
$ set
AB2_DEFAULTSERVER=http://docs.sun.com/
CUE_HOSTNAME=sunray10
DISPLAY=:46.0
DOMAIN=renegades.Central.Sun.COM
DOMAIN_COUNT=1
DTSOURCEPROFILE=true
DTUSERSESSION=milner-sunray10-46
DTXSERVERLOCATION=local
EDITOR=vi
ERRNO=25
FCEDIT=/bin/ed
LANG=C
LOGNAME=milner
LPDEST=hutchence
MAIL=/var/mail/milner
MAILCHECK=600
MANPATH=/usr/dt/man:/usr/man:/usr/openwin/share/man:/usr/man:/usr/openwin
/share/man:/usr/dt/man:/usr/dist/local/man/5.7
PRINTER=hutchence
PS1='$ '
PS2='> '
PS3='#? '
PS4='+ '
PWD=/home/milner/K-Shell-Course/InstructorGuide/Examples
SHELL=/bin/ksh
TZ=US/Mountain
...
<output truncated>
```

Shell Variables

```
$ env
DTSOURCEPROFILE=true
DOMAIN=renegades.Central.Sun.COM
DTUSERSESSION=milner-sunray10-46
EDITOR=vi
LOGNAME=milner
MAIL=/var/mail/milner
CUE_HOSTNAME=sunray10
PRINTER=hutchence
DISPLAY=:46.0
TERM=dtterm
TZ=US/Mountain
LPDEST=hutchence
DOMAIN_COUNT=1
...
<output truncated>
```

Variable names can contain uppercase or lowercase letters, digits, and underscores. Variable names cannot begin with a digit. Some of the boot scripts show mixed-case variable names. Be sure to use meaningful variable names.

Creating Variables in the Shell

To set a variable in the shell, use the syntax:

```
var=value
```

Note – Variables are case-sensitive so, after they are defined, they must later be spelled using exactly the same uppercase and lowercase letters. Variable names have a limit of 255 characters.

Do not place spaces around the = sign. If the value contains spaces or special characters, use single or double quotes; for example:

```
$ name="Susan B. Anthony"
```

To display the value of a shell variable, use the echo command and place a \$ immediately in front of the variable name.

When you have finished a variable, you can unset the value and release the resources with the unset command; for example:

```
$ MYNUM=21  
$ MACHTYPE=sparc  
$ echo $MYNUM  
21  
$ echo $MYNUM $MACHTYPE  
21 sparc  
$ unset MYNUM  
$ echo $MYNUM $MACHTYPE  
sparc
```

Creating Variables in the Shell

In the following example, notice how variable substitution happens before file name expansion:

```
# VAR="*"
# ls $VAR
hsperfdata_noaccess:
805

hsperfdata_root:
3101
# >-1
# ls $VAR
hsperfdata_noaccess:
total 64
-rw-----    1 noaccess noaccess   32768 Nov 13 15:43 805

hsperfdata_root:
total 64
-rw-----    1 root      root       32768 Nov 23 15:27 3101
```

The \$VAR variable is first resolved and then passed to the ls command.

Exporting Variables to Subshells

Variables created in a shell are not known outside that shell. These variables are *local* variables. They are “local to” (known only in) the current shell.

Subshells of a shell do not automatically inherit the variables of their parent shell. Only the variables that are *exported* are passed onto subshells of the parent shell. Variables that are exported are referred to as *environment* variables.

To export a variable (with its current value) to subshells of a shell, use:

```
export variable_name1 variable_name2 ...
```

Note – A subshell can change the value of a variable it inherits from its parent. The change does not affect the value of the variable in the parent shell. Thus, subshells cannot alter values of variables in the parent shell.

The following example creates two variables in the current shell, *x* and *name*. The *name* variable is then exported to be inherited by any subshells of the parent.

```
$ x=25
$ name="Marion Morrison"
$ echo $x $name
25 Marion Morrison
$ export name
```

A subshell is created by invoking a new Korn shell using the ksh command.

```
$ ksh
$ print $x
$ print $name
Marion Morrison
$ export name="Marlena Dietrich"
$ print $name
Marlena Dietrich
$ exit
```

Exporting Variables to Subshells

When you try to access the value of the variable `x` using the `echo` command, no output is shown because the child shell does not know the variable `x`. When you access the value of the variable `name`, the value of the variable is output because this variable was exported in the parent shell.

Changing the contents of `name` in the subshell has no effect on the value of the `name` variable in the parent.

```
$ echo $name  
Marion Morrison
```

Reserved Variables

The shell knows about and uses certain variables. Use care when modifying the values for these variables. Before modifying a value, know how the shell uses the variable and how you might effect your interaction with the shell.

To learn about variables not listed in this module, read the man pages for `sh` or `ksh`, depending on your choice for an interactive shell. Table 3-1 describes the shell variables.

Table 3-1 Shell Variables

Variable	Meaning
HOME	User's login directory path.
IFS	Internal field separators, which hold a characters used as inter-field separators. By default, this string includes a space, a tab, and a new line. Don't change this string.
LOGNAME	User's login name.
MAILCHECK	How often, in seconds, the <code>mail</code> daemon checks for mail.
OPTIND	<code>getopts</code> statement uses this variable during parsing of options on command line.
PATH	Search path for commands.
PS1	The prompt. It defaults to <code>\$</code> .
PS2	Prompt used when command line continues. Defaults to <code>></code> .
PS3	Prompt used within a <code>select</code> statement. Defaults to <code>#?</code> .
PS4	Prompt used by the shell debug utility. Defaults to <code>+</code> .
PWD	Present working directory.
SHELL	Shell defined for the user in the <code>passwd</code> file.
TERM	Terminal type, which is used by the <code>vi</code> editor and other commands.

Special Shell Variables

Special Shell Variables

Several shell variables are available to users. The shell sets the variable value at a process creation or termination.

Process Identification

The \$ shell variable is available to users. It contains the current process ID number. Use this variable to create file names that are unlikely to already exist.

```
$ echo $$  
8873  
$ sh  
$ echo $$  
17716  
$ exit  
$ echo $$  
8873
```

Exit Status

The *exit status* is the integer value of the last executed command (program, script, or shell statement). The ? variable provides the exit status of the most recent foreground process. It:

- Determines if the process ran successfully.
- Sets the ? variable to 0 if a process succeeded and to a nonzero value if it didn't succeed.

Think of this as equating success with zero errors occurring and equating failure with one or more errors occurring or the command not being able to do what was requested.

```
$ grep "root" /etc/passwd  
root:x:0:1:Super-User:/sbin/sh  
$ echo $?  
0  
  
$ grep "rot" /etc/passwd  
$ echo $?  
1
```

Background Process Identification

A shell allows you to run a command in the background using the & operator. To find the process identification number (PID) of the last background job started, echo the ! variable.

```
$ date &
1175
$ Mon Oct 12 06:23:11 IST 2009

$ echo $!
1175
```

Quoting Characters

Quoting Characters

With the exception of the letters and the digits, practically every key on the keyboard is a *metacharacter* in some context or other (including the space, the tab, and the newline). A metacharacter is a character that represents something other than its literal self.

When you want a metacharacter to be taken literally, you must tell the shell to leave it alone (rather than interpreting it) by using one of the three following mechanisms:

- Single quotes, which turn off the special meaning of all characters
- Double quotes, which turn off the special meaning of characters except \$, ', " , and \
- Backslash, which turns off the special meaning of the following character

A Pair of Single Quotes

A pair of single quotes, ' . . . ', turns off the special meaning of *all* characters enclosed by the single quotes, including the \ character.

Note – A single quote character is not allowed to appear within a pair of single quotes.

```
$ echo a           b  
a b  
  
$ echo 'a           b'  
a           b  
  
$ num=25  
  
$ echo 'The value of num is $num'  
The value of num is $num
```

A Pair of Double Quotes

A pair of double quotes (as in " . . . ") turns off the special meaning of all characters enclosed by the double quotes, except for the four characters \$, `, ", and \. Within double quotes, use the backslash to selectively turn off the special meaning of these four characters.

Note – A double quote character is not allowed to appear within a pair of double quotes unless it is preceded by a \.

```
$ num=25  
  
$ echo "The value of num is $num"  
The value of num is 25  
  
$ name=Roger  
  
$ echo "Hi $name, I'm glad to meet you!"  
Hi Roger, I'm glad to meet you!  
  
$ echo "Hey $name, the time is `date`"  
Hey Roger, the time is Fri Jul  9 16:52:50 PDT 1999
```

Backslash

If you need to have the special meaning of any character *turned off* and the character to have its literal value, precede it with the backslash (\) character, which is often called an *escape* character. Use the backslash to turn off the special meaning of any shell metacharacter.

```
$ name=jessica  
  
$ echo "Hello $name. \  
> Where are you going?"  
Hello jessica. Where are you going?
```

Include the backslash (\) character before \$name.

```
$ echo "Hello \$name. \  
> Where are you going?"  
Hello $name. Where are you going?
```

Quoting Characters

The eval Command

The eval command reads its arguments as input to the shell and executes the resulting command(s). This is usually used to execute commands generated as the result of command or variable substitution.

```
$ eval whence -v ps  
ps is /usr/bin/ps
```

Shell Command Substitution

Command substitution allows the output of one command to be used as an argument to another command:

- The Bourne shell uses `` (back quotes).
- The Korn shell supports the older Bourne shell syntax.
- The Korn shell uses the \$(*command*) syntax.

```
$ ksh
```

```
$ date +%H:%M  
12:50
```

```
$ print "The time is now $(date +%H:%M) ."  
The time is now 12:50.
```

```
$ print "There are $(ps -ef | tail +2 | wc -l) users on  
the system"  
There are      12 users on the system
```

```
$ ls -l $(which passwd)  
-r-sr-sr-x  1 root      sys          22644 Aug  7 16:47  
/usr/bin/passwd
```

```
$ cat currentinfo.ksh  
#!/bin/ksh  
# outputs the day, time, & current month.  
day=$(date +%D)  
time=$(date +%T)  
print "Today is $day."  
print  
print "The time is $time."  
print  
print "This month's calendar:"  
cal
```

Shell Command Substitution

```
$ ksh currentinfo.ksh
Today is 10/12/09.
```

The time is 06:33:19.

This month's calendar:

October 2009

S	M	Tu	W	Th	F	S
					1	2
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Korn Shell Tilde Expansion

Use the tilde character (~) at the beginning of words as follows:

~/	Full path name of current user's home directory
~ <i>username</i>	Full path name of <i>username</i> 's home directory
~+	Full path name of a working directory
~-	Previous working directory
-	Previous working directory

Examples

The cd command accepts a hyphen to change the working directory to the previous working directory.

```
$ pwd  
/users/milner/examples
```

```
$ cd ../course_notes/mod1
```

```
$ pwd  
/users/milner/course_notes/mod1
```

```
$ cd -  
/users/milner/examples
```

Arithmetic Operations on Bourne Shell Variables

Arithmetic Operations on Bourne Shell Variables

The Bourne shell assigns string values to variables only in the assignment statement. The Bourne shell has no built-in ability to perform arithmetic. The external statement, `expr`, treats the variables as numbers and performs the arithmetic operation.

Note – The `expr` statement is space-sensitive and expects each operand and operator to be separated by white spaces.

The `expr` statement recognizes the operators shown in Table 3-2.

Table 3-2 The `expr` Operators

Operator	Operation	Example
+	Addition	<code>num2='expr "\$num1" + 25'</code>
-	Subtraction	<code>num3='expr "\$num1" - "\$num2"'</code>
*	Multiplication	<code>num3='expr "\$num1" * "\$num2"'</code>
/	Division	<code>num4='expr "\$num2" / "\$num1"'</code>
%	Integer remainder	<code>num5='expr "\$num1" % 3'</code>

Note – Precede a multiplication operator with a backslash because it is a shell metacharacter.

The Bourne shell has no built-in arithmetic capability.

```
$ num1=5
$ echo $num1
5

$ num2=$num1+10
$ echo $num2
5+10
```

The `expr` statement must have white space around operators and operands. The `expr` statement automatically sends the value to standard output:

```
$ echo $num1  
5  
$ expr $num1 + 7  
12  
  
$ expr $num1+7  
5+7  
  
$ echo $num1  
5
```

The following shows the need for properly quoting the multiplication operator.

```
$ expr $num1 * 2  
expr: syntax error  
  
$ expr "$num1 * 2"  
5 * 2  
  
$ expr $num1 \* 2  
10
```

Arithmetic Operations on Bourne Shell Variables

When performing arithmetic, an operation result is stored in a variable. This is accomplished in the Bourne shell by placing the `expr` statement on the right side of a variable assignment statement:

```
$ echo $num1  
5  
  
$ num2='expr $num1 + 25'  
$ echo $num2  
30  
  
$ num3='expr $num1 + $num2'  
$ echo $num3  
35  
  
$ num4='expr $num1 \* $num2'  
$ echo $num4  
150  
  
$ num5='expr $num2 / $num1'  
$ echo $num5  
6  
  
$ num6='expr $num1 % 3'  
$ echo $num6  
2
```

Arithmetic Operations on Korn Shell Variables

Place an integer expression in two pairs of parentheses ((. . .)) to invoke an Arithmetic evaluation. For readability, place a space before and after any arithmetic operator within the double pair of parentheses. Table 3-3 lists the standard arithmetic operations.

Table 3-3 Arithmetic Operations on Integers

Operator	Operation	Example	Result
+	Addition	((x = 24 + 25))	49
-	Subtraction	((x = 100 - 25))	75
*	Multiplication	((x = 4 * 5))	20
/	Division	((x = 10 / 3))	3
%	Modulo (remainder)	((x = 10 % 3))	1

All arithmetic operations are performed using integer arithmetic, which can cause minor surprises:

```
$ x=15.38
$ y=15.72

$ ((z = x + y))
$ echo $z
30
```

Note – Don't put a \$ in front of a variable reference within a calculation. Expansion is automatically performed on all variable references made within the double parentheses.

Everything is integer arithmetic. Note what happens when division is performed in a calculation. For example, when computing the percentage of the eligible voters who actually voted, the following statement would most likely assign 0 to prct:

```
((prct = voted / eligible_voters * 100))
```

The following algebraically equivalent statement is the correct way to compute the percentage of eligible voters in the Korn shell:

```
((prct = voted * 100 / eligible_voters))
```

Arithmetic Precedence

- Expressions (operations) within parentheses are evaluated first. Use a single pair of parentheses around an expression to force it to be evaluated first.
- Multiplication (*) and division (%) and (/) have greater precedence than addition (+) and subtraction (-).
- Arithmetic expressions are evaluated from left to right.
- When there is the slightest doubt, use parentheses to force the evaluation order.

Bit-wise Operations

The six bit-wise operators are listed in Table 3-4.

Table 3-4 Bit-wise Operations

Operator	Operation	Example	Result
#	Base	2#1101010 or 16#6A	10#106
<<	Shift bits left	((x = 2#11 << 3))	2#11000
>>	Shift bits right	((x = 2#1001 >> 2))	2#10
&	Bit-wise AND	((x = 2#101 & 2#110))	2#100
	Bit-wise OR	((x = 2#101 2#110))	2#111
^	Bit-wise exclusive OR	((x = 2#101 ^ 2#110))	2#11

The # operator designates the base of the value that follows it. For example, 2#101 means that the value 101 is a base 2 (binary) value (2#101 would be 5 in the decimal base).

Note – If you are not familiar with bit-wise operations, examine the values in binary (as is shown here) rather than in decimal.

The `<<` operator performs a binary *shift left* by as many bits as are indicated by the number that follows the operator. The expression `2#10 << 1` yields the value `2#100`. The expression `2#10100 << 2` yields the value `2#1010000`.

Initial value			0	1	0	0
<code><< 2</code>	0	1	0	0		
Result	0	1	0	0	0	0

Vacated positions are padded with 0 or 1 based on whether the number is positive or negative, respectively.

The `>>` operator performs a binary *shift right* by as many bits as are indicated by the number that follows the operator. The expression `2#10 >> 1` yields the value `2#1`. The expression `2#10100 >> 2` yields the value `2#101`.

The `&` operator *ANDs* two binary numbers together. This means that the AND operation is performed on the corresponding digit of 0 or 1 in each number, resulting in a 1 if both digits are 1; otherwise, the result is 0.

Initial value	1	1	0	0
<code>&</code>	1	0	1	0
Result	1	0	0	0

The `|` operator *ORs* two binary numbers together. This means that the OR operation is performed on the corresponding digit of 0 or 1 in each number, resulting in a 1 if either digit is a 1; otherwise, the result is 0.

Initial value	1	1	0	0
<code> </code>	1	0	1	0
Result	1	1	1	0

Arithmetic Operations on Korn Shell Variables

The `^` operator performs an *exclusive OR* on two binary numbers together. This means that an exclusive OR is performed on the corresponding digit of 0 or 1 in each number, resulting in 1 if only one of the digits is a 1. If both digits are 0 or both are 1, then the result is 0.

Initial value	1	1	0	0
<code>^</code>	1	0	1	0
Result	0	1	1	0

The Korn Shell let Statement

The `let` statement is an alternative to the `((...))` statement. Type the arithmetic formula with no spaces unless the formula is enclosed in double-quote ("") characters. Assign a numeric value, using the formula syntax, without using the `let` statement. It's common to use the Korn shell `((...))` syntax instead of the `let` statement:

```
$ let a=1
$ let b=3
$ let c=4

$ echo $a
1

$ let "a = a + 1"

$ echo $a
2

$ let a=b+c

$ echo $a
7

$ ((a = b + c))

$ echo $a
7
$ a=b+c
$ echo $a
b+c
```

Script Math

Script Math

The following script:

1. Assigns the value 99 to the variable `y`.
2. Computes the cube of the number entered, the quotient of the number entered divided by 4, and the remainder of the number entered divided by 4.
3. Prints the results with an appropriate message.
4. Computes the number input using the quotient, the divisor 4, and the remainder.
5. The result is multiplied by 2 and saved in the variable `z`.
6. A message showing the `z` value prints.

```
$ cat math.ksh
#!/bin/ksh

# Script name: math.ksh

# This script finds the cube of a number, and the
# quotient and remainder of the number divided by 4.

y=99

(( cube = y * y * y ))
(( quotient = y / 4 ))
(( rmdr = y % 4 ))

print "The cube of $y is $cube."
print "The quotient of $y divided by 4 is $quotient."
print "The remainder of $y divided by 4 is $rmdr."

# Notice the use of parenthesis to
# control the order of evaluating.
(( z = 2 * (quotient * 4 + rmdr) ))
print "Two times $y is $z.

$ ./math.ksh
The cube of 99 is 970299.
The quotient of 99 divided by 4 is 24.
The remainder of 99 divided by 4 is 3.
Two times 99 is 198.
```

Korn Shell Aliases

An alias is a name that you give to a command. You can create an alias in the Korn shell. Aliases are not available in the Bourne shell. Commands, scripts, or programs that require a user to perform a lot of typing to execute them are good alias candidates. Other reasons for using aliases may include:

- Several versions of a command exist on a system, and you want to use a particular one by default. You create an alias that lists the entire path name to that command:
`alias mycommand=/fullpathname/cmd`
- You frequently and incorrectly spell a command, so you make an alias of the incorrect spelling:
`alias mroe=more`
- You set up default options for a command:
`alias dk='df -k'`

Removing Aliases

If you don't want to use an alias, delete it using the `unalias` command. The syntax is:

```
unalias aliasname
```

Alias Inheritance

Aliases are not inherited by subshells. Placing the aliases in the `$HOME/.kshrc` file and having the value of `ENV` set to `$HOME/.kshrc` allows new Korn shells to know about the aliases.

Note – Aliases created on the command line are known only in the current shell.

Korn Shell Aliases

```
$ alias lpgut='lp -d guttenberg'  
$ alias lpgut  
lpgut='lp -d guttenberg'  
$ ksh  
$ alias lpgut  
lpgut alias not found  
$ exit  
$ alias lpgut  
lpgut='lp -d guttenberg'  
$ unalias lpgut  
$ alias lpgut  
lpgut alias not found
```

Automatic (Tracked) Aliases

After it is enabled, the Korn shell automatically makes an alias to the full path name of the command whenever you use a UNIX command. If you use the command later, the alias is used instead of searching \$PATH. To turn on this Korn shell feature, use the following syntax:

```
$ set -o trackall  
$ date  
Thur Mar 14 12:10:00 EST 1991  
$ alias  
date=/usr/bin/date  
  
<output truncated>
```

The aliases are listed alphabetically, so you might need to search through the list to find the alias you want.

If a Bourne shell user types “alias,” the user executes /bin/alias, which lists the Korn shell-predefined aliases. This happens because /bin/alias is a shell script that:

- Executes in a Korn subshell; therefore, it knows all predefined ksh aliases
- Sets a variable named cmd to "alias"
- Executes the value of the variable cmd, passing to it all arguments passed to /bin/alias

Built-in Aliases

You can use the functions and autoload aliases with function names.

Use the `r` alias to re-execute a previously given command. If the `r` alias is followed by a number, it re-executes the command from the history file that has that number as its command number. If it is followed by a string, it re-executes the most recent command that begins with the string given.

Table 3-5 lists some Korn shell predefined aliases.

Table 3-5 Built-in Aliases

Command	Meaning
<code>functions='typeset -f'</code>	Use this alias to list names of functions, with their definitions, that are known in the current shell.
<code>history='fc -l'</code>	Displays a list of the 16 most recently given commands. Each command is preceded by its command number.
<code>integer='typeset -i'</code>	Use the <code>integer</code> alias to declare a variable as an integer data type.
<code>nohup='nohup '</code>	Use to run a command that is immune to the hangup (HUP) and terminate (TERM) signals. If standard output is a terminal, the output is redirected to the <code>nohup.out</code> file. Standard error is redirected to follow standard output.
<code>r='fc -e -'</code>	Use this alias to re-execute a previous command.
<code>suspend='kill -STOP \$\$'</code>	Issues the <code>kill</code> command with the STOP (SIGSTOP) signal, Signal 23, on the current shell (\$\$). The shell process is placed on the jobs list and might be restarted later with the <code>fg</code> command.

Shell Parse Order

Parsing controls how a shell processes a command line before it executes a command. A shell has a predefined order in which it interprets or evaluates different tokens (metacharacters, spaces, keywords, and so on) on the command line. Based on rules built into it, a shell interprets tokens in a specific order (the parse order) and then executes the command. The following list shows the parse order for the Bourne and Korn shells. The Korn shell has extra features, ignore those if you are reading this in preparation to use the Bourne shell.

1. Read the command.
2. Evaluate keywords.

For example: `if`, `else`, `fi`, `do`, `done`, and `time`. You can verify these with "`whence -v`".

3. Evaluate the alias.
4. Evaluate built-in commands.

For example: `alias`, `bg`, `cd`, `echo`, and `let`. You can verify these with "`whence -v`".

5. Evaluate functions.
6. Perform tilde expansion.
7. Perform command substitution.
8. Perform arithmetic expression substitution.
9. Evaluate metacharacters for expansion of file names.
10. Look up the command or script.
11. Execute the command.

Special Characters Precedence

The special characters precedence is as follows:

1. ' (outside single quotes)
2. '' (single quotes)
3. \
4. () {}
5. ; &
6. && ||
7. |
8. Variable substitution ` ` (back quotes)
9. " " (double quotes)
10. (()) (calculations)
11. * ? @() *() +() !()

Exercise: Using the Shell Environment

Exercise: Using the Shell Environment

Exercise objective – Set and unset variables.

Preparation

Refer to the lecture notes as necessary to perform the following tasks.

In the first part of the exercise, you create and manipulate variables. When you are asked to display a list of variables, guess whether your variable will be in the specific shell or environmental list.

Execute the smc & command and bring up the Solaris Management Console (SMC) GUI prior to creating new users.

Edit the /etc/skel/local.profile file to add a variable, because this file is copied to \$HOME/.profile for each new Bourne or Korn shell user you create. You may also choose to modify your /.profile to invoke a Korn shell.

Tasks

1. Create the var1 shell variable, and initialize it to twenty.
2. Create the var2 shell variable, and initialize it to 21.
3. Create the var3 shell variable, and initialize it to temp.
4. Display the values of all three variables.
5. Display the list of local shell variables, and determine if either var1, var2, or var3 is in the listing.
6. Display the list of environmental variables, and determine if either var1, var2, or var3 is in the listing.
7. Use the export command to make var1 and var3 environmental variables.
8. Display the list of environmental variables, and determine if either var1 or var3 is in the listing.
9. Remove the var3 variable with the unset command.
10. Display the list of local shell variables, and determine if either var1, var2, or var3 is in the listing.

11. Display the list of environmental variables, and determine if either var1, var2, or var3 is in the listing.
12. Create a new *terminal* window by executing the dtterm command. If necessary, you can execute the command using the absolute path name, /usr/dt/bin/dtterm.
13. In the new terminal window, display the list of local shell variables, and determine if either var1 or var2 is in the listing.
14. In the new terminal window, display the list of environmental variables, and determine if either var1 or var2 is in the listing.
15. For the benefit of all your Bourne and Korn shell users, add the following to the /etc/profile file:

Make sure you are logged in as the root user to perform the following steps.

```
ENV=$HOME/.kshrc  
EDITOR=vi  
export ENV EDITOR
```

16. Create or modify the /.profile file to make the Korn shell your login shell.
17. For root, set the prompt to the *hostname* followed by a #.
18. Create a /.kshrc file, and add some Korn shell commands that you would find useful; for example:

```
set -o vi  
alias ll='ls -l'  
alias pss='ps -ef | sort +1n | more'
```

Exercise: Using the Shell Environment

19. Edit the `/etc/skel/local.profile` file to make the following change. This is the file that is copied a new Bourne or Korn shell user's home directory. This file is renamed to `.profile`.
- Append the `/usr/dt/bin` directory onto the `PATH` variable.
 - Create a Bourne and a Korn shell user using the SMC GUI by using the information in the following table:

User Name:	user1	user2
User ID:	<next available>	<next available>
Primary Group:	10	10
Login Shell:	Bourne	Korn
Password:	<code>!cangetin</code>	<code>!cangetin</code>
Home Dir Path:	<code>/export/home/user1</code>	<code>/export/home/user2</code>

- Log out and log in as either user, and verify that the changes made to `/etc/skel/local.profile` were copied into the user's `$HOME/.profile`.
- Log out and log back in as `root`.
- Write a script in which you assign an integer number to a variable. The output from the script should be the number and the value of that number multiplied by itself.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Task Solutions

1. Create the var1 shell variable, and initialize it to twenty.

```
$ var1=twenty
```

2. Create the var2 shell variable, and initialize it to 21.

```
$ var2=21
```

3. Create the var3 shell variable, and initialize it to temp.

```
$ var3=temp
```

4. Display the value of all three variables.

```
$ echo $var1 $var2 $var3
```

```
twenty 21 temp
```

5. Display the list of local shell variables, and determine if either var1, var2, or var3 is in the listing.

```
$ set | grep var
```

```
var1=twenty
```

```
var2=21
```

```
var3=temp
```

6. Display the list of environmental variables, and determine if either var1, var2, or var3 is in the listing.

```
$ env | grep var
```

7. Use the export command to make var1 and var3 environmental variables.

```
$ export var1 var3
```

8. Display the list of environmental variables, and determine if either var1 or var3 is in the listing.

```
$ env | grep var
```

```
var1=twenty
```

```
var3=temp
```

9. Remove the var3 variable with the unset command.

```
$ unset var3
```

10. Display the list of local shell variables, and determine if either var1, var2, or var3 is in the listing.

```
$ set | grep var
```

```
var1=twenty
```

```
var2=21
```

11. Display the list of environmental variables, and determine if either var1, var2, or var3 is in the listing.

```
$ env | grep var  
var1=twenty
```

12. Create a new *terminal* window by executing the dtterm command. If necessary, you can execute the command using the absolute path name, /usr/dt/bin/dtterm.

```
$ dtterm
```

13. In the new terminal window, display the list of local shell variables, and determine if either var1 or var2 is in the listing.

```
$ set | grep var  
var1=twenty
```

14. In the new terminal window, display the list of environmental variables, and determine if either var1 or var2 is in the listing.

```
$ env | grep var  
var1=twenty
```

15. For the benefit of all your Bourne and Korn shell users, add the following to the /etc/profile file:

Make sure you are logged in as the root user to perform the following steps.

```
ENV=$HOME/.kshrc  
EDITOR=vi  
export ENV EDITOR
```

16. Create or modify the /.profile file to make the Korn shell your login shell.

Use vi and add the following to the /.profile file:

```
SHELL=/usr/bin/ksh  
export SHELL  
/usr/bin/ksh
```

17. For root, set the prompt to the *hostname* followed by a #.

Use vi to modify the /.profile file:

```
PS1="'`uname -n`# "  
export PS1
```

18. Create a /.kshrc file. Add some Korn shell commands that you would find useful; for example:

```
set -o vi  
alias ll='ls -l'  
alias pss='ps -ef | sort +1n | more'
```

Task Solutions

19. Edit the `/etc/skel/local.profile` file to make the following change. This is the file that is copied a new Bourne or Korn shell user's home directory. This file is renamed to `.profile`.
- Append the `/usr/dt/bin` directory onto the `PATH` variable.
`PATH=/usr/bin:/usr/ucb:/etc:/usr/dt/bin:.`
 - Create a Bourne and a Korn shell user using the SMC GUI by using the information in the following table.

User Name:	user1	user2
User ID:	<next available>	<next available>
Primary Group:	10	10
Login Shell:	Bourne	Korn
Password:	<code>!cangetin</code>	<code>!cangetin</code>
Home Dir Path:	<code>/export/home/user1</code>	<code>/export/home/user2</code>

20. Log out and log in as either user, and verify that the changes made to `/etc/skel/local.profile` were copied into the user's `$HOME/.profile`.
21. Log out and log back in as `root`.
22. Write a script in which you assign an integer number to a variable. The output from the script should be the number and the value of that number multiplied by itself.

```
#!/bin/ksh

# Purpose: To take a value from the parameter list and
#           multiple it by itself.
#
# Name: multiply
integer num
num=8
print
print "The number is: $num"
(( result = num * num ))
print "The number multiplied by itself is: $result"

print
```

Module 4

Regular Expressions and the grep Command

Objectives

Upon completion of this module, you should be able to:

- Use and describe regular expressions
- Describe the grep command
- Use the grep command to find patterns in a file
- Use regular expression characters with the grep command

The grep Command

The grep Command

Like most UNIX commands, grep is a mnemonic. The grep mnemonic is derived from ex editor commands. The meaning is globally (*g*) search for a regular expression (*re*) and print (*p*) the results (*grep*).

The grep utility searches text files for a specified pattern and prints all lines that contain that pattern. If no files are specified, grep assumes it will receive text from standard input.

Consider the following scenario. A user comes to you and says that the `msxyz` program on her machine has locked up her machine. She cannot get the program to halt. It is your job to find the process and stop it.

The `ps -ef` command gives you a long list of running processes. The output is probably too long to find this user's process. You want a single line or a specific process list.

```
$ ps -ef | grep 'msxyz'
```

The previous grep command takes the `ps -ef` command output as its input. The grep command performs a search for the string `msxyz` and prints the results. This gives you specific lines to review. The grep command syntax is:

```
grep [options] pattern_file
```

You have multiple terminal and console windows open. To see those specific processes, execute a `ps` command and search for the `dtterm` command.

```
$ ps -e | grep 'dtterm'
 352 ??          0:00 dtterm
 353 ??          0:13 dtterm
 354 ??          0:11 dtterm
 1766 pts/5      0:00 dtterm
```

The grep Options

Table 4-1 shows grep command options. These options modify grep behavior.

Table 4-1 The grep Options

Option	Meaning
-i	Makes the command case insensitive
-c	Prints the count of lines that match
-l	Prints the names of the files in which the lines match
-v	Prints the lines that do not contain the search pattern
-n	Prints the line numbers

The -i option specifies that grep ignore the case of the letters in the search pattern. In the following example, grep does a case-insensitive search for the pattern the.

```
$ grep -i 'the' /etc/default/login
# Set the TZ environment variable of the shell.
# ULIMIT sets the file size limit for the login. Units are disk blocks.
# The default of zero means no limit.
# ALTSHELL determines if the SHELL environment variable should be set
# PATH sets the initial shell PATH variable
<output truncated>
```

Compare this to the output of the following command, which prints only the lines of text that match the pattern The.

```
$ grep 'The' /etc/default/login
# The default of zero means no limit.
# bad password is provided. The range is limited from
# The SYSLOG_FAILED_LOGINS variable is used to determine how many failed
```

The -c option counts the number of lines that match the pattern. It then prints the count and not the actual lines that matched the pattern.

```
$ grep -ci 'the' /etc/default/login
19

$ grep -c 'The' /etc/default/login
3
```

The grep Options

Use the -l option to:

- Search for a string in many files.
- Have the output list the only files in which the string is found.

The -l option is often useful when you want to feed the output of grep to another utility to process a list.

```
# grep -l 'grep' /etc/init.d/*
/etc/init.d/apache
/etc/init.d/cachefs.daemon
/etc/init.d/dhcp
/etc/init.d/dodatadm.udapl
/etc/init.d/dtlogin
/etc/init.d/imq
/etc/init.d/init.wbem
/etc/init.d/ncakmod
/etc/init.d/swupboots
```

To find a search pattern in a large file, print the line number before each match using the -n option. This is useful when you are editing files.

```
# grep -n 'user' /etc/passwd
18:user1:x:100:10::/export/home/user1:/bin/sh
19:user2:x:101:10::/export/home/user2:/bin/ksh
```

The -v option prints lines that do not contain the search pattern.

```
# grep -v 'root' /etc/group
staff::10:
sysadmin::14:
smmsp::25:
gdm::50:
webservd::80:
postgres::90:
nobody::60001:
noaccess::60002:
nogroup::65534:
```

Regular Expression Metacharacters

A regular expression (RE) is a character pattern that matches the same characters in a search. Regular expressions:

- Allow you to specify patterns to search in text.
- Provide a powerful way to search files for specific pattern occurrences.
- Give additional meaning to patterns (as shown in Table 4-2).

When you use regular expression characters with the `grep` command, enter quotes around the pattern. Some regular expression characters used by `grep` are also metacharacters to one or more shells, and a shell might use a metacharacter as a file name metacharacter. Use single (') quotes. Doing this hides more metacharacters from a shell.

Table 4-2 `grep` command Metacharacters

Metacharacter	Function
\	Escapes the special meaning of an RE character
^	Matches the beginning of the line
\$	Matches the end of the line
\<	Matches the beginning of word anchor
\>	Matches the end of word anchor
[]	Matches any one character from the specified set
[-]	Matches any one character in the specified range
*	Matches zero or more of the preceding character
.	Matches any single character
\{ \}	Specifies the minimum and maximum number of matches for a regular expression

Regular Expressions

Regular Expressions

Using a regular expression, you can search the current process table (and header) for any process that contains a capital letter. Do this by using the following range as the pattern to the grep command:

```
# ps -ef | grep '[A-Z]'  
UID PID PPID C STIME TTY TIME CMD  
root 647 1 0 06:14:45 ? 0:00 /usr/lib/dmi/snmpXdmid -s sls-s10-  
host  
noaccess 797 1 0 06:15:03 ? 1:34 /usr/java/bin/java -server -Xmx128m -  
XX:+UseParallelGC -XX:ParallelGCThreads=4  
root 708 704 4 06:14:50 ? 5:22 /usr/X11/bin/Xorg :0 -depth 24 -  
nobanner -auth /var/dt/A:0-9Aaayb  
root 813 739 0 06:15:16 ? 0:00 /bin/ksh /usr/dt/bin/Xsession  
root 905 903 0 06:15:27 pts/2 0:00 -sh -c unset DT; DISPLAY=:0;  
/usr/dt/bin/dtsession_res -merge  
root 1045 1 1 06:15:51 ? 1:10 /usr/lib/mixer_applet2 --oaf-  
activate-iid=OAFIID:GNOME_MixerApplet_Factory --oa  
root 1050 1 0 06:15:52 ? 0:01 /usr/lib/notification-area-applet --  
oaf-activate-iid=OAFIID:GNOME_NotificationA  
root 1440 1284 0 08:20:35 pts/4 0:00 grep [A-Z]  
  
<output truncated>
```

If you are only interested in current processes that contain the capital letter A in the line, limit the pattern to specify that character.

```
# ps -ef | grep 'A'  
root 708 704 7 06:14:50 ? 5:26 /usr/X11/bin/Xorg :0 -depth 24 -  
nobanner -auth /var/dt/A:0-9Aaayb  
root 905 903 0 06:15:27 pts/2 0:00 -sh -c unset DT; DISPLAY=:0;  
/usr/dt/bin/dtsession_res -merge  
root 1442 1284 0 08:21:17 pts/4 0:00 grep A  
  
<output truncated>
```

Escaping a Regular Expression

To escape a regular expression, use a \ (backslash) followed by a single character matches that character. Thus, a \\$ matches a dollar sign and a \. matches a period. Doing this divests a metacharacter of its special meaning. The following example shows the \$ as a regular expression character that matches the end of a line.

```
# grep '$' /etc/init.d/nfs.server
#!/sbin/sh
#
# Copyright 2004 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
#ident  "@(#)nfs.server 1.43      04/07/26 SMI"

# This service is managed by smf(5). Thus, this script provides
# compatibility with previously documented init.d script behaviour.

case "$1" in
'start')
    svcadm enable -t network/nfs/server
    ;;
'stop')
    svcadm disable -t network/nfs/server
    ;;
*)
    echo "Usage: $0 { start | stop }"
    exit 1
    ;;
esac
```

The output contains all the lines from the script because the \$ matches the end-of-line character for each line in the script. Verify this with the wc command.

```
$ grep '$' /etc/init.d/nfs.server | wc -l
24

$ wc -l /etc/init.d/nfs.server
24 /etc/init.d/nfs.server
```

Escaping a Regular Expression

To display only the lines from the `nfs.server` boot script that contain the literal character `$`, hide its special meaning by preceding the character with the `\` regular expression character.

```
$ grep '\$' /etc/init.d/nfs.server
case "$1" in
    echo "Usage: $0 { start | stop }"
$ grep '\$' /etc/init.d/nfs.server | wc -l
2
```

Line Anchors

An anchor is a symbol that matches a character position on a line. The ^ and \$ anchors match text patterns relative to the beginning ^ or ending \$ of a line of text. For example, the following command finds all lines that contain the pattern root in the /etc/group file.

```
$ grep 'root' /etc/group
root::0:
other::1:root
bin::2:root,daemon
sys::3:root,bin,adm
adm::4:root,daemon
uucp::5:root

<output truncated>
```

If you intend to display only the one entry for the root group in the /etc/group file, then the pattern must specify that the line begins with the pattern (given the syntax of the file).

```
$ grep '^root' /etc/group
root::0:
```

The regular expression character allows you to anchor the pattern match to the beginning of the line.

Similarly the \$ regular expression character allows you to anchor the pattern match to the end of the line. Lines print only if the specified pattern represents the characters preceding the end-of-line character.

```
$ grep 'mount$' /etc/vfstab
#device      device      mount      FS      fsck      mount      mount
```

Word Anchors

Word Anchors

A backslash used with an angle bracket is a word anchor. The less-than bracket (<) marks the beginning of a word. Any text that follows this bracket is matched only when it occurs at the beginning of a word. The greater-than bracket (>) marks the end of a word. Text that precedes this bracket is matched only when it occurs at the end of a word. Words are delimited by spaces, tabs, beginnings of line, ends of line, and punctuation.

For example, if you wanted to print the group file entry for the uucp group, issuing the grep command without regular expression characters gives you the uucp and nuucp group entries. Using the following command, however, should give only the single group entry for uucp.

```
$ grep '\<uucp\>' /etc/group
uucp::5:root
```

Use both word anchors at the same time to ensure your pattern is a complete word by itself, rather than a sub-string of another word. Note the output if you search for the pattern user in the /etc/passwd file.

```
$ grep 'user' /etc/passwd
user:x:100:1::/home/user:/bin/sh
user2:x:101:1::/home/user2:/bin/sh
user3:x:102:1::/home/user3:/bin/sh
```

The preceding output includes lines with user as a sub-string of words, such as user2, and user3. If you were searching for the specific user named user, you should use both word anchors (or the -w option).

```
$ grep '\<user\>' /etc/passwd
user:x:100:1::/home/user:/bin/sh
```

Character Classes

A string enclosed in square brackets specifies a character class. Any single character in the string is matched. For example, the grep '[abc]' frisbee command displays every line that contains an a, b, or c in the frisbee file.

The following command prints the lines from the /etc/group file that contain either the letter i or the letter u.

```
$ grep '[iu]' /etc/group
bin:::2:root,daemon
sys:::3:root,bin,adm
uucp:::5:root
mail:::6:root
nuucp:::9:root
sysadmin:::14:
nogroup:::65534:
```

You might also specify a range of characters, which results in printing lines that contain at least one of the specified characters in the range.

```
$ grep '[u-y]' /etc/group
sys:::3:root,bin,adm
uucp:::5:root
tty:::7:root,adm
nuucp:::9:root
sysadmin:::14:
webservd:::80:
nobody:::60001:
nogroup:::65534:
```

The following examples show the contents of the teams file and how character classes can be used to find the word the or The in any line in the teams file.

```
$ cat teams
Team one consists of
Tom
Team two consists of
Fred
The teams are chosen randomly.
Tea for two and Dom
Tea for two and Tom

$ grep '\<[Tt]he\>' teams
The teams are chosen randomly.
```

Character Match

Single Character Match

The . regular expression character matches any one character except the newline character.

The following command looks for all lines containing c, followed by any three characters, followed by h.

```
$ grep 'c...h' /usr/dict/words
```

The following command looks for all lines that do not have a character before the c; that is, the c is the first character, followed by any three characters, followed by h.

```
$ grep '^c...h' /usr/dict/words
```

The following command looks for all lines that do not have a character before the c, followed by any three characters, followed by h, which is the end of the word; that is five-letter words that begin with c and end with h.

```
$ grep '^c...h$' /usr/dict/words
```

Character Match by Specifying a Range

The \{ and \} expressions allow you to specify the minimum and maximum number of matches for a regular expression.

The following example shows the use of this expression.

```
$ cat test
root
rooot
rooooot
roooooot

$ grep 'ro\{3\}t' test
rooot

$ grep 'ro\{2,4\}t' test
root
rooot
rooooot
```

Closure (*)

The *, when used in a regular expression, is termed a closure. The closure symbol matches the preceding symbol or character zero or more times.

```
$ grep 'Team*' teams
Team one consists of
Team two consists of
Tea for two and Dom
Tea for two and Tom
```

For example, to find all lines that contain a word beginning with T and a word ending with m, use the following command:

```
$ grep '\<T.*m\>' teams
Team one consists of
Tom
Team two consists of
Tea for two and Dom
Tea for two and Tom
```

An asterisk (*) has special meaning only when it follows another character. If it is the first character in a regular expression or if it is by itself, it has no special meaning. The following example searches for lines containing a literal asterisk within the file called teams.

```
$ grep '*' teams
```

The asterisk has another meaning outside of the regular expression. The following command searches all files in the current directory for the string abc.

```
$ grep 'abc' *
data1:abcd
```

In the above example, the * is a shell metacharacter rather than a grep metacharacter.

Closure (*)

The egrep Command

The egrep command (or the extended grep command) searches a file for a pattern using full regular expressions. For example:

```
# grep "two | team" teams
# egrep "two | team" teams
Team two consists of
The teams are chosen randomly.
Tea for two and Dom
Tea for two and Tom
```

Exercise: Using Regular Expressions and the grep Command

Exercise objective – Write grep commands to search for the specified strings within text files.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change the directory to mod4/lab before beginning the exercise.

You might want to look at the /var/sadm/install/contents file to re-acquaint yourself with the information contained in each entry, specifically the first and last fields.

Tasks

1. Print all the entries from /etc/passwd for users who have ksh as their login shell.
2. Print all the entries from /etc/passwd for users who have sh as their login shell.
3. Print the lines from /etc/group that contain the pattern root preceded by the line number.
4. Print the lines from /etc/group that do not contain the pattern root.
5. Determine the package name of the Java™ runtime environment installed on your machine. (Hint: Search the /var/sadm/install/contents file for jre/bin/java.)
6. Print all the file and directory entries from the /var/sadm/install/contents file that are part of or dependent on the Java runtime environment package you identified in Step 5. This step could return hundreds of lines.
7. Print the number of files and directories that are part of the Java runtime environment package; use the /var/sadm/install/contents file.

Exercise: Using Regular Expressions and the grep Command

8. Use two grep commands on a single command line to print only the files that were installed in the java/bin directory that are part of the Java runtime environment package; use the /var/sadm/install/contents file.
9. List all entries of local disk devices from /etc/vfstab.
10. Place the two grep commands from Step 8 in a shell script. Execute the script.
11. Start a script named adduser by copying one of the two template files in the lab directory. Add a single grep command into the script that allows you to determine if the user name (stored in the name variable) already exists in the mypasswd file. Test all values for name that appear in the template file. (You will build on this script in later modules as you encounter new concepts and constructs.)

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Task Solutions

1. Print all the entries from /etc/passwd for users who have ksh as their login shell.

```
$ grep 'ksh$' /etc/passwd
```

2. Print all the entries from /etc/passwd for users who have sh as their login shell.

```
$ grep '/sh$' /etc/passwd
```

or

```
$ grep '\<sh\>$' /etc/passwd
```

3. Print the lines from /etc/group that contain the pattern root preceded by the line number.

```
$ grep -n 'root' /etc/group
```

4. Print the lines from /etc/group that do not contain the pattern root.

```
$ grep -v 'root' /etc/group
```

5. Determine the package name of the Java runtime environment installed on your machine. (Hint: Search the /var/sadm/install/contents file for jre/bin/java.)

```
$ grep 'jre/bin/java' /var/sadm/install/contents
```

6. Print all the file and directory entries from the /var/sadm/install/contents file that are part of or dependent on the Java runtime environment package you identified in Step 5. This step could return hundreds of lines.

```
$ grep 'SUNWj5rt' /var/sadm/install/contents
```

7. Print the number of files and directories that are part of the Java runtime environment package; use the /var/sadm/install/contents file.

```
$ grep -c 'SUNWj5rt' /var/sadm/install/contents
```

8. Use two grep commands on a single command line to print only the files that were installed in the java/bin directory that are part of the Java runtime environment package; use the /var/sadm/install/contents file.

```
$ grep 'SUNWj5rt' /var/sadm/install/contents | grep 'java/bin'
```

9. List all entries of local disk devices from /etc/vfstab.

```
$ grep '^/dev' /etc/vfstab
```

10. Place the two grep commands from Step 8 in a shell script, and then execute the script.

```
$ more twogreps.sh  
#!/bin/sh  
  
echo "Here are the JRE files installed in java/bin"  
grep 'SUNWj3rt' /var/sadm/install/contents | grep 'java/bin'  
  
echo  
echo "Now a listing of local disk devices from /etc/vfstab"  
grep '^/dev' /etc/vfstab
```

11. Start a script named adduser by copying one of the two template files in the lab directory. Add a single grep command into the script that allows you to determine if the user name (stored in the name variable) already exists in the mypasswd file. Test all values for name that appear in the template file. (You will build on this script in later modules as you encounter new concepts and constructs.)

```
$ cp adduser_sh.template adduser.sh (or cp adduser_ksh.template  
adduser.ksh)
```

```
$ cat adduser.sh (or cat adduser.ksh)  
#!/bin/sh
```

```
# Purpose: To write a script to add user to the system.  
# Name: adduser.sh
```

```
### Set a shell variable to test the grep command.  
name=nuucp      # Should match a single entry  
#name=uucp      # Should match a single entry  
#name=root      # Should match a single entry  
#name=rot       # Should not match any entry  
  
grep "^\$name:" mypasswd  
  
# End of lab
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 5

Using the sed Editor

Objectives

Upon completion of this module, you should be able to:

- Use the `sed` editor to perform noninteractive editing tasks
- Use regular expression characters with the `sed` command

Introduction to the `sed` Editor

The term *sed* stands for *stream editor*.

A stream is a source or destination for bytes, which means `sed` can take its input from standard in, apply the requested edits on the stream, and automatically put the results to standard out. The `sed` syntax allows for an input file to be specified on the command line. The syntax does not allow an output-file specification; this can be accomplished through output redirection.

As shown in Figure 5-1, the `sed` editor edits a file by reading the file line-by-line into a pattern buffer, modifying the line, and then outputting the buffer to standard out (`stdout`), which can be redirected to another file. The original file is not modified.

The holding buffer is used for advanced operations. It is limited to a copy, append, compare, or retrieval command. Use the holding buffer to find duplicate lines in a sorted input file or to concatenate multiple lines together for future output.

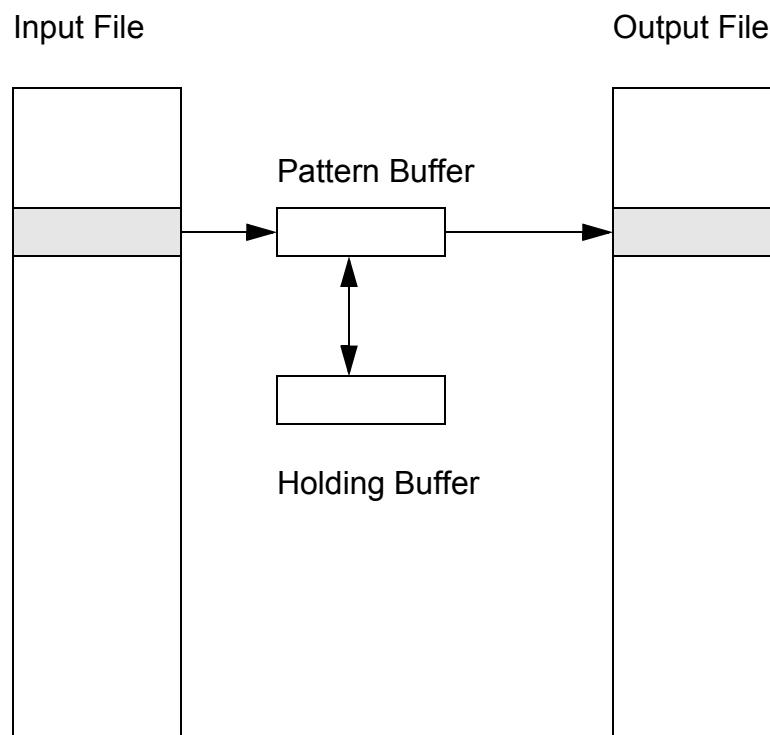


Figure 5-1 How `sed` Works

Command Format

The following shows the syntax for the `sed` command:

```
sed [options] ' [addresses] action [args]' files  
[ > outfile]
```

You do not need to interact with the `sed` editor while it is running; therefore, it has also been termed a batch editor. This is in contrast to such editors as `vi` and `ed`, which are interactive. Because `sed` does not require interaction, you can place `sed` commands in a script. You can call the script file and run it against the data file to perform repetitive editing operations.

The `sed` editor is capable of performing text-pattern substitutions and text-pattern deletions using regular expression syntax. These are the same regular expression characters used by `grep`.

The `sed` command offers capabilities that are an extension of interactive text editing. If you need to search and replace text strings in a large number of files, `sed` is most useful.

Editing Commands

Editing Commands

The `sed` editor uses a editing commands (Table 5-1) that are similar to those you would use for `vi` and `ed`.

Table 5-1 The `sed` Commands

Command	Function
d	Deletes line(s)
p	Prints line(s)
r	Reads a file
s	Substitutes one string for another
w	Writes to a file

The `sed` command has two options (Table 5-2).

Table 5-2 The `sed` Options

Option	Function
-n	Suppresses the default output
-f	Reads <code>sed</code> commands from a script file

Addressing

The sed editor processes all lines of the input file unless you specify an address. This address can be a range of line numbers, a regular expression, or a combination of both.

When you specify a line number or a range of line numbers, the line numbers represent lines from the input file against which the specified changes are applied. All other lines from the input file are displayed, unchanged, to standard out.

Similarly when a pattern, which may contain a regular expression, is used to select lines, only lines containing the pattern are edited. All other lines from the input file are displayed, unchanged, to standard out.

If two patterns are specified, this creates a range, beginning with the first line of the file containing the first pattern, including all subsequent lines of the file up to, and including, the one containing the second pattern or the end of the file.

A \$ (dollar character) represents the last line of a file when used in address. The \$ character represents the end of line (EOL) when used in a regex.

Using sed to Print Text

Using sed to Print Text

The following examples execute sed commands against the `data.file` file.

```
$ cat data.file
```

northwest	NW	Joel Craig	3.0 .98 3	4
western	WE	Sharon Kelly	5.3 .97 5	23
southwest	SW	Chris Foster	2.7 .8 2	18
southern	SO	May Chin	5.1 .95 4	15
southeast	SE	Derek Johnson	5.0 .70 4	17
eastern	EA	Susan Beal	4.4 .8 5	20
northeast	NE	TJ Nichols	5.1 .94 3	13
north	NO	Val Shultz	4.5 .89 5	9
central	CT	Sheri Watson	5.7 .94 5	13

The following example shows the use of the `p` (print) command, which prints a range of lines to `stdout`. The range is specified by a starting address followed by a comma and then the ending address.

```
$ sed '3,5p' data.file
```

northwest	NW	Joel Craig	3.0 .98 3	4
western	WE	Sharon Kelly	5.3 .97 5	23
southwest	SW	Chris Foster	2.7 .8 2	18
southwest	SW	Chris Foster	2.7 .8 2	18
southern	SO	May Chin	5.1 .95 4	15
southern	SO	May Chin	5.1 .95 4	15
southeast	SE	Derek Johnson	5.0 .70 4	17
southeast	SE	Derek Johnson	5.0 .70 4	17
eastern	EA	Susan Beal	4.4 .8 5	20
northeast	NE	TJ Nichols	5.1 .94 3	13
north	NO	Val Shultz	4.5 .89 5	9
central	CT	Sheri Watson	5.7 .94 5	13

The default output of `sed` is each line that it reads. To suppress the default output, use the `-n` option.

```
$ sed -n '3,5p' data.file
```

southwest	SW	Chris Foster	2.7 .8 2	18
southern	SO	May Chin	5.1 .95 4	15
southeast	SE	Derek Johnson	5.0 .70 4	17

The following command prints all lines with the pattern west in it. Use the forward slash (/) to delimit the regular expression.

```
$ sed -n '/west/p' data.file
```

northwest	NW	Joel Craig	3.0 .98 3	4
western	WE	Sharon Kelly	5.3 .97 5	23
southwest	SW	Chris Foster	2.7 .8 2	18

The following command prints the first line containing the pattern west, up to and including the next line containing the pattern southern.

```
$ sed -n '/west/,/southern/p' data.file
```

northwest	NW	Joel Craig	3.0 .98 3	4
western	WE	Sharon Kelly	5.3 .97 5	23
southwest	SW	Chris Foster	2.7 .8 2	18
southern	SO	May Chin	5.1 .95 4	15

The following command prints the first line containing the pattern Chris, up through the last line of the file.

```
$ sed -n '/Chris/,$/p' data.file
```

southwest	SW	Chris Foster	2.7 .8 2	18
southern	SO	May Chin	5.1 .95 4	15
southeast	SE	Derek Johnson	5.0 .70 4	17
eastern	EA	Susan Beal	4.4 .8 5	20
northeast	NE	TJ Nichols	5.1 .94 3	13
north	NO	Val Shultz	4.5 .89 5	9
central	CT	Sheri Watson	5.7 .94 5	13

The pattern might contain the regular expression characters used by grep. The following example prints all lines that begin with an s and end with a 5.

```
$ sed -n '/^s.*5$/p' data.file
```

southern	SO	May Chin	5.1 .95 4	15
----------	----	----------	-----------	----

Using sed to Substitute Text

Using sed to Substitute Text

The **sed s** command allows for a search and substitution operation to occur on the text. The command uses a pattern search and a literal string replacement. The replacement string characters are taken literally without metacharacter expansion.

```
$ sed 's/3/X/' data.file
```

northwest	NW	Joel Craig	X.0 .98	3	4
western	WE	Sharon Kelly	5.X .97	5	23
southwest	SW	Chris Foster	2.7 .8	2	18
southern	SO	May Chin	5.1 .95	4	15
southeast	SE	Derek Johnson	5.0 .70	4	17
eastern	EA	Susan Beal	4.4 .8	5	20
northeast	NE	TJ Nichols	5.1 .94	X	13
north	NO	Val Shultz	4.5 .89	5	9
central	CT	Sheri Watson	5.7 .94	5	1X

The **sed** command checks each line of the file and substitutes the first occurrence of the old string with the new string. Subsequent occurrences of the old string within the same line are left unchanged.

The following example shows the **g** (global) command with the **s** (search and substitute) command, and it replaces all occurrences of the old string with the new string.

```
$ sed 's/3/X/g' data.file
```

northwest	NW	Joel Craig	X.0 .98	X	4
western	WE	Sharon Kelly	5.X .97	5	2X
southwest	SW	Chris Foster	2.7 .8	2	18
southern	SO	May Chin	5.1 .95	4	15
southeast	SE	Derek Johnson	5.0 .70	4	17
eastern	EA	Susan Beal	4.4 .8	5	20
northeast	NE	TJ Nichols	5.1 .94	X	1X
north	NO	Val Shultz	4.5 .89	5	9
central	CT	Sheri Watson	5.7 .94	5	1X

Occasionally with a search and substitute, the old string will be part of the new replacement string, which you can accomplish by placing an & (ampersand) in the replacement string. The location of the & determines the location of the old string in the replacement string.

The objective of the following examples is to write a command that searches for all lines that end with a single digit in the last field and replace the single digit with the single-digit number plus the string Single_Digit.

To properly identify the lines with single-digit numbers in the last field, consider the following sed command. Tabs separate the fields with each line.

```
$ sed -n '/ [0-9]$/p' data.file
northwest      NW      Joel Craig      3.0 .98 3      4
north          NO      Val Shultz     4.5 .89 5      9
```

The following command searches for all lines that end with a single digit in the last field and replaces the single digit with the single-digit number plus the string Single Digit.

```
$ sed 's/[0-9]$/& Single Digit/' data.file
northwest      NW      Joel Craig      3.0 .98 3      4 Single Digit
western        WE      Sharon Kelly    5.3 .97 5      23
southwest      SW      Chris Foster   2.7 .8 2       18
southern        SO      May Chin      5.1 .95 4       15
southeast       SE      Derek Johnson 5.0 .70 4       17
eastern         EA      Susan Beal    4.4 .8 5       20
northeast       NE      TJ Nichols   5.1 .94 3       13
north          NO      Val Shultz    4.5 .89 5      9 Single Digit
central         CT      Sheri Watson 5.7 .94 5       13
```

Reading From a File for New Text

Reading From a File for New Text

Instead of inserting a line of text once, you might want to repeat the procedure several times, either in the same file or across multiple files. The **r** (read) command specifies a file name, and the contents of the file are inserted into the output after the lines specified by the address. The address may be a line number or pattern combination.

```
$ cat northmesg
*** The northern regions are the newest in the company ***
*** and the people are still being trained. *****

$ sed '/north/r northmesg' data.file
northwest      NW      Joel Craig      3.0 .98 3      4
*** The northern regions are the newest in the company ***
*** and the people are still being trained. *****
western        WE      Sharon Kelly     5.3 .97 5      23
southwest       SW      Chris Foster    2.7 .8   2      18
southern         SO      May Chin       5.1 .95 4      15
southeast        SE      Derek Johnson  5.0 .70 4      17
eastern          EA      Susan Beal     4.4 .8   5      20
northeast        NE      TJ Nichols    5.1 .94 3      13
*** The northern regions are the newest in the company ***
*** and the people are still being trained. *****
north           NO      Val Shultz    4.5 .89 5      9
*** The northern regions are the newest in the company ***
*** and the people are still being trained. *****
central          CT      Sheri Watson   5.7 .94 5      13
```

Note – The space following the **r** (read) command is required. You must also immediately follow the file name with a closing quote.

Using sed to Delete Text

The following command deletes Lines 4 through 8 from the output.

```
$ sed '4,8d' data.file
```

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
central	CT	Sheri Watson	5.7	.94	5	13

The following command deletes any line containing the pattern west.

```
$ sed '/west/d' data.file
```

southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	5.0	.70	4	17
eastern	EA	Susan Beal	4.4	.8	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

The following command deletes any line beginning with the pattern west.

```
$ sed '/^west/d' data.file
```

northwest	NW	Joel Craig	3.0	.98	3	4
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	5.0	.70	4	17
eastern	EA	Susan Beal	4.4	.8	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

The following command deletes the range of lines beginning with the first line containing the pattern south, up through the next line of the file containing north.

```
$ sed '/south/,/north/d' data.file
```

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

Reading sed Commands From a File

Reading sed Commands From a File

Multiple sed commands can be put in a file and executed using the -f option. When you place the commands in a file you:

- Do not use quotes around the action and address
- Ensure that there is no trailing white space at the end of each line

```
$ cat script1.sed
1,4d
s/north/North/
s/^east/East/
```

The commands in the script:

- Delete the first four lines
- Replace any instances of north with North
- Change any line that starts with east to East

These commands can then be applied to the data.file file with the following command:

```
$ sed -f script1.sed data.file
southeast      SE      Derek Johnson   5.0 .70 4      17
Eastern        EA      Susan Beal     4.4 .8   5      20
Northeast      NE      TJ Nichols    5.1 .94 3      13
North          NO      Val Shultz    4.5 .89 5      9
central        CT      Sheri Watson  5.7 .94 5      13
```

Using sed to Write Output Files

The w (write) command allows a specific sed command to write the output to a named file. Different sed commands can write to different files; for example:

```
$ cat script5.sed
/north/w northregions
s/9[0-9]/& Great job!/w topperformers
```

The previous script of sed commands writes to two different output files. The first file, northregions, contains all lines with the pattern north. The second file, topperformers, has all lines containing a 9 followed by another digit. This output contains the matched number, followed by the string Great job!.

```
$ sed -n -f script5.sed data.file

$ more northregions topperformers
::::::::::::::::::
northregions
::::::::::::::::::
northwest      NW      Joel Craig      3.0 .98 3      4
northeast      NE      TJ Nichols      5.1 .94 3      13
north          NO      Val Shultz      4.5 .89 5      9
::::::::::::::::::
topperformers
::::::::::::::::::
northwest      NW      Joel Craig      3.0 .98 Great job!      3      4
western        WE      Sharon Kelly      5.3 .97 Great job!      5      23
southern        SO      May Chin       5.1 .95 Great job!      4      15
northeast      NE      TJ Nichols      5.1 .94 Great job!      3      13
central         CT      Sheri Watson     5.7 .94 Great job!      5      13
```

Note – The space following the w (write) command is required. Also, follow the file name immediately by the end of line.

Exercise: Using the sed Editor

Exercise: Using the sed Editor

Exercise objective – Write `sed` commands to delete, write, search, and substitute, as well as print lines from the input file to standard output or to a specified file.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change to the `mod5/lab` directory before beginning the exercise.

Use the `passwd2` file provided in the `mod5/lab` directory for the duration of this lab.

Tasks

1. Print to standard out only the lines that end in the pattern `sh`.
2. Print to standard out only the lines that have the letter `o` as the second character.
3. Delete the first four characters of each line.
4. Delete each line that contains the pattern `uu`.
5. Print to standard out only the lines containing the pattern `oo`.
6. Create a file of `sed` commands to perform the previous two tasks, and then do the following.
 - a. Add another command in the file to cause the `/etc/hosts` file to be displayed after the `passwd2` file in the output.
 - b. Remember to use the `-n` option.
7. The input to the `sed` command is a long listing of the `/` (root) directory. Send this to the `sed` command through a `|` (pipe).
8. Create a file of `sed` commands to do the following:
 - a. If the first letter on the line is an `l`, print the entire line appended with
`**** SYM LINK`
 - b. If the first letter on the line is a `-`, print the entire line appended with
`**** PLAIN FILE`.

9. Suppress default printing and create a sed script named `notrailingspaces.sed` to perform the following on the `hostwithspaces` input file:

- a. Delete trailing spaces at the end of the line.
- b. Write all lines to a `hostsnospaces` output file.

Hint: You can use the `vi :set list` feature to view trailing spaces.

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

- Print to standard out only the lines that end in the pattern sh.

```
$ sed -n '/sh$/p' passwd2
```

- Print to standard out only the lines which have the letter o as the second character.

```
$ sed -n '/^.o/p' passwd2
```

- Delete the first four characters of each line.

```
$ sed 's/^....//' passwd2
```

- Delete each line that contains the pattern uu.

```
$ sed '/uu/d' passwd2
```

- Print to standard out only the lines containing the pattern oo.

```
$ sed -n '/oo/p' passwd2
```

- Create a file of sed commands to preform the previous two tasks, and then do the following:

- In addition to these, add another command in the file to cause the /etc/hosts file to be displayed after the passwd2 file in the output.

- Remember to use the -n option.

```
$ cat labscript1.sed
```

```
/uu/d
```

```
/oo/p
```

```
$r /etc/hosts
```

To execute this sed script, type the following:

```
$ sed -n -f labscript1.sed passwd2
```

- The input to the sed command is a long listing of the / (root) directory. Send this to the sed command through a | (pipe).

- Create a file of sed commands to do the following:

- If the first letter on the line is an l, print the entire line appended with **** SYM LINK.

- If the first letter on the line is a -, print the entire line appended with **** PLAIN FILE.

```
$ more labscript2.sed
```

```
s/^l.*/& **** SYM LINK/
```

```
s/^-.*/& **** PLAIN FILE/
```

To perform Step 7and Step 8 to execute this script, type the following:

```
$ ls -l / | sed -f labscript2.sed
```

Task Solutions

9. Suppress default printing, and create a sed script named `notailingspaces.sed` to perform the following on the `hostwithspaces` input file:

- a. Delete trailing spaces at the end of the line.

```
s/ *$/ /
```

- b. Write all lines to a `hostsnoespaces` output file.

```
w hostsnoespaces
```

Hint: You can use the `vi :set list` feature to view trailing spaces.

```
$ cat notailingspaces.sed
```

```
s/ *$/ /
```

```
w hostsnoespaces
```

```
$ sed -nf notailingspaces.sed hostwithspaces
```

Module 6

The nawk Programming Language

Objectives

Upon completion of this module, you should be able to:

- Use nawk commands from the command line
- Write simple nawk programs to generate data reports from text files
- Write simple nawk programs to generate numeric and text reports from text files

Introduction to the nawk Programming Language

The awk programming language grew out of the recognition that many data processing problems are specialized applications of the concept of filtering, where the data is structured into records to which transformations are repetitively applied. The awk programming language is a record-oriented language that is named for its authors Aho, Weinberger, and Kernighan of AT&T Bell Labs.

“New awk” (nawk), added functionality to the awk programming language. This module describes the nawk programming language.

Unlike sed, nawk looks at data by records and fields. By default, records are delimited by newline characters, and the fields within them are delimited by spaces or tabs, but these can be set to the delimiters that are built-in to your data, such as colons or commas.

Applications written in the nawk programming language provide the following capabilities:

- Filtering
- Numerical processing on rows and columns of data
- Text processing to perform repetitive editing tasks
- Report generation

Using the nawk programming language, you can develop programs within a script. Many programming concepts, such as conditionals, looping, variables, and functions, are included in the nawk programming language. This module focuses on the basic concepts of nawk, not including those programming concepts.

Format of the nawk Command

The nawk command has the following format:

```
nawk 'statement' input.file
```

The *statement* in enclosed is single quotes and might be in one of three forms:

- *pattern { ACTION }*
The action is taken on those records that match the pattern.
- *pattern*
All records that match the pattern are printed.
- *{ ACTION }*
The action is taken on all records in the input file.

Executing nawk Scripts

Execute nawk scripts using this format:

```
nawk -f scriptfile input.file
```

You can combine repetitive nawk commands into nawk scripts that consist of one or more lines of the form:

```
pattern { ACTION }
```

where *pattern* is commonly a RE enclosed in slashes (/RE/) and *ACTION* is one or more statements of the nawk language.

Using nawk to Print Selected Fields

Using nawk to Print Selected Fields

The print statement outputs data from the file.

When nawk reads a record, it divides the record into fields based on the FS (input field separator) variable. This variable is predefined in nawk to be one or more spaces or tabs.

The variables \$1, \$2, \$3 hold the values of the first, second, and third fields. The variable \$0 holds the value of the entire line.

In the following example, Field 2 (office), Field 3 (first name), and Field 4 (last name) are printed.

```
$ cat data.file
northwest      NW      Joel Craig      3.0 .98 3      4
western        WE      Sharon Kelly     5.3 .97 5      23
southwest      SW      Chris Foster     2.7 .8 2      18
southern        SO      May Chin        5.1 .95 4      15
southeast       SE      Derek Johnson   5.0 .70 4      17
eastern         EA      Susan Beal       4.4 .8 5      20
northeast       NE      TJ Nichols      5.1 .94 3      13
north          NO      Val Shultz      4.5 .89 5      9
central         CT      Sheri Watson    5.7 .94 5      13

$ nawk '{ print $3, $4, $2 }' data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT
```

Formatting With print Statements

By adding tabs or other text inside double quotes, you can format the output neatly.

In nawk, anything in double quotes is a string constant. You can use string constants in many places in nawk—in print statements and as values to be assigned to variables among other things.

Some special formatting characters use letters, such as \t for tab. You can also specify an octal value, such as \011 for tab. Some of the formats you can use are shown in Table 6-1.

Table 6-1 Formatting Characters

Characters	Meaning
\t	Tab
\n	Newline
\007	Bell
\011	Tab
\012	Newline
\042	"
\044	\$
\045	%

If the fields in the nawk print statement are separated by commas (,), then the fields are separated by a space when they are printed. The comma is not a required part of the syntax. In a print statement, the comma actually represents the value of the nawk variable OFS (output field separator), which is described in the “Output Field Separator” on page 6-15. The default value of the OFS variable is a single space.

Formatting With print Statements

The following example adds a single tab character between Field 4 and Field 2.

```
$ awk '{ print $3, $4 "\t" $2 }' data.file
Joel Craig      NW
Sharon Kelly    WE
Chris Foster    SW
May Chin        SO
Derek Johnson   SE
Susan Beal       EA
TJ Nichols      NE
Val Shultz      NO
Sheri Watson    CT
```

Notice that it was not necessary to use the comma before or after the tab (\t) in the previous print statement. The tab forces the output that follows to begin at the next tab position.

Using Regular Expressions

Regular expression metacharacters can be used in the pattern.

In the following example, nawk searches for the pattern east, and prints all lines containing that pattern.

```
$ nawk '/east/' data.file
southeast      SE      Derek Johnson  5.0 .70 4      17
eastern        EA      Susan Beal     4.4 .8   5      20
northeast      NE      TJ Nichols    5.1 .94 3      13
```

The following example prints only Fields 1, 5, and 4 from lines containing the pattern east.

```
$ nawk '/east/ { print $1, $5, $4 }' data.file
southeast 5.0 Johnson
eastern 4.4 Beal
northeast 5.1 Nichols
```

The following example prints Fields 1 and 5, and then a tab before Field 4, for all lines containing the pattern east.

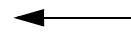
```
$ nawk '/east/ { print $1, $5 "\t" $4 }' data.file
southeast 5.0      Johnson
eastern 4.4       Beal
northeast 5.1      Nichols
```

The string can contain regular expression characters. In the following example, the pattern east must be at the beginning of the line.

```
$ nawk '/^east/' data.file
eastern        EA      Susan Beal     4.4 .8   5      20
```

The . specifies *any single character*. In this example, any character followed by a 9 would be a match.

```
$ nawk '/.9/' data.file
northwest     NW      Joel Craig    3.0 .98 3      4
western       WE      Sharon Kelly  5.3 .97 5      23
southern      SO      May Chin     5.1 .95 4      15
northeast     NE      TJ Nichols   5.1 .94 3      13
north         NO      Val Shultz   4.5 .89 5      9
central       CT      Sheri Watson 5.7 .94 5      13
```



Using Regular Expressions

Thus, the arrow points to a line of output you might not want if you are trying to identify lines that contain .9. To take away the special meaning of a regular expression character, precede it with a backslash (\).

```
$ awk '/\.9/' data.file
```

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southern	SO	May Chin	5.1	.95	4	15
northeast	NE	TJ Nichols	5.1	.94	3	13
central	CT	Sheri Watson	5.7	.94	5	13

The BEGIN and END Special Patterns

There are two special patterns that are not used to match text in a file. The BEGIN pattern (all letters must be uppercase) indicates an action that occurs before any of the input lines are read. It is commonly used for printing a heading or title line for the report before any data is processed and to assign values to built-in variables.

The END pattern (all uppercase) indicates an action occurs after all input records have been read and fully processed. It is commonly used to print summary statements or numeric totals.

BEGIN and END statements can each occur multiple times in any awk program (and in any order). If there are multiple occurrences of either pattern, they are executed in the order in which they are found in the file.

The following example has a BEGIN statement to add a header to the output.

```
$ awk 'BEGIN { print "Eastern Regions\n" }; /east/ { print $5, $4 }'  
data.file  
Eastern Regions
```

```
5.0 Johnson  
4.4 Beal  
5.1 Nichol
```

Although you can use multiple lines for the awk command, the beginning brace of the action for the BEGIN and END patterns must appear on the same line as the keyword BEGIN or END.

The correct use is:

```
$ awk 'BEGIN {  
> print "Eastern Regions\n"}; /east/ {print $5, $4}' data.file  
Eastern Regions  
  
5.0 Johnson  
4.4 Beal  
5.1 Nichols
```

The BEGIN and END Special Patterns

An example of incorrect use is:

```
$ awk 'BEGIN  
> { print "Eastern Regions\n" }; /east/ { print $5, $4 }' data.file  
nawk: syntax error at source line 2  
    context is  
        BEGIN >>>  
        <<<  
nawk: bailing out at source line 2
```

The END pattern allows the action to occur at the end of the input file.

```
$ awk 'BEGIN { print "Eastern Regions\n"}; /east/ {print $5, $4}  
> END {print "Eastern Region Monthly Report"}' data.file  
Eastern Regions  
  
5.0 Johnson  
4.4 Beal  
5.1 Nichols  
Eastern Region Monthly Report
```

Using nawk Scripts

A nawk script is a collection of nawk statements (patterns and actions) stored in a text file. A nawk script reduces the chance for errors because the commands are stored in a file and are read from the file each time they are needed.

Give the script file a descriptive name. To instruct nawk to read the script file, use the command:

```
nawk -f script_file data_file

$ cat report.nawk
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}

$ nawk -f report.nawk data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```

Using a nawk script makes it easy to make changes or additions. In the following example, a second BEGIN statement is added to print an overall heading for the report. Remember, the BEGIN statements are executed in order.

```
$ cat report2.nawk
BEGIN {print "*** Acme Enterprises ***"}
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}

$ nawk -f report2.nawk data.file
*** Acme Enterprises ***
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```

Using Built-in Variables

Using Built-in Variables

As `nawk` is processing the input file, it uses several variables. You can provide a value to some of these variables, while other variables are set by `nawk` and cannot be changed. Table 6-2 lists some of the built-in variables.

Table 6-2 Built-In `nawk` Variables

Name	Default Value	Description
FS	Space or tab	The input field separator
OFS	Space	The output field separator
NR		The number of records from the beginning of the first input file

Working With Variables

A variable value can be a number, a string, or a set of values in an array. The awk programming language uses several built-in variables. To assign a value to a variable, use the format:

```
variablename = value
```

Input Field Separator

The default input field separator (FS) is white space, which can be either a space or a tab. Frequently, other characters can separate the input, such as a colon or comma. You can set the input field separator variable with the -F option or set the value with an assignment. The following two examples both set the input field separator to a colon:

```
nawk -F: 'statement' filename  
nawk 'BEGIN { FS=":" } ; statement' filename
```

When using the -F option or the FS variable, you can specify more than one field separator either by placing the value in square brackets (creating a character class), or by separating the values with a | (OR statement) within double quotes.

```
nawk -F"[ :]" 'statement' filename  
nawk -F" |:" 'statement' filename  
nawk 'BEGIN { FS="[ :]" } next_statement' filename  
nawk 'BEGIN { FS=" |:" } next_statement' filename
```

Working With Variables

For example, set the default input field separator if you want to process the /etc/group file, which has fields separated by a colon. In this case, you set the FS variable before processing the first record of the file.

```
$ awk 'BEGIN { FS=":" } { print $1, $3 }' /etc/group
root 0
other 1
bin 2
sys 3
adm 4
uucp 5
mail 6
tty 7
lp 8
nuucp 9
staff 10
daemon 12
sysadmin 14
nobody 60001
noaccess 60002
nogroup 65534
```

You could save the previous command into a awk script.

```
$ cat report3.awk
BEGIN { FS=":" }
{ print $1, $3 }

$ awk -f report3.awk /etc/group
root 0
other 1
bin 2
sys 3
adm 4
uucp 5
mail 6
tty 7
lp 8
nuucp 9
staff 10
daemon 12
sysadmin 14
nobody 60001
noaccess 60002
nogroup 65534
```

Output Field Separator

The default output field separator is a space. In the print statement, a comma specifies using the output field separator. If you omit the comma, the fields run together. You can also specify a field separator directly in the print statement. Compare the following three lines:

```
$ awk '{ print $3 $4 $2 }' data.file  
$ awk '{ print $3, $4, $2 }' data.file  
$ awk '{ print $3, $4 "\t" $2 }' data.file
```

To set the output field separator, place the assignment within a BEGIN statement.

```
$ awk 'BEGIN { OFS="\t" } ; { print $3, $4, $2 }' data.file
```

Joel	Craig	NW
Sharon	Kelly	WE
Chris	Foster	SW
May	Chin	SO
Derek	Johnson	SE
Susan	Beal	EA
TJ	Nichols	NE
Val	Shultz	NO
Sheri	Watson	CT

Working With Variables

Number of Records

The number of records (NR) variable counts the number of input lines read from the beginning of the first input file. The variable's value updates each time another input line is read.

Inside the BEGIN pattern the value of NR is zero. Inside the END pattern the value of NR is the number of the last record processed.

```
$ more report4.nawk
{ print $3, $4, $2 }
END { print "The number of employee records is " NR }

$ nawk -f report4.nawk data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT
The number of employee records is 9
```

Exercise: Using `nawk` and Regular Expressions

Exercise objective – Write `nawk` commands to process text files and generate specified output or simple reports.

Preparation

Refer to the lecture notes as necessary to correctly compose the `nawk` commands.

Change the directory to `mod6/lab` before beginning the exercise.

For the multi-statement `nawk` commands, you might want to execute the individual statements on the system to ensure each gives the required output. After these individual statements prove correct, combine them into a multi-statement `nawk` command.

Tasks

Use the file named `data.txt` for these exercises.

1. For every line of the file, print the string `Name:`, followed by the person's first then last name, followed by a tab, followed by the string `Region:`, and finally followed by the abbreviation for the region.
2. For every line that matches the pattern of a W followed or preceded by another capital letter, print Fields 2, 3, 4, and 1.
3. For the output of Step 2, add a leading title to the report called `Developers in the Western Regions`.
4. For the output of Step 2, add a trailing footer to the report called `End of Report on the Western Regions`.
5. For every line of the file, print a line number, followed by a colon, followed by a tab, and finally the entire record. (Hint: Check Table 6-2 on page Module 6-12.)
6. Set the output field separator to a tab, and make the initial heading for this report `***** Regional Report *****`. For every line of the file, print a line number, followed by the last name of the person, and finally the region name.

Exercise Solution

Exercise Solution

Use the file named `data.file` for these exercises.

1. For every line of the file, print the string `Name:`, followed by the person's first then last name, followed by a tab, followed by the string `Region:`, and finally followed by the abbreviation for the region.

```
$ awk '{ print "Name:", $3, $4, "\t" "Region:", $2 }' data.file
```

2. For every line that matches the pattern of a `W` followed or preceded by another capital letter, print Fields 2, 3, 4, and 1.

```
$ awk '/[A-Z]W/ { print $2, $3, $4, $1 }
> /W[A-Z]/ { print $2, $3, $4, $1 }' data.file
```

3. For the output of Step 2, add a leading title to the report called `Developers in the Western Regions`.

```
$ awk 'BEGIN { print "Developers in the Western Regions" }
> / [A-Z]W/ { print $2, $3, $4, $1 }
> /W[A-Z]/ { print $2, $3, $4, $1 }' data.file
```

4. For the output of Step 2, add a trailing footer to the report called `End of Report on the Western Regions`.

```
$ awk 'BEGIN { print "Developers in the Western Regions" }
> / [A-Z]W/ { print $2, $3, $4, $1 }
> /W[A-Z]/ { print $2, $3, $4, $1 }
> END { print "End of Report on the Western Regions" }' data.file
```

5. For every line of the file, print a line number, followed by a colon, followed by a tab, and finally the entire record. (Hint: Check Table 6-2 on page Module 6-12.)

```
$ awk '{ print NR, ":\t" $0 }' data.file
```

6. Set the output field separator to a tab, and make the initial heading for this report `***** Regional Report *****`. For every line of the file, print a line number, followed by the last name of the person, and finally the region name.

```
$ awk 'BEGIN { OFS="\t" }
> BEGIN { print "***** Regional Report *****" }
> { print NR, $4, $1 }' data.file
```

User-Defined Variables

To create a variable, use a name that is not one of the `nawk`-predefined variable or function names.

The first time `nawk` encounters a reference to a given variable, the variable is created and initialized to a null string. (The null string evaluates to 0 if it is used in a numeric expression.) For all subsequent references, the value of the variable is whatever value was assigned last. The value assigned can be a string (string of characters) or numeric. It can be a literal value ("hello") or the contents from a field from the input file.

Assign a value directly, or use the current value of the variable and use a combination operation on the variable; for example:

```
$ cat numexample.nawk
{ counter = counter + 1 }
{ print $0 }
END { print "*** The number of records is " counter }

$ nawk -f numexample.nawk data.file
northwest      NW      Joel Craig      3.0 .98 3      4
western        WE      Sharon Kelly     5.3 .97 5      23
southwest      SW      Chris Foster    2.7 .8 2      18
southern        SO      May Chin       5.1 .95 4      15
southeast       SE      Derek Johnson   5.0 .70 4      17
eastern         EA      Susan Beal      4.4 .8 5      20
northeast       NE      TJ Nichols     5.1 .94 3      13
north          NO      Val Shultz     4.5 .89 5      9
central         CT      Sheri Watson   5.7 .94 5      13
*** The number of records is 9
```

Adding a Field of Numbers

Each field of a record has a variable assigned to it. The first field is `$1`, the second field `$2`, and so forth. The value of these variables change each time `nawk` reads a new record. To sum the values of a field, create a new variable (for example, name the variable *total*). Add the value of the field to be summed to the variable each time a record is read.

User-Defined Variables

The following example prints the value in Field 8 for each record, and adds that value to the *total* variable. At the end of file, it prints the accumulated total of all values found in Field 8.

```
$ cat numexample2.nawk
{ total = total + $8 }
{ print "Field 8 = " $8 }
END { print "Total = " total }

$ nawk -f numexample2.nawk data.file
Field 8 = 4
Field 8 = 23
Field 8 = 18
Field 8 = 15
Field 8 = 17
Field 8 = 20
Field 8 = 13
Field 8 = 9
Field 8 = 13
Total = 132
```

A total can be written at the end of the file.

```
$ cat numexample3.nawk
{ total = total + $8 }
{ print $0 }
END { print "The total of field 8 is " total }

$ nawk -f numexample3.nawk data.file
northwest      NW      Joel Craig      3.0 .98 3      4
western        WE      Sharon Kelly     5.3 .97 5      23
southwest      SW      Chris Foster    2.7 .8 2      18
southern        SO      May Chin       5.1 .95 4      15
southeast       SE      Derek Johnson   5.0 .70 4      17
eastern         EA      Susan Beal      4.4 .8 5      20
northeast       NE      TJ Nichols     5.1 .94 3      13
north          NO      Val Shultz     4.5 .89 5      9
central         CT      Sheri Watson   5.7 .94 5      13
The total of field 8 is 132
```

Variable Examples

The following example prints the record number followed by the entire record of any record containing a NE, NO, or NW:

```
$ awk '/N[EOW]/ { print NR, $0 }' data.file
1 northwest      NW      Joel Craig      3.0 .98 3      4
7 northeast      NE      TJ Nichols      5.1 .94 3      13
8 north          NO      Val Shultz      4.5 .89 5      9
```

The following example prints the record number followed by the entire record if it contains an N. In addition, at the end of the file, it prints the number of records that matched the specified pattern.

```
$ awk 'BEGIN { count = 0 }
> /N/ { print NR, $0; count = count + 1 }
> END { print "count of North regions is", count }' data.file
1 northwest      NW      Joel Craig      3.0 .98 3      4
7 northeast      NE      TJ Nichols      5.1 .94 3      13
8 north          NO      Val Shultz      4.5 .89 5      9
count of North regions is 3
```

The following example prints the record number and then the contents of the last field of the record.

```
$ awk '{ print "Record:", NR, $NF }' data.file
Record: 1 4
Record: 2 23
Record: 3 18
Record: 4 15
Record: 5 17
Record: 6 20
Record: 7 13
Record: 8 9
Record: 9 13
```

The previous awk command used \$NF rather than \$8 to obtain the value for the last field in each record. This works because, in awk, the NF variable returns the number of fields in a record. Because awk variables are not preceded by a \$, the \$ is used to convert the result from 8 to \$8. NF equals 8 and \$NF equals \$; therefore, \$NF returns the value in field 8.

Variable Examples

The nawk variable NF is set for each record to the number of fields in the record. Therefore, if your data file is not guaranteed to have the same number of fields per record, this variable always reports that unknown count.

```
$ cat raggeddata.file
northwest      NW      Joel Craig      3.0 .98 3      4
WE      Sharon Kelly    5.3 .97 23
southwest     SW      Chris Foster    2.7 .8 2      18
southern       SO      May Chin      5.1 .95 15
southeast      SE      Derek      5.0      17
eastern        Susan Beal    4.4 .8      20
NE      TJ Nichols    5.1 .94 3      13
Val Shultz     4.5      5      9
central        CT      Sheri Watson   .94      5
```

The following example prints the record number followed by the number of fields in the record, for each record in the `raggeddata.file` input file.

```
$ nawk '{ print "Record:", NR, "has", NF, "fields." }' raggeddata.file
Record: 1 has 8 fields.
Record: 2 has 6 fields.
Record: 3 has 8 fields.
Record: 4 has 7 fields.
Record: 5 has 5 fields.
Record: 6 has 6 fields.
Record: 7 has 7 fields.
Record: 8 has 5 fields.
Record: 9 has 6 fields.
```

The nawk `length()` function returns the number of characters in a variable value. The following example prints the string `Field 1 has`, followed by the number of characters in `$1`, followed by the string `letters.`.

```
$ nawk '{ print "Field 1 has", length($1), "letters." }' raggeddata.file
Field 1 has 9 letters.
Field 1 has 2 letters.
Field 1 has 9 letters.
Field 1 has 8 letters.
Field 1 has 9 letters.
Field 1 has 7 letters.
Field 1 has 2 letters.
Field 1 has 3 letters.
Field 1 has 7 letters.
```

Writing Output to Files

In any statement that generates output, it is possible to have that output redirected to a file name instead, by using the UNIX redirection symbol (>) followed by a file name at the end of the output-generating statement. Enclose the file name in quotes if it is a string of characters. If the file name is stored in a variable or field, use the field number (preceded by the \$ character) or the variable name.

In the following example, the second and first fields are written to the `textfile` file.

```
$ nawk '{ print $2, $1 > "textfile" }' data.file  
  
$ cat textfile  
NW northwest  
WE western  
SW southwest  
SO southern  
SE southeast  
EA eastern  
NE northeast  
NO north  
CT central
```

The printf() Statement

The printf() Statement

The printf() statement allows you to print a character string. This string can contain place holders that represent other values. These place holders can represent integers, floating points, characters, and character strings. For every place holder in the character string of printf(), there must be a value following the character string. These values must be separated by a comma.

The syntax of printf() is:

```
printf ("string_of_characters" [ , data_values ] )
```

where the *data_values* are used to fill in placeholders in the "string_of_characters" argument. Some of the placeholders are:

%d	Integer value
%c	Character value
%s	String value

Examples:

```
printf( "Hello World\n" )
printf( "The value is %d\n", num )
printf( "My name is %-10s %20s\n", $3, $4 )
```

Note – Notice the use of \n in the preceding examples. The printf statement does not generate a newline character by default.

Formatting options are available for each of the place holders. Some of them are:

%numberd	These options indicate a field width. The value is right-justified within the field.
%numbers	
%numberc	
%-numberd	These options indicate the value is left-justified within the indicated field width.
%-numbers	
%-numberc	

The following is an example of formatting with the place holders:

```
printf( "%d, %20s\n", 54, "John Smith" )
```

The previous example prints:

```
54,      John Smith
```

The printf statement does not use the OFS variable because of the printf statement's inherent ability to format the output line, so adjusting the fields to appear in column format is done by preceding the format specification with a digit. This digit refers to the number of spaces that the value should consume.

```
$ nawk '{ printf "%10s %3d \n", $4,$7 }' data.file
    Craig      3
    Kelly      5
    Foster     2
    Chin       4
    Johnson    4
    Beal       5
    Nichols    3
    Shultz     5
    Watson     5
```

Exercise: Using nawk to Create a Report

Exercise: Using nawk to Create a Report

Exercise objective – Write a report on the top three users of disk space in a file system. The nawk command is only one of the commands you will use to obtain and manipulate the data. You can execute all the commands from the command line. Then, when you are certain of the commands and command sequence, you can place the commands in a shell script for execution.

Tasks

1. Report on the root file system. Use the `/sbin/quot` command to report on the number of blocks in the `/` file system currently owned by each user. The output is listed in descending order by the number of blocks.
2. Strip off the first line of output that specifies the actual logical device name for that slice of the disk. Save the results in a file named `users1`.
3. Execute a command to list only the first three lines of the `users1` file.
4. Save the listing of the first three lines into the `users2` file.
5. Write a nawk script to read the `users2` file and produce the following output:
 - Print a heading at the top of the report.
 - Put in column headings the user name and the number of blocks.
 - For each record in `users2`:
 - Print the user's name
 - Print the number of blocks reported for that user

The report should resemble the following:

```
Report on Top 3 Users in / Filesystem
  Username Number of Blocks
    root      1647941
      bin      804135
    daemon     27009
```

6. Write a shell script that executes, in order, the commands identified in the previous steps.

Exercise: Using nawk Scripts to process Text Files

Exercise objective – Write nawk commands to process text files and generate specified output or simple reports.

Preparation

Refer to the lecture notes as necessary to correctly compose the nawk commands.

Change the directory to mod6/lab before beginning the exercise.

For the multi-statement nawk commands, you might want to execute the individual statements on the system to ensure each generates the required output. After these individual statements prove correct, combine them into a multi-statement nawk command.

Tasks

1. Write a nawk program called northsouthreport.nawk that reads the data.file file and prints a report of all records that contain north or south in the region name. The program must count how many records are listed for both the north and south regions and should output that count at the end of the report.

Make the report look like the following.

Record #	Region Name	Employee	Region Count
1	northwest	Craig	1
3	southwest	Foster	1
4	southern	Chin	2
5	southeast	Johnson	3
7	northeast	Nichols	2
8	north	Shultz	3

Total of North region is: 3

Total of South region is: 3

2. Write a nawk script called raggedcount.nawk that counts the number of fields in the raggeddata.file file.

Note – Use the mypasswd file found in the mod6/lab directory for the tasks that follow.

Exercise: Using nawk Scripts to process Text Files

3. Write a nawk script called korncounter.nawk that counts and lists the Korn shell user entries in the mypasswd file.
4. Write a nawk script called bournecounter.nawk that counts and lists the Bourne shell user entries in the mypasswd file.
5. Write a shell script called kshshscript.ksh that calls both of the nawk scripts that count users in the mypasswd file based on their default shell. The output of the shell script should look like the following:

This is a report on default shells for the local machine

Here is the listing of Korn shell users

```
user2:x:1002:10::/export/home/user2:/bin/ksh
==> Total Korn shell users: 1
```

Here is the listing of Bourne shell users

```
root:x:0:1:Super-User:/sbin/sh
user1:x:1001:10::/export/home/user1:/bin/sh
==> Total Bourne shell users: 2
```

6. Copy the adduser script from the mod4/lab directory, and modify it to do the following.
 - a. Use nawk to extract all the user identification numbers (UIDs) out of the mypasswd file into a file called currentuid.
 - b. Use sed to remove all the system UID numbers, meaning all the one-, two-, and three-digit UIDs. You might want to put the remaining UIDs in a file named currentuid2.
 - c. Use sed to remove the nobody (60001), noaccess (60002), and nobody4 (65534) UIDs. You might want to put the remaining UIDs in a file named currentuid3.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Task Solutions

1. Write a nawk program called `northsouthreport.nawk` that reads the `data.file` file and prints a report of all records that contain north or south in the region name. The program must count how many records are listed for both the north and south regions and should output that count at the end of the report.

Make the report look like the following.

Record #	Region Name	Employee	Region Count
1	northwest	Craig	1
3	southwest	Foster	1
4	southern	Chin	2
5	southeast	Johnson	3
7	northeast	Nichols	2
8	north	Shultz	3

Total of North region is: 3

Total of South region is: 3

```
$ cat northsouthreport.nawk
BEGIN { Ncount = 0; Scount=0 }
BEGIN { printf "%s %15s %15s %15s \n", "Record #", "Region Name",
"Employee",
"Region Count" }
/north/ { printf "%6d %15s %15s %15d \n", NR, $1, $4, Ncount = Ncount + 1
}
/south/ { printf "%6d %15s %15s %15d \n", NR, $1, $4, Scount = Scount + 1
}
END { print "Total of North region is:", Ncount }
END { print "Total of South region is:", Scount }
```

2. Write a nawk script called `raggedcount.nawk` that counts the number of fields in the `raggeddata.file` file.

```
$ cat raggedcount.nawk
BEGIN { print "This program is counting the number of fields in" }
BEGIN { print "the file raggeddata.file, please wait." }

{ totalfields = totalfields + NF }

END { print "==> The total number of fields is:", totalfields }
```

Note – Use the `mypasswd` file found in the `mod6/lab` directory for the tasks that follow.

3. Write a nawk script called `korncounter.nawk` that counts and lists the Korn shell user entries in the `mypasswd` file.

```
$ cat korncounter.nawk
BEGIN { print "\nHere is the listing of Korn shell users \n" }

/\ksh$/ { print $0; korncounter = korncounter + 1 }

END { print "==> Total Korn shell users:", korncounter }
```

4. Write a nawk script `bournecounter.nawk` that counts and lists the Bourne shell user entries in the `mypasswd` file.

```
$ cat bournecounter.nawk
BEGIN { print "\nHere is the listing of Bourne shell users \n" }

/\sh$/ { print $0; bournetotal = bournetotal + 1 }

END { print "==> Total Bourne shell users:", bournetotal }
```

5. Write a shell script called `kshshscript.ksh` that calls both of the nawk scripts that count users in the `mypasswd` file based on their default shell. The output of the shell script should look like the following:

This is a report on default shells for the local machine

Here is the listing of Korn shell users

```
user2:x:1002:10:::/export/home/user2:/bin/ksh
==> Total Korn shell users: 1
```

Here is the listing of Bourne shell users

```
root:x:0:1:Super-User:/sbin/sh
user1:x:1001:10:::/export/home/user1:/bin/sh
==> Total Bourne shell users: 2
```

Task Solutions

```
$ cat kshshscript.ksh
#!/bin/ksh

print "This is a report on default shells for the local machine"

nawk -f korncounter.nawk mypasswd

nawk -f bournecounter.nawk mypasswd
```

6. Copy the adduser script from the mod4/lab directory, and modify it to do the following.

```
$ cp ../../mod4/lab/solutions/adduser.sh .
```

- a. Use nawk to extract all the UIDs out of the mypasswd file into a file called currentuid.

```
nawk -F: '{print $3}' ./mypasswd > ./currentuid
```

- b. Use sed to remove all the system UID numbers, meaning all the one-, two-, and three-digit UIDs. You might want to put the remaining UIDs in a file named currentuid2.

```
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2
```

- c. Use sed to remove the nobody (60001), noaccess (60002), and nobody4 (65534) UIDs. You might want to put the remaining UIDs in a file named currentuid3.

```
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3
```

```
$ cat adduser.sh
#!/bin/sh

# Purpose: To write a script to add user to the system.
# Name: adduser

### Set a shell variable to test the grep command.
name=nuucp      # Should match a single entry
#name=uucp      # Should match a single entry
#name=root      # Should match a single entry
#name=rot       # Should not match any entry

grep "^$name:" ./mypasswd

# End of lab 4

# Use awk to extract all the UIDs into the temp file currentuid
awk -F: '{print $3}' ./mypasswd > ./currentuid

# Use sed to remove all 1, 2, and 3 digit UIDs from currentuid, and
# place the output in the temp file currentuid2
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2

# Use sed to remove UIDs 60001, 60002, and 65534 from currentuid2,
# and place the output in the temp file currentuid3
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3

# End of lab 6
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 7

Conditionals

Objectives

Upon completion of this module, you should be able to:

- Use the command exit status as conditional control
- Use the `if` statement to test a condition
- Pass values using command-line arguments (positional parameters) into a script
- Create USAGE messages
- Place parameters on the command line
- Use conditional `if`, `then`, `elif`, `else`, and `fi` constructs
- Use `exit`, `let`, and `test` statements (`[[]]`, `(())`)
- Apply the `&&`, `||`, and `!` Boolean logic operators
- Use the `case` statement

The if Statement

The if Statement

The if statement:

- Allows you to specify courses of action to be taken in a shell script, depending on the success or failure of some command.
- Is a conditional statement that allows a test before performing another statement.
- Has several forms (more complicated forms are described in the “Syntax for if/then/else Statements” on page 7-10 and in the “Syntax for if/then/elif/else Statements” on page 7-11). The syntax for the simplest form is:

```
if command
then
    block_of_statements
fi
```

See the example below. This example uses the single brackets so the code works for both Bourne and Korn shells.

```
$ cat snoopy.sh
#!/bin/sh

# Script name: snoopy.sh

name=snoopy

if [ "$name" = "snoopy" ]
then
    echo "It was a dark and stormy night."
fi
```

Note – When you compare a variable in a condition test, always use double quotes around the variable to ensure it is evaluated properly. Without the double quotes, problems will arise if a string value has a space or if the value of the variable is null.

When there are double quotes around a variable in the if statement, and the variable is null, and the quotes aren't present, the error message is **test: argument expected**.

When a statement executes, the statement returns a numeric value called an exit status. A statement that executes successfully, returns an exit status of 0, meaning 0 errors. A statement that executes unsuccessfully returns an exit status of nonzero, meaning one or more errors occurred.

Therefore, when the statement following the *if* keyword executes successfully and returns a 0 exit status, the statements following the *then* statement are executed. However, when the statement following the *if* keyword executes unsuccessfully and returns a nonzero exit status, the statements following the *fi* statement are executed.

The *then* is a separate statement in a shell. It must appear on a separate line unless it is preceded by a ; (semicolon). This is similar to the way several OS commands are provided on one line as long as a ; separates each command. For example, the statement:

if command; then

is treated as being the same as:

*if command
then*

Parts of the `if` Statement

Parts of the `if` Statement

The `if` statement contains three parts:

1. The command
2. The block of statements
3. The end of the `if` statement

Command

The *command* in the `if` statement often involves a numerical or string test comparison, but it can also be any command that returns a status of 0 when it succeeds and some nonzero status when it fails.

Some of the conditions frequently used within an `if` are:

- The `((...))` command performs arithmetic comparisons (Korn shell).
- The `[[...]]` command is primarily used to compare strings (Korn shell).
- The `[...]` command performs arithmetic or string comparisons (Bourne or Korn shells).

Block of Statements

The statements that follow the `then` statement can be any valid UNIX command, any executable user program, any executable shell script, or any shell statement with the exception of `fi`.

End of the `if` Statement

End every `if` statement with the `fi` statement.

Exit Status

A shell reserved variable (?) holds the integer representing the exit status of the previously executed statement, script, or shell statement. This is the most recently completed foreground process.

The following example uses grep to search the /etc/passwd file for the root string. The success or failure of the statement is displayed by echoing \$?.

```
$ grep root /etc/passwd
root:x:0:1:Super-User:/sbin/sh
$ echo $?
0
```

The following example sends the output to the /dev/null file, and tests it for successful completion.

```
$ grep root /etc/passwd > /dev/null
$ echo $?
0
```

The following example places the condition testing within the if statement:

```
$ grep root /etc/passwd
root:x:0:1:Super-User:/sbin/sh
$ if [ $? -eq 0 ]
> then
>     echo "Found root!"
> fi
Found root!
```

Instead of executing a statement, and then testing if it completed successfully, the following example places the statement in the condition:

```
$ if grep root /etc/passwd
> then
>     echo "Found root!"
> fi
root:x:0:1:Super-User:/sbin/sh
Found root!
```

You may redirect or pipe stdout as part of the output.

```
$ if grep root /etc/passwd > /dev/null
> then
>     echo "Found root!"
> fi
Found root!
```

Numeric and String Comparison

When you compare numbers in the Korn shell, use double parentheses `((...))`. There is no requirement about white space around any of the components. Use white space around variables and operators for improved readability.

The `((...))` construct is an abbreviation of the `let` statement. Variables can be preceded by a `$` within `((...))`, but the `$` character is not required.

When using square brackets `[]`, the shell requires a space after the opening `[` and before the closing `]`. For example, `[$a -eq $b]` is invalid. `[$a -eq $b]` is the correct use.

Table 7-1 shows numeric comparisons.

Table 7-1 Numeric Comparison

Bourne and Korn Shells	Korn Shell	Returns true (0) if:
<code>[\$num1 -eq \$num2]</code>	<code>((num1 == num2))</code>	<code>num1</code> equals <code>num2</code>
<code>[\$num1 -ne \$num2]</code>	<code>((num1 != num2))</code>	<code>num1</code> does not equal <code>num2</code>
<code>[\$num1 -lt \$num2]</code>	<code>((num1 < num2))</code>	<code>num1</code> is less than <code>num2</code>
<code>[\$num1 -gt \$num2]</code>	<code>((num1 > num2))</code>	<code>num1</code> is greater than <code>num2</code>
<code>[\$num1 -le \$num2]</code>	<code>((num1 <= num2))</code>	<code>num1</code> is less than or equal to <code>num2</code>
<code>[\$num1 -ge \$num2]</code>	<code>((num1 >= num2))</code>	<code>num1</code> is greater than or equal to <code>num2</code>

The following example sets a variable and tests the value of the variable using the double parenthesis in an `if` statement. The then statement is placed on the same line with the `if`.

```
$ num=21

$ if (( num > 15 )); then
>     echo "You are old enough to drive in most places."
> fi
You are old enough to drive in most places.
```

The (()) syntax is the alternative form to the let statement. The (()) syntax is preferred because it is more readable. The previous syntax can also be written as:

```
$ if let "num > 15"; then
>     echo "You are old enough to drive in most places."
> fi
You are old enough to drive in most places.
```

When comparing strings in Korn shell, use double square brackets [[...]]. Here, the shell is particular about white space. You *must* use white space around every component within [[...]]. This includes putting a white space after [[and before]].

When comparing a string to a pattern, the string must appear on the left side of the equal sign and the pattern must appear on the right.

Table 7-2 shows string comparisons.

Table 7-2 String Comparison

Bourne and Korn Shell	Korn Shell	Returns true (0) if:
[str1 = str2]	[[str1 == str2]]	str1 equals str2
[str1 != str2]	[[str1 != str2]]	str1 does not equal str2
Not available	[[str1 == pattern]]	str1 matches pattern
Not available	[[str1 != pattern]]	str1 does not match pattern
[str1 < str2]	[[str1 < str2]]	str1 precedes str2 in lexical order
[str1 > str2]	[[str1 > str2]]	str1 follows str2 in lexical order
[-z str1]	[[-z str1]]	str1 has length zero (holds null value)
[-n str1]	[[-n str1]]	str1 has nonzero length (contains one or more characters)

Note – Lexical order means that lowercase letters have greater value than uppercase letters. As such, attempting to compare mixed-case strings might produce unexpected results.

In regard to string comparisons with either the [] or [[]] statement, use quotes around literal strings or variables. The quotes are necessary when the variable value or string value has embedded spaces.

Numeric and String Comparison

The following example sets a variable and then tests the value of the variable using the double square brackets in an `if` statement.

```
$ name=fred
$ if [[ "$name" == "fred" ]]
> then
>   echo "fred is here!"
> fi
fred is here!
```

When comparing a variable or string against a pattern, the pattern is not quoted; using quotes hides the pattern-match capability of the pattern metacharacters. Table 7-3 shows the metacharacters used for comparison.

Table 7-3 Pattern Match Metacharacters

Metacharacter	Meaning
?	Matches any one single character
[]	Matches one character in the specified set
*	Matches zero or more occurrences of any characters
? (<i>pat1 pat2 ... patn</i>)	Matches zero or one of the specified patterns
@ (<i>pat1 pat2 ... patn</i>)	Matches exactly one of the specified patterns
* (<i>pat1 pat2 ... patn</i>)	Matches zero, one, or more of the specified patterns
+ (<i>pat1 pat2 ... patn</i>)	Matches one or more of the specified patterns
! (<i>pat1 pat2 ... patn</i>)	Matches any pattern except the specified patterns

It is important that the contents of the double-square-bracket conditional tests start and end with a space. Surround the conditional operators by at least one space. Also, as the square brackets act as a quoting mechanism, so do not use quotes to surround the pattern text.

The following example sets a variable and then compares the value of the variable against a pattern. The pattern is required to be on the right side of the operator.

```
$ name=fred  
  
$ if [[ "$name" == f* ]]; then  
>   echo "fred is here!"  
> fi  
fred is here!
```

Incorrect quoting is shown in the following example:

```
$ name=fred  
  
$ if [[ "$name" == "f*" ]] ; then  
> echo "fred is here"  
> fi
```

Syntax for if/then/else Statements

Syntax for if/then/else Statements

In a conditional, you frequently have tasks to perform when the tested command succeeds or fails. The shell can accommodate this with the if/then/else syntax.

In the if/then/else form of the if statement, the block of statements after the then statement is executed if the command succeeds. Execution continues with the statement following the fi statement. If the command in the if statement fails, then the block of statements after the then statement is skipped, and statements following the else are executed. Execution continues with the statement following the fi statement. The syntax for if/then/else is:

```
if command
then
    block_of_statements
else
    block_of_statements
fi
```

The following are two examples:

```
$ cat snoopynap.ksh
#!/bin/ksh

# Script name: snoopynap.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
    echo "It was a dark and stormy night."
else
    echo "Snoopy is napping."
fi

$ cat findroot.ksh
#!/bin/ksh

# Script name: findroot.ksh

if grep root /etc/passwd > /dev/null
then
    echo "Found root!"
else
    echo "root not in the passwd!"
    echo "Do not logout until the passwd file is repaired!"
fi
```

Syntax for if/then/elif/else Statements

In the if/then/elif/else form of the if statement, the first else becomes another if statement or “else if” instead of a simple else.

The shell first evaluates *command1*, then *command2*, and so on, stopping with the first command that succeeds. The statements associated with that successful command are then executed, followed by any statements after the fi statement.

If none of the commands succeeds, then the statements following the else statement are executed. Execution then continues with any statements following the fi statement.

The syntax for if/then/elif/else is:

```
if command1
then
    block_of_statements
elif command2
then
    block_of_statements
else
    block_of_statements
fi
```

Syntax for if/then/elif/else Statements

For example:

```
$ cat snoopy2.ksh
#!/bin/ksh

# Script name: snoopy2.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
    echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
    echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
    echo "The doctor is in."
elif [[ "$name" == "schroeder" ]]
then
    echo "In concert."
else
    echo "Not a Snoopy character."
fi
```

Note – The if/then/elif/else syntax can have as many elif command statements as needed before the final else statement. Follow each elif by a then statement.

In the following example, the value of the TERM variable is tested by comparing it against different known values. As soon as one of the tests succeeds, the statements that follow the then statement are executed. If all comparisons of TERM fail, the statements following the else statement are executed.

```
$ cat termcheck.ksh
#!/bin/ksh

# Script name: termcheck.ksh

if [[ "$TERM" == "sun" ]]
then
    echo "You are using the sun console device."
elif [[ "$TERM" == "vt100" ]]
then
    echo "You are using a vt100 emulator."
elif [[ "$TERM" == "dtterm" ]]
then
    echo "You are using a dtterm emulator."
else
    echo "I am not sure what emulator you are using."
fi
```

Positional Parameters

Positional Parameters

The execution of scripts is made versatile by the ability to run a script based on arguments supplied on the command line. In this way, the operation of the script varies depending on what arguments are given to the script.

The shell automatically assigns special variable names, called *positional parameters*, to each argument supplied to a script on the command line. The positional parameter names and meanings are shown in Table 7-4.

Table 7-4 Positional Parameters

Positional Parameter Name	Description
\$0	The name of the script
\$1	The first argument to the script
\$2	The second argument to the script
\$9	The ninth argument to the script
$\${10}, \${11}, \${n}$	The tenth and up argument to the script (Korn shell only)
\$#	The number of arguments to the script
\$@	A list of all arguments to the script
\$*	A list of all arguments to the script
$\${\#N}$	The length of the value of positional parameter N (Korn shell only)

\$@ and \$* have the same meaning. This is true if they are not enclosed in double quotes " ". The distinction is described in greater detail later in the “The Values of the \$@” and “\$*” Positional Parameters” on page 11-17.

Note – The Bourne shell stores all the arguments passed to the script, but can refer to only the positional parameters \$1 through \$9. To obtain the values beyond the ninth argument, use the `shift` statement, which is destructive in nature.

Using if to Check Command-Line Arguments

The following script captures the command-line arguments in special variables: \$# , \$1 , \$2 , \$3 , and so on. The \$# variable captures the number of command-line arguments. The following example sets variables within a script.

```
$ cat numtest.ksh
#!/bin/ksh

# Script name: numtest.ksh

num1=5
num2=6

if (( $num1 > $num2 ))
then
    print "num1 is larger"
else
    print "num2 is larger"
fi
```

The following example shows changing the script so that you can enter command-line arguments:

```
$ cat argtest.ksh
#!/bin/ksh

# Script name: argtest.ksh

if (( $1 > $2 ))
then
    print "num1 is larger"
else
    print "num2 is larger"
fi
```

Do the following to execute the script:

```
$ ./argtest.ksh 21 11
num1 is larger

$ ./argtest.ksh 1 11
num2 is larger
```

Creating the USAGE Message

Creating the USAGE Message

When you write a script and set up user interaction, generally the script expects a certain type of user input. For example, if the script prints a request for an age, it expects you to enter a numeric value. If you enter a name instead, the script might attempt an arithmetic operation on the variable and get an error.

The script should verify that the type of input and the number of values input are correct. If they are not correct, then the script can print an error message in the form of a USAGE message; for example:

```
if (( $# != 2 ))
then
    print "USAGE: $0 arg1 arg2 "
    exit
fi
```

In this case, the script expects two positional parameters to be entered when the script is run. If you do not provide the two parameters, the script prints out the USAGE message.

Including the USAGE message is a good programming practice. It is the conventional way to notify the person that executes the script that execution was not done correctly. Most operating system commands provide this type of information when you try to execute a command with an undefined option, such as ls -z.

```
$ ls -z
ls: illegal option -- z
usage: ls -1RaAdCxmnlhogrtuvVcpFbqisfHLeE@ [files]

$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright 2006 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# ident "@(#)volmgt      1.9      06/01/20 SMI"

case "$1" in
'start')
    if [ -f /etc/vold.conf -a -f /usr/sbin/vold -a \
        "${_INIT_ZONENAME:=`/sbin/zonename`}" = "global" ]; then
        echo 'volume management starting.'
        svcadm enable svc:/system/filesystem/volfs:default
    fi
    ;;

'stop')
    svcadm disable svc:/system/filesystem/volfs:default
    ;;

*)
    echo "Usage: $0 { start | stop }"
    exit 1
    ;;

esac
exit 0
```

Using if to Check Leap Years

Using if to Check Leap Years

The next example script:

- Uses the date statement to set the variable mth to an integer representing the current month (01 is January, 12 is December).
- Tests variable mth value:
 - If the current month is February (mth is 02), prints messages concerning the number of days in February.
 - If the current month is April, June, September, or November, prints the message: The current month has 30 days.
 - If the current month is January, March, May, July, August, October, or December -- all have 31 days -- uses the else statement.

The test in the elif branch uses pattern matching specific to the Korn shell. The specific syntax used in the example requires that the value of mth exactly matches one of the specified patterns.

```
$ cat monthcheck.ksh
#!/bin/ksh

# Script name: monthcheck.ksh

mth=$(date +%m)

if (( mth == 2 ))
then
    echo "February usually has 28 days."
    echo "If it is a leap year, it has 29 days."
elif [[ $mth = @(04|06|09|11) ]]
then
    echo "The current month has 30 days."
else
    echo "The current month has 31 days."
fi

$ date
Fri Oct 16 13:28:24 GMT 2009

$ ./monthcheck.ksh
The current month has 31 days.
```

Nested if Statements

You can use an `if` statement inside another `if` statement (it's ok to have nested `if` statements). You can have as many levels of nested `if` statements as you can track.

In the following example:

1. You input the year as a command-line argument.
2. The script assigns the value of `$1` to the `year` variable.
3. The rest of the script determines if the year entered is a leap year and prints an appropriate message if it is.

The second `if` statement contains an `if` statement as one of its statements, which is where the nesting occurs. For the script to print that the year as a leap year:

- The condition that the year entered be evenly divisible by 4 must be true.
- The condition that the year not be evenly divisible by 100 must also be true.

The script is then run several times from the command line. The numbers input to the script are chosen to test that something gets printed each time. It is not necessary to always have an `else` with an `if`. However, you must include an `fi` statement for each `if` statement.

Nested if Statements

```
$ cat leap.ksh
#!/bin/ksh

# Script name: leap.ksh

# Make sure the user enters the year on the command line
# when they execute the script.

if [ $# -ne 1 ]
then
    echo "You need to enter the year."
    exit 1
fi

year=$1

if (( (year % 400) == 0 ))
then
    print "$year is a leap year!"
elif (( (year % 4) == 0 ))
then
    if (( (year % 100) != 0 ))
    then
        print "$year is a leap year!"
    else
        print "$year is not a leap year."
    fi
else
    print "$year is not a leap year."
fi

$ ./leap.ksh 2000
2000 is a leap year!

$ ./leap.ksh 1900
1900 is not a leap year.

$ ./leap.ksh 2050
2050 is not a leap year.
```

Testing File Objects

Sometimes a script might need to determine the permissions on a file or file type. You can test a file using one of the flags shown in Table 7-5. The file or path name might be literal or stored in a variable.

Table 7-5 Determining File Permissions and Ownerships

Bourne or Korn Test	Korn Test	Description
[-r file]	[[-r file]]	Can the file be read by user?
[-w file]	[[-w file]]	Can the file be altered by user?
[-x file]	[[-x file]]	Can the file be executed by user?
Not available	[[-O file]]	Is the file owned by the effective user ID of this process?
Not available	[[-G file]]	Is the file group the effective group ID of this process?
[-u file]	[[-u file]]	Does the file have the set-user-ID bit set?
[-g file]	[[-g file]]	Does the file have the set-group-ID bit set?
[-k file]	[[-k file]]	Does the file have the sticky bit set?

Knowing the file type with which you are working is important. For example, you cannot perform the same operations on a directory as you can on a regular file. (Try running the cat command on a directory file or the cd command on a regular file.)

Testing File Objects

The type of file can be determined using the flags shown Table 7-6.

Table 7-6 Determining the File Type

Bourne or Korn Test	Korn Test	Description
[-f <i>file</i>]	[[-f <i>file</i>]]	Is the file a regular file?
[-d <i>file</i>]	[[-d <i>file</i>]]	Is the file a directory?
[-c <i>file</i>]	[[-c <i>file</i>]]	Is the file a character special file?
[-b <i>file</i>]	[[-b <i>file</i>]]	Is the file is a block special file?
[-p <i>file</i>]	[[-p <i>file</i>]]	Is the file a named pipe?
Not available	[[-S <i>file</i>]]	Is the file a socket?
Not available	[[-L <i>file</i>]]	Is the file a symbolic link?
[-s <i>file</i>]	[[-s <i>file</i>]]	Does the file exist and have a size greater than 0?
Not available	[[-e <i>file</i>]]	Does the file exist?

Boolean AND, OR, and NOT Operators

The Boolean AND operator is represented by `&&` for the Korn shell and by `-a` for the Bourne shell. It requires two statements. If both statements succeed, the AND operator returns true; otherwise, it returns false.

The Boolean OR operator is represented by `||` for the Korn shell and by `-o` for the Bourne shell. It requires two statements. If both statements fail, the OR operator returns false; otherwise, it returns true.

The Boolean NOT operator is represented by `!` for both the Bourne and Korn shells. It performs a logical inversion of the value passed to it. (Success becomes failure or failure becomes success.) It can be used inside `((...))` or within `[[...]]`.

The operators `&&` and `||` can occur both outside and inside `((...))` or `[[...]]`. The following example shows the `||` operator outside `((...))` and the `&&` operator inside `((...))`.

```
$ cat leap2.ksh
#!/bin/ksh

# Script name: leap2.ksh

# Make sure the user enters the year on the command line
# when they execute the script.

if [ $# -ne 1 ]
then
    echo "You need to enter the year."
    exit 1
fi

year=$1

if (( ( year % 400 ) == 0 )) ||
   (( ( year % 4 ) == 0 && ( year % 100 ) != 0 ))
then
    print "$year is a leap year!"
else
    print "$year is not a leap year."
fi

$ ./leap2.ksh 2000
2000 is a leap year!

$ ./leap2.ksh 2003
2003 is not a leap year.
```

Boolean AND, OR, and NOT Operators

The following is the same program written with Bourne shell Boolean relational operators:

```
$ cat leap2.sh
#!/bin/sh

# Script name: leap2.sh

# Make sure the user enters the year on the command line
# when they execute the script.

if [ $# -ne 1 ]
then
    echo "You need to enter the year."
    exit 1
fi

year=$1

if [ `expr $year % 400` -eq 0 -o \
      `expr $year % 4` -eq 0 -a `expr $year % 100` -ne 0 ]
then
    echo "$year is a leap year!"
else
    echo "$year is not a leap year."
fi

$ ./leap2.sh 2000
2000 is a leap year!

$ ./leap2.sh 2003
2003 is not a leap year.
```

The previous example was complicated in the Bourne shell because of the lack of built-in arithmetic. A simpler example of the AND and OR operators is the following Bourne shell version of the Korn shell monthcheck.sh script shown earlier in this module.

```
$ cat monthcheck.sh
#!/bin/sh

# Script name: monthcheck.sh

mth='date +%m'

if [ "$mth" -eq 02 ]
then
    echo "February usually has 28 days."
    echo "If it is a leap year, it has 29 days."
elif [ \($mth" -eq 04 \|) -o \($mth" -eq 06 \|) -o \
      \($mth" -eq 09 \|) -o \($mth" -eq 11 \|) ]
then
    echo "The current month has 30 days."
else
    echo "The current month has 31 days."
fi
```

The Boolean NOT is often used in testing file objects. For example, to test if a file is *not* readable as opposed to readable or to test if an object is *not* a regular file, use the following statements:

```
if [[ ! -r $var ]]      # if $var is not readable ...
if [[ ! -f $var ]]      # if $var is not a regular file...
```

The case Statement

The case Statement

The `if/then/else` construct makes it easy to write programs that choose between two alternatives. Sometimes, however, a program needs to choose one of several alternatives. You could do this by using the `if/then/elif/else` construct, but in many cases it is more convenient to use the `case` statement. The syntax for the `case` statement is:

```
case value in
  pattern1)
    statement1
    ...
    statementN
    ;;
  pattern2)
    statement1
    ...
    statementN
    ;;
  *)
    statement1
    ...
    statementN
    ;;
esac
```

In the syntax given for the `case` statement, `value` refers to any value, but it is usually the value of some variable. The patterns used in the `case` statement can be any pattern, including the shell metacharacters.

Include two semicolon terminators (that is, `; ;`) after the last statement in the group for each pattern. This prohibits the shell from *falling through to* the statements for the next pattern and executing them. Think of `; ;` as saying “Break out of the `case` statement now and go to the statement that follows the `esac` statement.”

The last pattern in a `case` statement is often the `*` metacharacter. If the value of the variable being tested does not match any of the other patterns, it always matches the `*` metacharacter. This is often called the *default pattern match*, and it is analogous to the `else` statement in the `if` constructs. The patterns used in the `case` statement can be any pattern, including the shell metacharacters listed in Table 7-3.

Example of Using the case Statement

The example for the `case` statement is the `if_scr2` script rewritten to use the `case` statement in place of the `if` statement.

The first two patterns within the `case` statement are the string patterns that the variable `nth` was compared against in the `if` statement in the `if_scr2` script.

The last pattern in the `case` statement is the `*` metacharacter. If `nth` does not match the first two patterns, it always matches this one. It is analogous to the `else` within an `if` statement.

```
$ cat case.ksh
#!/bin/ksh

# Script name: case.ksh

mth=$(date +%m)

case "$mth" in
02)
    print "February usually has 28 days."
    print "If it is a leap year, it has 29 days."
    ;;

04|06|09|11)
    print "The current month has 30 days."
    ;;

*)
    print "The current month has 31 days."
    ;;
esac

$ date
Tue Oct 16 08:25:16 PDT 2009

$ ./case.ksh
The current month has 31 days.
```

Example of Using the case Statement

A more familiar example is a boot script, such as the /etc/init.d/volmgt script. In many boot scripts, the first argument to the script is checked in a case statement to determine if the daemon should be started or stopped. In the following example, the case not only searches for the patterns start and stop, but it also uses the default * metacharacter, which it executes when you type something other than start or stop.

```
$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright 2006 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# ident "@(#)volmgt      1.9      06/01/20 SMI"

case "$1" in
'start')
    if [ -f /etc/vold.conf -a -f /usr/sbin/vold -a \
        "${_INIT_ZONENAME:='sbin/zonename'}" = "global" ]; then
        echo 'volume management starting.'
        svcadm enable svc:/system/filesystem/volfs:default
    fi
    ;;

'stop')
    svcadm disable svc:/system/filesystem/volfs:default
    ;;

*)
    echo "Usage: $0 { start | stop }"
    exit 1
    ;;

esac
exit 0
```

Replacing Complex if Statements With a case Statement

When you have many alternatives to check a variable against, the syntax for an if/then/elif/else can get very confusing. Many shell programmers use the case statement in these situations.

The snoopy example was written with if/then/elif/else statement.

```
$ cat snoopy2.ksh
#!/bin/ksh

# Script name: snoopy2.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
    echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
    echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
    echo "The doctor is in."
elif [[ "$name" == "schroeder" ]]
then
    echo "In concert."
else
    echo "Not a Snoopy character."
fi
```

Replacing Complex if Statements With a case Statement

You can rewrite it with an easier-to-read case statement as follows.

```
$ cat snoopy3.ksh
#!/bin/ksh

# Script name: snoopy3.ksh

name=snoopy

case $name in
"snoopy")
    echo "It was a dark and stormy night."
    ;;

"charlie")
    echo "You're a good man Charlie Brown."
    ;;

"lucy")
    echo "The doctor is in."
    ;;

"schroeder")
    echo "In concert."
    ;;

*)
    echo "Not a Snoopy character."
    ;;

esac
```

The following example is compressed. It performs the same operations as the previous example; however, it places several statements on the same line and does not have blank lines.

```
#!/bin/ksh
case $name in
"snoopy") echo "It was a dark and stormy night." ;;
"charlie") echo "You're a good man Charlie Brown." ;;
"lucy") echo "The doctor is in." ;;
"schroeder") echo "In concert." ;;
*) echo "Not a Snoopy character." ;;

esac
```

The exit Statement

The `exit` statement terminates execution of the entire script. It is most often used if the input requested from the user is incorrect, a statement ran unsuccessfully, or some other error occurred.

The `exit` statement takes an optional argument that is an integer. This number is passed back to the parent shell, where it is stored in the shell reserved `$?` variable. A nonzero integer argument to the `exit` statement means that the script ran unsuccessfully. A zero argument means the script ran successfully.

If no argument is given to the `exit` statement, the shell uses the current value of the `$?` variable.

```
$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright 2006 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# ident "@(#)volmgt      1.9      06/01/20 SMI"

case "$1" in
'start')
    if [ -f /etc/vold.conf -a -f /usr/sbin/vold -a \
        "${_INIT_ZONENAME:='/sbin/zonename'}" = "global" ]; then
        echo 'volume management starting.'
        svcadm enable svc:/system/filesystem/volfs:default
    fi
    ;;
'stop')
    svcadm disable svc:/system/filesystem/volfs:default
    ;;
*)
    echo "Usage: $0 { start | stop }"
    exit 1
    ;;
esac
exit 0
```

The exit Statement

```
$ /etc/init.d/volmgt go
Usage: /etc/init.d/volmgt { start | stop }
$ echo $?
1

$ /etc/init.d/volmgt start
volume management starting.
$ echo $?
0
```

The `exit` statement is often used to terminate a script because of unexpected user input or the occurrence of an error.

Exercise: Using Conditionals

Exercise objective – Answer questions about one of the system boot scripts and write scripts to practice the concepts, statements, and commands described in this module.

Preparation

Refer to the lecture notes as necessary to write the scripts specified in the following tasks.

Change to the to mod7/lab directory before beginning the exercise.

When composing the scripts, you might want to execute commands on the system to ensure you have the correct commands and options to give you the requested output.

Tasks

1. Modify your `/.profile` file to ensure the Korn shell is invoked as your login shell only after you test and verify that the shell file exists. Therefore, if `/usr` was not mounted, the `/.profile` file would not invoke the Korn shell. Make sure you are logged in as the `root` user to perform this step.
2. Read through the following lines, and answer the questions that follow.

Note – The ^[] syntax in the examples is a tab and a space character.

```
startnfsd=0
if [ -f /etc/dfs/dfstab ] && /usr/bin/egrep -v '^[' ]*(#|\$)' \
/etc/dfs/dfstab >/dev/null 2>&1; then
    /usr/sbin/shareall -F nfs
```

The preceding lines of code, determine when the machine should attempt to share all the directories listed in the `dfstab` file. Given that information, answer the following questions.

- a. What does the following command do?

[-f /etc/dfs/dfstab]

Exercise: Using Conditionals

- b. What does the following command do?

```
/usr/bin/egrep -v '^ [ ]* (#| $)' /etc/dfs/dfstab >/dev/null 2>&1
```

- c. What does the `if` statement evaluate?
-

3. Write a script (named `filetype`) that takes exactly one command-line argument and assumes that the command-line argument is a path name to a file or directory, and do the following.

- a. Output a **USAGE** message, and exit the script immediately when the script is executed with more or less than one command-line argument.
- b. If the path name is a directory:
 - List the directory name.
 - List the contents of the directory.
 - List the file space consumed by the directory and all files and subdirectories, in Kbytes. (Realize that this command must perform a calculation before the number is displayed. Therefore, the script will have to wait if you choose a directory structure that is deep or large.)
- c. If the path name is not a directory, assume the path name is a file, and do the following:
 - List the file name.
 - List the file size in bytes.
 - Report whether the file is:
 - A block-special file
 - A character-special file
 - A set-user-id bit file

Hint: To validate the character-special and blocked-special tests, use the physical device name associated with a slice of your local disk.

An example might be the following for the block-special device:

/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a

or the following for the character-special device:

/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a,raw

or the following for the file with set-user-id permission:

/usr/bin/crontab

4. Copy the adduser script from the mod6/lab directory, and then modify the script to do the following.
 - a. Take exactly one command-line argument, which is assumed to be the new user's name. Output a USAGE message and, exit the script immediately when the script is executed with more or less than one command-line argument.
 - b. Either assign the value of \$1 to the name variable or change the grep command to search for \$1.
 - c. If the user name already exists in the mypasswd file, print a message, print the actual mypasswd entry for the user, and then exit the script.
 - d. In the previous lab, you used sed to create a file of nonsystem UID numbers. Numerically sort the contents of this file, and store the output into currentuid4 file.
 - e. The last line in currentuid4 has the maximum UID number currently in use. Increment that number by 1 for the new user.
 - f. Print the new user name and UID.

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Modify your `/.profile` file to ensure the Korn shell is invoked as your login shell only after you test and verify that the shell file exists. Therefore, if `/usr` was not mounted, the `/.profile` file would not invoke the Korn shell. Make sure you are logged in as the `root` user to perform this step.

```
if [ -f /usr/bin/ksh ]
then
    SHELL=/usr/bin/ksh
    export SHELL
    /usr/bin/ksh
fi
```

2. Read through the following lines, and answer the questions that follow.

```
startnfsd=0
if [ -f /etc/dfs/dfstab ] && /usr/bin/egrep -v '^[' ]*(#|$)' \
/etc/dfs/dfstab >/dev/null 2>&1; then
    /usr/sbin/shareall -F nfs
```

The preceding lines of code, determine when the machine should attempt to share all the directories listed in the `dfstab` file. Given that information, answer the following questions.

- a. What does the following command do?

```
[ -f /etc/dfs/dfstab ]
```

Tests to determine if the `/etc/dfs/dfstab` file exists and if it is a regular file. The exit status is 0 if it exists and 1 if it is nonexistent.

- b. What does the following command do?

```
/usr/bin/egrep -v '^[' ]*(#|$)' /etc/dfs/dfstab >/dev/null 2>&1
```

The `egrep` command finds all lines that do not begin with zero or more white spaces followed by a # or end of line (\$) in the `/etc/dfs/dfstab` file. The output and error messages are put into the `/dev/null` file. Then `egrep` returns 0 if it successfully found any lines with the pattern, and it returns 1 if it failed to find the specified pattern.

Task Solutions

- c. What does the `if` statement evaluate?

The `if` statement waits for the results of the Boolean of the `-f` command and the results of the `egrep` command.

The `&&` operator defines that if both commands succeed (return 0 exit status), then the `if` statement returns true. In this situation, the `if` executes the statements following the `then` statement.

The `&&` operator defines that if one or both of the commands fail (return a nonzero exit status), then the `if` statement returns false. In this situation, the `if` executes the statements following the `else` statement (if an `else` statement exists), or the `fi` statement.

3. Write a script (named `filetype`) that takes exactly one command-line argument and assumes that the command-line argument is a path name to a file or directory and then do the following.
- Output a **USAGE** message, and exit the script immediately when the script is executed with more or less than one command-line argument.

```
if [ $# -ne 1 ]; then
    echo "Usage: filetype pathname"
    exit 1
fi
```

- b. If the path name is a directory:

- List the directory name.
- List the contents of the directory.
- List the file space consumed by the directory and all files and subdirectories, in Kbytes. (Realize that this command must perform a calculation before the number is displayed. Therefore, the script will have to wait if you choose a directory that is deep or large.)

```
if [ -d $1 ]; then
    echo "$1 is a directory."
    echo "Following is a listing of the $1 directory."
    ls $1
    echo ""
    echo "The number of kilobytes used by directory $1 is: \c"
    du -s -k $1 | awk '{print $1}'
else
```

- c. If the path name is not a directory, assume the path name is a file, and do the following:

- List the file name.
- List the file size in bytes.
- Report whether the file is:
 - A block-special file
 - A character-special file
 - A set-user-id bit file

```
if [ -d $1 ] ; then
...
else
    echo "$1 is a file."
    echo "The size of $1 in bytes is: \c"
    ls -l $1 | awk '{print $5}'
    echo
    if [ -b $1 ] ; then
        echo "$1 is a block special file."
    elif [ -c $1 ] ; then
        echo "$1 is a character special file."
    elif [ -u $1 ] ; then
        echo "$1 has the set-user-id permission."
    else
fi
```

Hint: To validate the character-special and blocked-special tests, use the physical device name associated with a slice of your local disk.

An example might be the following for the block-special device:

/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a

or the following for the character-special device:

/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a,raw

or the following for the file with set-user-id permission:

/usr/bin/crontab

Task Solutions

The following is the full solution for Step 3:

```
$ cat filetype.sh
#!/bin/sh

# Purpose: This file will determine if the pathname passed on
#           the command line is a file or directory. Then it will
#           print information about the pathname.
#
# Scriptname: filetype.sh

if [ $# -ne 1 ]; then
    echo "Usage: filetype pathname"
    exit 1
fi

if [ -d $1 ]; then
    echo "$1 is a directory."
    echo "Following is a listing of the $1 directory."
    ls $1
    echo " "
    echo "The number of kilobytes used by directory $1 is: \c"
    du -s -k $1 | awk '{print $1}'
else
    echo "$1 is a file."
    echo "The size of $1 in bytes is: \c"
    ls -l $1 | awk '{print $5}'
    echo
    if [ -b $1 ]; then
        echo "$1 is a block special file."
    elif [ -c $1 ]; then
        echo "$1 is a character special file."
    elif [ -u $1 ]; then
        echo "$1 has the set-user-id permission."
    else
        echo "There is nothing special to say about file $1."
    fi
fi

echo "Finished"
```

4. Copy the adduser script from the mod6/lab directory, and then modify the script to do the following:

- Take exactly one command-line argument, which is assumed to be the new user's name. Output a USAGE message, and exit the script immediately when the script is executed with more or less than one command-line argument.

```
if [ $# -ne 1 ]
then
    echo "Usage: adduser newusername"
    exit 1
fi
```

- Either assign the value of \$1 to the name variable or change the grep command to search for \$1.

```
name=$1
```

- If the user name already exists in the mypasswd file, print a message, print the actual mypasswd entry for the user, and then exit the script.

```
if grep "^\$name:" ./mypasswd > /dev/null 2>&1
then
```

```
    echo "The username $1 is already in use, here "
    echo "is the entry from the ./mypasswd file."
    echo
    grep "^\$name:" ./mypasswd
    exit 2
fi
```

- In the previous lab, you used sed to create a file of nonsystem UID numbers. Numerically sort the contents of this file, and store the output into the currentuid4 file.

```
sort -n ./currentuid3 > ./currentuid4
```

- The last line in currentuid4 has the maximum UID number currently in use. Increment that number by 1 for the new user.

```
lastuid='sed -n '$p' ./currentuid4'
uid='expr $lastuid + 1'
```

- Print the new user name and UID.

```
echo "The uid for $name is: $uid"
```

Task Solutions

The following is the full solution for Step 4:

```
$ cat adduser.sh
#!/bin/sh

# Purpose: To write a script to add user to the system.
# Name: adduser_sh

# Check for only a single command line argument
if [ $# -ne 1 ]
then
    echo "Usage: adduser newusername"
    exit 1
fi

# Assign the value of $1 to the variable name
name=$1

if grep "^$name:" ./mypasswd > /dev/null 2>&1
then
    echo "The username $1 is already in use, here "
    echo "is the entry from the ./mypasswd file."
    echo
    grep "^$name:" ./mypasswd
    exit 2
fi

# Use awk to extract all the UIDs into the temp file currentuid
awk -F: '{print $3}' ./mypasswd > ./currentuid

# Use sed to remove all 1, 2, and 3 digit UIDs from currentuid, and
# place the output in the temp file currentuid2
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2

# Use sed to remove UIDs 60001, 60002, and 65534 from currentuid2,
# and place the output in the temp file currentuid3
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3

# Sort the UIDs in currentuid3 numerically
sort -n ./currentuid3 > ./currentuid4
```

```
# Calculate the next available UID
lastuid='sed -n '$p' ./currentuid4'

uid='expr $lastuid + 1'
echo "The uid for $name is: $uid"
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 8

Interactive Scripts

Objectives

Upon completion of this module, you should be able to:

- Use the `print` and `echo` statements to display text
- Use the `read` statement to interactively assign data to a shell variable
- Read user input into one or more variables using one `read` statement
- Use special characters, with `print` and `echo`, to make the displayed text more user friendly
- Use file descriptors to read from and write to multiple files
- Create a *here* document

Input and Output in a Script

Input and Output in a Script

While some scripts run without any interaction with the user, many scripts require input from the user or give output to the user as the script is running.

Advantages of no interaction include:

- The script runs the same way every time.
- The script can run in the background.

Advantages of user interaction include:

- The script is more flexible.
- The user can customize the script as it runs.
- The script can report its progress as it runs.

The Korn Shell print Statement

Use the `print` statement to provide output. Use it wherever you would use the `echo` statement. It is more versatile than the `echo` statement. The `print` statement has several options:

- n Suppresses the newline character after printing the message. This usually is used when printing a prompt for user input.
- r Turns off the special meaning of the \ character.
- R Does not interpret the - that follows as an option to the `print` statement, except if -R is followed by n; that is, if the -n option follows -R, -n is still taken as an option. The -R option is useful if you need to print negative numbers.
- Same as -R, except that a following -n option is taken literally.

The `print` statement also interprets the following as special characters when they are enclosed in double quotes:

- \n Prints a newline character, which enables you to print a message on several lines using one `print` statement
- \t Prints a tab character, which is useful when creating tables or a report
- \a Rings the bell on the terminal, which draws the attention of the user
- \b Specifies a backspace character, which overwrites the previous character

Note – Provide the special characters to the `print` statement within double quotes. The statements `print \n` and `print "\n"` do not give the same results.

Examples of Using the print Statement

Examples of Using the print Statement

The following are examples of the print options and special characters displayed on the command line. Experiment with these to see how they work.

```
$ print "Hello there.\nHow are you?"
```

```
Hello there.
```

```
How are you?
```

```
$ print -r "Hello there.\nHow are you?"
```

```
Hello there.\nHow are you?"
```

```
$ print "-2 was the temperature this morning."
```

```
ksh: print: bad option(s)
```

```
$ print -R "-2 was the temperature this morning."
```

```
-2 was the temperature this morning.
```

```
$ print -- "-2 was the temperature this morning."
```

```
-2 was the temperature this morning.
```

```
$ print -- -n "is the option."
```

```
-n is the option.
```

```
$ print -R -n "is the option."
```

```
is the option.$
```

```
$ print -n "No newline printed here. "
```

```
No newline printed here. $
```

```
$ print "Hello\tout\tthere!"
```

```
Hello    out      there!
```

```
$ print "\aListen to me!"
```

```
<bell rings>Listen to me!
```

```
$ print "Overwrite\b the 'e' in 'Overwrite'."
```

```
Overwrit the 'e' in 'Overwrite'.
```

Examples of Using the echo Statement

The following are examples of the echo options and special characters displayed on the command line. Experiment with these to see how they work. These examples show that you have an alternative to the Korn shell-specific print statement. Although it looks as though the last example does not work, this is only because the shell overwrites the closing square bracket when the next command line prompt is displayed.

```
$ echo "Hello there.\nHow are you?"  
Hello there.  
How are you?  
  
$ echo "Hello there.\nHow are you?"  
Hello there.\nHow are you?"  
  
$ echo "-2 was the temperature this morning."  
-2 was the temperature this morning.  
  
$ echo "No newline printed here. \c"  
No newline printed here. $  
  
$ echo "Hello\tout\tthere!"  
Hello      out      there!  
  
$ echo "\007Listen to me!"  
<bell rings>Listen to me!  
  
$ echo "Overwrite\b the 'e' in 'Overwrite'."  
Overwrit the 'e' in 'Overwrite'.  
  
$ echo "Type a letter [ ]\b\b\c"  
Type a letter [$
```

The `read` Statement

The counterpart to the `print` or `echo` statement is the `read` statement. The `read` statement reads input typed in by the user from the keyboard, or it reads a line from a file.

- The input is parsed (broken down) into *tokens*.
A token is any group of consecutive characters that do not contain white spaces. Thus, the first token begins with the first non-white-space character and ends when a white-space character is encountered. The second token begins with the next non-white-space character and ends with the first white-space character thereafter, and so on. White space can be included if quotes are used around the token.
- The `read` statement continues reading characters until a newline character is encountered.
- The `read` statement uses the contents of the `IFS` variable as token delimiters. By default, the values are white space; however, you can change this to include other characters, such as commas, colons, or other field separators.
- The first token is saved in the first variable name following the `read` statement, the second token is saved in the second variable, and so on. If there are more tokens than variables supplied to `read`, the last variable holds all remaining tokens.

Note – If there are more variables than tokens, the extra variables have no value assigned to them; however, they still exist as valid variables in the shell.

- If there are no variable names supplied to the `read` statement, the Korn shell puts the values read in the `REPLY` variable. The Bourne shell returns an error.

Examples of Using the `read` Statement

In the following example, the number of variables supplied to `read` matches the number of tokens typed on the command line.

```
$ read var1 var2 var3  
abc def ghi  
  
$ echo $var1  
abc  
  
$ echo $var2  
def  
  
$ echo $var3  
ghi
```

In the following example, the number of variables supplied to `read` is less than the number of tokens typed on the command line.

```
$ read num string junk  
134 bye93;alk the rest of the line is put in 'junk'  
  
$ echo $num  
134  
  
$ echo $string  
bye93;alk  
  
$ echo $junk  
the rest of the line is saved in 'junk'
```

In the following example, the number of variables supplied to `read` is more than the number of tokens typed on the command line.

```
$ read token1 token2  
one  
  
$ echo $token1  
one  
  
$ echo $token2
```

Examples of Using the `read` Statement

If variable names are not supplied to `read` when using Korn shell, it populates the `REPLY` variable with the user input, as in the following example:

```
$ read  
What is this saved in?  
  
$ echo $REPLY  
What is this saved in?
```

If variable names are not supplied to `read` when using Bourne shell, it gives an error message, as in the following example:

```
$ read  
read: missing arguments
```

Capturing a Command Result

Use the `read` statement to capture a command result. To assign the command result to a variable:

```
var=`ls -l /etc/passwd`
```

You can capture the command output into a file and then use the `read` statement to assign the values to variables as shown below:

```
$ ls -l /etc/passwd > file  
$ read a b c d e f g h < file  
$ echo $a  
-rw-r--r--  
$ echo $b  
1  
$ echo $h  
16:20 /etc/passwd
```

Alternatively, you can pipe the output of a command to the `read` statement:

```
# ls -l /etc/passwd | read x y z p q r  
# echo $x  
-rw-r--r--  
# echo $y  
1  
# echo $z  
root
```

Printing a Prompt

This example shows how to prompt a user for some input using a \c in the output string. A read statement obtains the input. The input is read and saved in variables using the read statement. An extra variable, junk, is supplied with both read statements to pick up any extra input by the user. The input is used in a message that prints to the screen with the echo statement.

```
$ cat io1.sh
#!/bin/sh

# Script name: io1.sh

# This script prompts for input and prints messages
# involving the input received.

echo "Enter your name: \c"
read name junk

echo "Hi $name, how old are you? \c"
read age junk

echo "\n\t$age is an awkward age, $name,"
echo "    You're too old to depend on your parents,"
echo "and not old enough to depend on your children."

$ ./io1.sh
Enter your name: Murdock.
Hi Murdock, how old are you? 25

        25 is an awkward age, Murdock.
        You're too old to depend on your parents,
        and not old enough to depend on your children.
```

Printing a Prompt

Similarly, you could prompt for input using the print statement with the -n option. The -n option leaves the cursor at the end of the text line.

```
$ cat io2.ksh
#!/bin/ksh

# Script name: io2.ksh

# This script prompts for input and prints messages
# involving the input received.

print -n "Enter your name: "
read name junk

print -n "Hi $name, how old are you? "
read age junk

print "\n\t$age is an awkward age, $name,"
print "    You're too old to depend on your parents,"
print "and not old enough to depend on your children."

$ ./io2.ksh
Enter your name: Murdock.
Hi Murdock, how old are you? 25

      25 is an awkward age, Murdock.
      You're too old to depend on your parents,
      and not old enough to depend on your children.
```

Prompting for Input – Korn Shell Shortcut

When you use one variable with the `read` statement, you can put the prompt for the input within the `read` statement itself. The syntax is:

```
read var?"prompt"
```

Note – There is no space between the variable name and the ?, nor between the ? and the prompt string that is enclosed within quotes.

Although this prompting facility is available in the Korn shell, it's better to use either `echo` or `print` to display a prompt message. The `echo` and `print` statements are more flexible because they can display many text lines as one output string. It's also more difficult to debug a script when combining the `read` statement with a prompt string in one statement line. The following example shows how to put the prompts into `read` statements instead of using `print` statements. The output from running the script is the same as the output from the execution of the script `io2.ksh` script.

```
$ cat io3.ksh
#!/bin/ksh

# Script name: io3.ksh

# This script prompts for input and prints messages
# involving the input received.

read name?"Enter your name: "

read age?"Hi $name. How old are you? "

print "\n\t$age is an awkward age, $name,"
print "    You're too old to depend on your parents,"
print "and not old enough to depend on your children.

$ ./io3.ksh
Enter your name: Murdock
Hi Murdock. How old are you? 25

25 is an awkward age, Murdock.
You're too old to depend on your parents,
and not old enough to depend on your children.
```

Prompting for Input – Korn Shell Shortcut

When you put the prompt for input into the read statement, you can supply an extra variable to pick up extra input by the user, for example:

```
$ read name?" Enter your name: "
Enter your name: John Murdock
```

```
$ print $name
John Murdock
```

```
$ read name?" Enter your name: " more
Enter your name: John Murdock
```

```
$ print $name
John
```

```
$ print $more
Murdock
```

File Input and Output

A script can receive input from a file and send output to a file so that the script can run without user interaction. Later, a user can review the output file, or another script can use the output file as its input.

File input and output are accomplished in the shell by integer handles that the kernel uses to track all open files in a process. These numeric values are *file descriptors*. The best known file descriptors are 0 (`stdin`), 1 (`stdout`), and 2 (`stderr`).

The numbers 3 through 9 are for programmer-defined file descriptors. Use these to associate numeric values to path names. In a program, if there is a need to read or write multiple times to the same file, a shorthand file descriptor value might reduce errors in referencing the file name. Also, a change to the file name must be done only once if the file is subsequently accessed through the file descriptor. Table 8-1 shows the syntax for file redirection.

Table 8-1 Redirection in the Shell

Command	Description
<code>< file</code>	Takes standard input from <i>file</i>
<code>0< file</code>	Takes standard input from <i>file</i>
<code>> file</code>	Puts standard output to <i>file</i>
<code>1> file</code>	Puts standard output to <i>file</i>
<code>2> file</code>	Puts standard error to <i>file</i>
<code>exec fd> /some/filename</code>	Assigns the file descriptor <i>fd</i> to <code>/some/filename</code> for output
<code>exec fd< /some/filename</code>	Assigns the file descriptor <i>fd</i> to <code>/some/filename</code> for input
<code>read <&fd var1</code>	Reads from the file descriptor <i>fd</i> and stores into variable <i>var1</i>
<code>cmd >& fd</code>	Executes <i>cmd</i> and sends output to the file descriptor <i>fd</i>
<code>exec fd<&-</code>	Closes the file descriptor <i>fd</i>

User-Defined File Descriptors

User-Defined File Descriptors

You can use a file descriptor to assign a numeric value to a file instead of using the file name.

The `exec` command is one of the built-in commands in a shell. Use this command to assign a file descriptor to a file. The syntax is:

```
exec fd> filename  
exec fd< filename
```

No spaces are allowed between the file descriptor number (*fd*) and the redirection symbol (> for output, < for input).

After a file descriptor is assigned to a file, you can use the descriptor with the shell redirection operators. On output, if the file does not exist, it is created. If it does exist, it is emptied. On input, if the file does not exist, an error occurs.

```
command >&fd  
command <&fd
```

The file descriptor assigned to a file is valid in the current shell only.

File Descriptors in the Bourne Shell



Discussion – In the following example, the syntax for reading and writing using file descriptors uses the Bourne syntax for the `read` and `echo` statements. This syntax works with both shells.

Try doing the following:

1. Copy the `/etc/hosts` file to the `/tmp/hosts2` file.
2. Use `grep` to read the `/tmp/hosts2` file, strip out the comment lines, and send the output to the `/tmp/hosts3` file.
3. Assign file descriptor 3 to the `/tmp/hosts3` file for input, and then each `read` statement issued to file descriptor 3 reads a record from the file. The statement used to associate file descriptor 3 to the input file named `/tmp/hosts3` is:
`exec 3< /tmp/hosts3`
4. Assign file descriptor 4 to the `/tmp/hostsfinal` file for output. If the output file does not exist, it is created. If it does exist, its size truncates to 0 bytes. The statement that associates file descriptor 4 to the output file is:
`exec 4> /tmp/hostsfinal`
5. Read the `/tmp/hosts3` file and output two fields to the `/tmp/hostsfinal` file.

Reading from the input file is accomplished by specifying the file descriptor number to the `<&` argument of the `read` statement. This causes a line of the input file to be read for each `read` operation beginning with the first line, and continuing sequentially.

Write text to the output file by specifying the file descriptor number to the `>&` argument of the `print` statement. This causes the output to be written as a line in the output file. The output file is written sequentially.

6. Close the input file when it is no longer needed. This is good practice. The OS closes the file automatically when the process terminates if the program fails to close it.

File Descriptors in the Bourne Shell

7. When all writes to the output file are complete, close the file.

```
$ cat readex.sh
#!/bin/sh

# Script name: readex.sh

##### Step 1 - Copy /etc/host
cp /etc/hosts /tmp/hosts2

##### Step 2 - Strip out comment lines
grep -v '^#' /tmp/hosts2 > /tmp/hosts3

##### Step 3 - fd 3 is input file /tmp/hosts3
exec 3< /tmp/hosts3

##### Step 4 - fd 4 is output file /tmp/hostsfinal
exec 4> /tmp/hostsfinal

##### The following 4 statements accomplish STEP 5

read <& 3 addr1 name1 alias      # Read from fd 3
read <& 3 addr2 name2 alias      # Read from fd 3

echo $name1 $addr1 >& 4          # Write to fd 4 (do not write aliases)
echo $name2 $addr2 >& 4          # Write to fd 4 (do not write aliases)

##### END OF STEP 5 statements

exec 3<&-
                                         # Step 6 - close fd 3

exec 4<&-
                                         # Step 7 - close fd 4

$ ./readex.sh
```

```
$ more /tmp/hosts2
#
# Internet host table
#
127.0.0.1      localhost
192.9.200.111   ultrabear        loghost
192.9.200.121   ladybear

$ more /tmp/hosts3
127.0.0.1      localhost
192.9.200.111   ultrabear        loghost
192.9.200.121   ladybear

$ more /tmp/hostsfinal
localhost 127.0.0.1
ultrabear 192.9.200.111
```

Korn Shell File Descriptors

Korn Shell File Descriptors

The following example shows how the previous example would be written in the Korn shell. In the Korn shell, the `read` statement uses a `-u fd` syntax rather than the `<& fd` syntax.

The `print` statement uses a `-u fd` syntax to direct the output to file descriptor 4 in the Korn shell.

```
$ cat readex.ksh
#!/bin/ksh

# Script name: readex.ksh

##### Step 1 - Copy /etc/host
cp /etc/hosts /tmp/hosts2

##### Step 2 - Strip out comment lines
grep -v '^#' /tmp/hosts2 > /tmp/hosts3

##### Step 3 - fd 3 is an input file /tmp/hosts3
exec 3< /tmp/hosts3

##### Step 4 - fd 4 is output file /tmp/hostsfinal
exec 4> /tmp/hostsfinal

##### The following 4 statements accomplish STEP 5

read -u3 addr1 name1 alias      # read from fd 3
read -u3 addr2 name2 alias      # read from fd 3

print -u4 $name1 $addr1          # write to fd 4 (do not write aliases)
print -u4 $name2 $addr2          # write to fd 4 (do not write aliases)

##### END OF STEP 5 statements

exec 3<&-
                           # Step 6 - close fd 3

exec 4<&-
                           # Step 7 - close fd 4
```

```
$ ./readex.ksh

$ more /tmp/hosts2
#
# Internet host table
#
127.0.0.1      localhost
192.9.200.111   ultrabear        loghost
192.9.200.121   ladybear

$ more /tmp/hosts3
127.0.0.1      localhost
192.9.200.111   ultrabear        loghost
192.9.200.121   ladybear

$ more /tmp/hostsfinal
localhost 127.0.0.1
ultrabear 192.9.200.111
```

The “here” Document

The “here” Document

Frequently, a script might call on another script or utility that requires input. To run a script without operator interaction, you must supply that input within your script. The *here* document provides a means to do this. The syntax for the *here* document is:

```
command << Keyword
input1
input2
...
Keyword
```

For example:

```
$ cat termheredoc.ksh
#!/bin/ksh

# Script name: termheredoc.ksh

print "Select a terminal type"
cat << ENDINPUT
sun
ansi
wyse50
ENDINPUT

print -n "Which would you prefer? "
read termchoice

print
print "Your choice is terminal type: $termchoice"

$ ./termheredoc.ksh
Select a terminal type
sun
ansi
wyse50
Which would you prefer? sun

Your choice is terminal type: sun
```

Note – All lines of the *here* document must be left-justified. Do not use leading spaces. The ending keyword must be on a line by itself.

```
#!/bin/ksh

# This script automates installing the SUNWaudio software package.
#
# The assumption is the answers to the installation questions have
# been documented by performing an actual install. Running the
# script command prior to doing a package installation would
# allow for such documentation.
#
# This software package only asks two questions:
# Do you want to install these conflicting files [y,n,?,q]
# Do you want to continue with the installation of <SUNWaudio> [y,n,?]
#
# In each case the answer we will give is y for yes. Hence the two
# lines containing y in the here document.

print "About to install the SUNWaudio package."
pkgadd -d spool SUNWaudio << ENDINPUT
Y
Y
ENDINPUT
```

Exercise: Using Interactive Scripts

Exercise: Using Interactive Scripts

Exercise objective – Write one or more of the shell scripts specified in the following tasks.

One task is to write a backup script that takes a file system as a command-line argument. This script asks the user to enter the `ufsdump` level (full or incremental) that should be specified in the command.

Another task is to continue writing the `adduser` script begun in an earlier lab exercise. The script should now prompt for the user's primary group, login shell, and comment information.

Preparation

Refer to the lecture notes as necessary to perform the following tasks.

Change the directory to `mod8/lab` before beginning the exercise.

When writing the script involving the file system backup, see the man page for `ufsdump` if you need syntax help.

When writing the script involving adding a new user, look at the `/etc/passwd` file, or read the man page on the `/etc/passwd` file (`man passwd.4`) if you are unsure of the field order.

When writing the script involving adding a new user, read the exercise from the last module, or your solution to it, to remind yourself what that script version accomplished.

Tasks

Write a script named `backupscript` that automates the `ufsdump` command so that the person who executes the script always uses the desired options. Implement the following features in the script. Write each step, and then test that portion of the code, rather than writing the entire script all at once.

Note – Write the backup to the file `/dev/null` file, which saves time and obviates the need to use physical media. Back up something small, such as the `/export/home` file.

The script should:

- a. Take exactly one argument that is assumed to be the file system name. If the test is not equal to one, output a `USAGE` message and exit.
- b. Determine if the name entered is a valid directory; if not, print output that says it is an unknown file system, and exit.
- c. If the name entered is a valid directory, do the following:
 1. Output the name of the file system as confirmation.
 2. Prompt the user to input the desired full or incremental backup level number. (You might want to list these for the user, perhaps as a *here* document.)
 3. Read the level number entered by the user, and test it to ensure it is in the proper range.
 4. If the level number is out of range, print output that says it is an invalid backup level, and then exit.
 5. If the level number is in range, show the user the backup command that will be issued, and prompt the user to indicate if the user is ready to have the script perform the backup of the file system. (Hint: Because the backup level is stored in a variable (`LEVEL`) and you want to include that variable as one of the options, enclose the variable name in {} to shield it from the option that follows; for example, `-${LEVEL}f`.)
 6. If the user enters a `y`, meaning the user wants to proceed with the backup, execute the `ufsdump` command. When it finishes, the script is done.
 7. If the user enters anything other than a `y`, instruct the user to try later when the user is ready.
- d. To test the script, run the script with the following arguments:

Exercise: Using Interactive Scripts

```
$ ./backupscript  
$ ./backupscript /foo  
$ ./backupscript /export/home
```

8. Copy the adduser script from the mod7/lab directory. Now that the user name and UID number have been determined, you continue to gather the remaining information to create an entry for the mypasswd file.

Modify the script to:

- a. Prompt the user to enter a group name, read in the number, and verify that the name entered exists in the /etc/group file.
- b. If the group name is not in the /etc/group file, tell the user that is not a valid group, and exit the script.
- c. Prompt the user for the login shell for the new user. You might want to list the ones you will accept with a *here* document.
- d. Read the requested shell choice.
- e. If the shell choice is acceptable:
 1. Prompt for the comment field information.
 2. Read the comment information.
 3. Assume the login directory for the user is the /export/home/username directory.
 4. Print an entry for the new user using colon delimiters between the seven fields.
 5. Append the new user entry at the end of the mypasswd file.
- f. If the shell requested is not one you expect, tell the user that the shell is not a valid shell, and exit the script.

Hint: You can run the solution to get a sense of the flow of the script. When doing this, put in invalid information if you want to test the error messages.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Task Solutions

Write a script named `backupscript` that automates the `ufsdump` command so that the person who executes the script always uses the desired options. Implement the following features in the script. Write each step, and then test that portion of the code, rather than writing the entire script all at once.

Note – Write the backup to the device `/dev/null`, which saves time and obviates the need to use physical media. Back up something small, such as `/export/home`.

The script should:

- a. Take exactly one argument that is assumed to be the file system name. If the test is not equal to one, output a **USAGE** message and exit.

```
if [ $# -ne 1 ] ; then
    echo "Usage: backupscript filesystem"
    exit 1
fi
```

- b. Determine if the name entered is a valid directory; if not, print output that says it is an unknown file system, and exit.

```
if [ -d $1 ] ; then
...
else
    echo "Unknown filesystem, exiting..."
    exit 4
fi
```

- c. If the name entered is a valid directory, do the following:

1. Output the name of the file system as confirmation.

```
if [ -d $1 ] ; then
    echo "$1 is the filesystem to be backed up."
```

1. Prompt the user to input the desired full or incremental backup level number. (You might list these for the user, perhaps as a *here* document.)

```
echo "Select a ufsdump level"
cat << ENDDUMPLEVELS
> 0 # which is a full backup
> 1
> 2
> 3
> 4
> 5
```

```
> 6
> 7
> 8
> 9
ENDDUMPLEVELS

2. Read the level number entered by the user, and test it to ensure it
is in the proper range.

read LEVEL

3. If the level number is out of range, print output that says it is an
invalid backup level, and then exit.

if [ $LEVEL -ge 0 -a $LEVEL -le 9 ] ; then
...
else
    echo "Invalid backup level, exiting..."
    exit 2
fi

4. If the level number is in range, show the user the backup
command that will be issued, and prompt the user to ask if she is
ready to have the script perform the backup of the file system.
(Hint: Because the backup level is stored in a variable (LEVEL)
and you want to include that variable as one of the options,
enclose the variable name in {} to shield if from the option that
follows; for example, ${LEVEL}f.)
```

```
if [ $LEVEL -ge 0 -a $LEVEL -le 9 ] ; then
    echo "Ready to do backup with the following command:"
    echo "  ufsdump ${LEVEL}uf /dev/null \$1"
    echo "Are you ready to do the backup now? [y, n]\c "
else
```

Task Solutions

5. If the user enters a y meaning they want to proceed with the backup, execute the ufsdump command. When it finishes, the script is done.

```
read ANSWER
if [ $ANSWER = "Y" ]; then
    ufsdump ${LEVEL}uf /dev/null $1
else

```

6. If the user enters anything other than a y, instruct the user to try later.

```
if [ $ANSWER = "Y" ]; then
...
else
    echo "You did not input a y, try later when you are ready."
    exit 3
fi
```

- d. To test the script, run the script with the following arguments:

```
$ ./backupscript
$ ./backupscript /foo
$ ./backupscript /export/home
```

The following is the full solution for Step 1:

```
$ cat backupscript.sh
#!/bin/sh

# Purpose - To ensure that the correct options are always used when
# performing a system backup with the ufsdump command.
#
# This script takes one argument which is assumed to be a
# filesystem name, it uses the u option to update the
# /etc/dumpdates file, and it asks the user which dump
# level number should be used.
#
# Name: backupscript.sh
#
if [ $# -ne 1 ]; then
    echo "Usage: backupscript filesystem"
    exit 1
fi

if [ -d $1 ]; then
    echo "$1 is the filesystem to be backed up."
    echo "Select a ufsdump level"
```

```

cat << ENDDUMPLEVELS
> 0 # which is a full backup
> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
ENDDUMPLEVELS
read LEVEL
if [ $LEVEL -ge 0 -a $LEVEL -le 9 ]; then
    echo "Ready to do backup with the following command:"
    echo "  ufsdump ${LEVEL}uf /dev/null \$1"
    echo "Are you ready to do the backup now? [y, n]\c "
    read ANSWER
    if [ $ANSWER = "y" ]; then
        ufsdump ${LEVEL}uf /dev/null \$1
    else
        echo "You did not input a y, try later when you are ready."
        exit 3
    fi
else
    echo "Invalid backup level, exiting..."
    exit 2
fi
else
    echo "Unknown filesystem, exiting..."
    exit 4
fi

```

2. Copy the adduser script from the mod7/lab directory. Now that the user name and UID number are determined, continue to gather the remaining information to create an entry for the mypasswd file.

\$ cp ../../mod7/lab/adduser .

Modify the script to:

- a. Prompt the user to enter a group name read in the number, and verify that the name entered exists in the /etc/group file.

```

echo "What primary group should $a belong to? "
echo "Please specify the group name: \c"
read group junk

```

Task Solutions

- b. If the group name is not in the /etc/group file, tell the user that is not a valid group, and exit the script.

```
grep "^\$group:" /etc/group > /dev/null 2>&1
if [ $? -ne 0 ]
then
    echo "The group $group is not valid."
    echo
    exit 2
fi
```

- c. Prompt the user for the login shell for the new user. You might want to list the ones you will accept with a *here* document.

```
echo "Good, now select a login shell for user $1"
cat << LOGINSHLLS
> csh
> ksh
> sh
LOGINSHLLS
```

- d. Read the requested shell choice.

```
read ushell
```

- e. If the shell choice is acceptable:

```
if [ $ushell = "csh" -o $ushell = "ksh" -o $ushell = "sh" ]
    1. Prompt for the comment field information.
echo "Please enter the comment information for the user:"
    2. Read the comment information.
read comment
    3. Assume the login directory for the user is the
        /export/home/username directory.
    4. Print an entry for the new user using colon delimiters between the
        seven fields.
echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell"
    5. Append the new user entry at the end of the mypasswd file.
echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell" >> ./mypasswd
```

- f. If the shell requested is not one you expect, tell the user that the shell is not a valid shell, and exit the script.

Hint: You can run the solution to see the script flow. When doing this, put in invalid information if you want to test the error messages.

```
else
    echo "Sorry, that is not a valid shell, exiting..."
    exit 3
```

The following is the full solution for Step 2:

```
##!/bin/sh
# Purpose: To write a script to add user to the system.
# Name: adduser.sh

# Check for only a single command line argument
if [ $# -ne 1 ]
then
    echo "Usage: adduser newusername"
    exit 1
fi

# Assign the value of $1 to the variable name
name=$1

if grep "^\$name:" ./mypasswd > /dev/null 2>&1
then
    echo "The username $1 is already in use, here "
    echo "is the entry from the ./mypasswd file."
    echo
    grep "^\$name:" ./mypasswd
    exit 2
fi

# Use awk to extract all the UIDs into the temp file currentuid
awk -F: '{print $3}' ./mypasswd > ./currentuid

# Use sed to remove all 1, 2, and 3 digit UIDs from currentuid, and
# place the output in the temp file currentuid2
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2

# Use sed to remove UIDs 60001, 60002, and 65534 from currentuid2,
# and place the output in the temp file currentuid3
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3

# Sort the UIDs in currentuid3 numerically
```

Task Solutions

```
sort -n ./currentuid3 > ./currentuid4

# Calculate the next available UID
lastuid='sed -n '$p' ./currentuid4'

uid='expr $lastuid + 1'
echo "The uid for $name is: $uid"

# Prompt for a group name and verify whether it exists in /etc/group
echo "What primary group should $a belong to? "
echo "Please specify the group name: \c"
read group junk

grep "^\$group:" /etc/group > /dev/null 2>&1
if [ $? -ne 0 ]
then
    echo "The group $group is not valid."
    echo
    exit 2
fi

echo "Good, now select a login shell for user $1"
cat << LOGINSHELLS
> csh
> ksh
> sh
LOGINSHELLS

read ushell

if [ $ushell = "csh" -o $ushell = "ksh" -o $ushell = "sh" ]
then
    echo "Please enter the comment information for the user:"
    read comment

    echo "The new user entry looks like this:"
    echo "$1:x:$uid:$group:$comment:/export/home/$1:/bin/$ushell"
    echo " "
    echo "New user $1 has been added to the ./mypasswd file"
    echo "$1:x:$uid:$group:$comment:/export/home/$1:/bin/$ushell" >>
./mypasswd
else
    echo "Sorry, that is not a valid shell, exiting..."
    exit 3
fi
```

Module 9

Loops

Objectives

Upon completion of this module, you should be able to:

- Write scripts that use `for`, `while`, and `until` loops
- Write a script using the `select` statement
- Describe when to use loops within a script
- Generate argument lists using command, variable, and file-name substitution

Shell Loops

Shell Loops

There are many times in a shell script when you want to repeatedly execute a statement or group of statements until some specified condition occurs. In addition, you might want to execute a group of statements for each element in a certain list. The shell provides three looping constructs for these situations: the `for`, `while`, and `until` constructs.

The `for` loop allows you to specify a list of values. A group of statements is executed for each value in the list.

The `while` and `until` loops continually execute a group of statements until a specified condition becomes false or true, respectively.

The following sections describe each of these loops in detail.

The for Loop Syntax

The `for` loop in the shell takes a list of words (strings) as an argument. The number of words in the list determines the number of times the statements in the `for` loop are executed. The syntax for the `for` loop is:

```
for var in argument_list ...
do
    statement1
    ...
    statementN
done
```

where:

- `var` is any variable name.
- `argument_list` can be any list of words, strings, or numbers, and it can be literal or generated by using a shell command or shell-command-line metacharacter.
- The statements are any operating-system commands, a user program, a shell script, or a shell statement that returns (produces) a list

The value of the variable `var` is set to the first word in the list the first time through the loop. The second time through the loop, its value is set to the second word in the list, and so on. The loop terminates when `var` has taken on each of the values from the argument list, in turn, and there are no remaining arguments.

The for Loop Argument List

The for Loop Argument List

The argument list in a `for` loop can be any list of words, strings, or numbers. Generate the argument list using any of the following methods (or combination of methods):

- Use an explicit list
- Use the contents of a variable
- Use command-line arguments
- Use command substitution
- Use file names in command substitution
- Use file-name substitution

Using an Explicit List to Specify Arguments

When arguments are listed for a `for` loop, they are called an *explicit list*. An explicit list is the simplest form of an argument list. The loop is executed once for each list element.

```
for var in arg1 arg2 arg3 arg4 ... argn
```

The following example is a `for` loop using an explicit list of elements:

```
for fruit in apple orange banana peach kiwi
do
print "Value of fruit is: $fruit"
done
```

The output of this `for` loop is:

```
Value of fruit is: apple
Value of fruit is: orange
Value of fruit is: banana
Value of fruit is: peach
Value of fruit is: kiwi
```

Using Variable Contents to Specify Arguments

When the arguments are in a variable and the statement is executed, the variable contents are substituted. If the variable is empty, the loop exits.

```
for var in $var_sub
```

In the following example, the text entered at the prompt becomes the value of the variable `INPUT`. This variable represents the argument list of the `for` construct. Therefore, the text in `INPUT` is broken into words or tokens based on white space.

```
$ cat ex1.sh
#!/bin/sh

# Script name: ex1.sh

echo "Enter some text: \c"
read INPUT

for var in $INPUT
do
    echo "var contains: $var"
done

$ ./ex1.sh
Enter some text: I like the Korn shell.
var contains: I
var contains: like
var contains: the
var contains: Korn
var contains: shell.
```

Using Command-Line Arguments to Specify Arguments

In the following example, the text entered on the command line becomes the argument list of the `for` construct. Therefore, the command line text is broken into words or tokens based on white space.

```
for var in (the_command-line_arguments)
$ cat ex2.sh
#!/bin/sh

# Script name: ex2.sh

for var in $*
do
    echo "command line contains: $var"
done

$ ./ex2.sh The Bourne shell is good too.
command line contains: The
command line contains: Bourne
command line contains: shell
command line contains: is
command line contains: good
command line contains: too.
```

Using Command Substitution to Specify Arguments

The syntax for command substitution in a Korn shell is:

```
for var in $(cmd_sub)
```

The syntax for command substitution in a Bourne or Korn shell is:

```
for var in `'cmd_sub'`
```

The following example uses the output of the `cat` command as the argument list.

```
$ cat fruit1
apple
orange
banana
peach
kiwi

$ cat ex3.ksh
#!/bin/ksh

# Script name: ex3.ksh

for var in $(cat fruit1)
do
    print "$var"
done

$ ./ex3.ksh
apple
orange
banana
peach
kiwi
```

The following example also uses the output of the `cat` command as the argument list, but it does so in a Bourne-shell syntax.

```
$ cat fruit2
Apple
Orange
Banana
Peach
Kiwi
```

The for Loop Argument List

```
$ cat ex4.sh
#!/bin/sh

# Script name: ex4.sh

for var in `cat fruit2`
do
    echo "var contains: $var"
done

$ ./ex4.sh
var contains: Apple
var contains: Orange
var contains: Banana
var contains: Peach
var contains: Kiwi
```

If the file containing the list was in the following format, what is the result of the for loop.

```
$ cat fruit3
Apple Orange Banana Peach Kiwi

$ cat ex5.ksh
#!/bin/ksh

# Script name: ex5.ksh

for var in $(cat fruit3)
do
    print "$var"
done
```

In the above example, the results do not change. The IFS variable is any white space, so both a carriage return and a space or tab separate each field in the file.

If the file containing the list was in the following format, what would you need to do to use each item as a separate argument?

```
$ cat fruit4
apple:orange:banana:peach:kiwi

$ cat ex6.ksh
#!/bin/ksh

# Script name: ex6.ksh

for var in $(cat fruit4)
do
    print "$var"
done

$ ./ex6.ksh
apple:orange:banana:peach:kiwi
```

In the above example, you must set the IFS variable to include the colon.

Using File Names in Command Substitution to Specify Arguments

Some commands provide file names and directory names as their output. In the following example, the shell substitutes the output of the command, ls /etc/p* (/etc/passwd /etc/profile and so on), as the argument list for the for loop.

```
$ cat ex7.ksh
#!/bin/ksh

# Script name: ex7.ksh

for var in $(ls /etc/p*)
do
    print "var contains: $var"
done
```

The for Loop Argument List

The for.ksh example script uses the output of an ls command to create the argument list in the for statement.

```
$ cat for.ksh
#!/bin/ksh

# Script name: for.ksh

print "Subdirectories in $(pwd) :"

for fname in $(ls)
do
    if [[ -d $fname ]]
    then
        print $fname
    fi
done
$ cd /usr/share

$ ../../mod9/examples/for.ksh
Subdirectories in /usr/share:
lib
man
src

$ cd

$ ./for.ksh
Subdirectories in /home/user200:
filesub
functions
```

Using File-Name Substitution to Specify Arguments

The ls command provides a file and directory list. You can use the output of the ls command as the argument list for the for loop. Additionally, there is a syntax for specifying just files and directories.

```
for var in file_list
```

In the following example, the shell substitutes the file names that match /etc/p* as the argument list.

```
ls /etc/p*
/etc/passwd /etc/profile /etc/prtvtoc
...
for var in /etc/p*
```

File-Name Metacharacters

Metacharacters are frequently used when specifying file names as the argument list; for example:

```
for var in a* ab??? s* [m-z]
```

If you use file-name metacharacters to generate some or all of a list of words and there are no files that match the pattern given, the shell does not expand the pattern. That is, if no file names match the metacharacter pattern given in the list in a `for` statement, the metacharacter pattern is taken literally as a value in the list.

Consider the following example:

```
for x in z*
do
    print "Working on file $x."
    <other statements>
done
```

The previous example is intended to work on files in the current directory that begin with the letter `z`; however, if the current directory does not contain any files beginning with the letter `z`, then there is only one element in the list for the `for` loop: `z*`. The statements in the `for` loop are executed only once, and the `print` statement prints:

```
Working on file z*.
```

The statements following the `print` statement result in an error message from the shell if they are statements that use the file `z*` as an argument. No such file exists in the current directory, so the statements fail.

Exercise: Using for Loops

Exercise: Using for Loops

Exercise objective – Write a script with a loop construct.

Preparation

Refer to the lecture notes as necessary to correctly compose the commands.

Change the directory to mod9/lab before beginning the exercise.

Tasks

Write a script that lists all the start or stop scripts in a specified `init.d` directory by doing the following:

1. Use `rcscripts` as the script name.
2. Write the script so that it does not require command-line arguments.
3. For all start scripts in the `/etc/init.d` directory print:
 - The script name
 - The script inode number
 - All other files under `/etc` with the same inode number

Task Solutions

Write a script that lists all the start or stop scripts in a specified `init.d` directory by doing the following:

1. Use `rcscripts` as the script name.
2. Write the script so that it does not require command-line arguments.

```
if [ $# -ne 0 ]; then  
    echo "Usage: rcscripts "  
    exit 1  
fi
```

3. For all start scripts in the `/etc/init.d` directory, print:

```
for sscript in /etc/init.d/*  
do
```

- The script name

```
print "The script name is $sscript"
```

- The inode number of the script

```
print -n "The inode number is "  
inode=`ls -i $sscript | awk '{print $1}'`  
print "$inode \n"
```

- All other files under `/etc` with the same inode number

```
print "The other files under /etc with the same inode number are: "  
find /etc -inum $inode
```

Task Solutions

The following is the complete solution:

```
#!/bin/ksh

# Purpose: This script will list all the start scripts in the /etc/init.d
#           directory. Additionally it will report the inode number for
each
#           script, and then list all files under /etc with the same inode
#           number.
#
# Scriptname: rcscripts.ksh

if [ $# -ne 0 ]; then
    echo "Usage: rcscripts.ksh "
    exit 1
fi

for sscript in /etc/init.d/**
do
    print "The script name is $sscript"

    print -n "The inode number is "
    inode='ls -i $sscript | awk '{print $1}''
    print "$inode \n"

    print "The other files under /etc with the same inode number are:
"
    find /etc -inum $inode
    print "\n\n"
done

echo "Finished"
```

The while Loop

The `while` loop allows you to repeatedly execute a group of statements while a command executes successfully. The syntax for the `while` loop is:

```
while control_command
do
    statement1
    ...
    statementN
done
```

where:

- `control_command` can be any command that exits with a success or failure status.
- The statements in the body of the `while` loop can be any utility commands, user programs, shell scripts, or shell statements.

When a `while` statement is executed, the `control_command` is evaluated. If the `control_command` succeeds, all the statements between the `do` and `done` are executed, and then the controlling command is again executed.

As long as the `control_command` succeeds, the loop body continues to execute.

As soon as the `control_command` fails, the statement following the `done` statement is executed.

The `do` statement is a separate statement and must appear on a line by itself (not on the line containing the `while control_command` statement) unless it is preceded by a semicolon. The same is true of the `done` statement. It must exist on a line of its own or be preceded by a semicolon if it exists at the end of the last statement line in the loop.

The while Loop Syntax

The following examples are all valid forms of the `while` syntax.

While the contents of `$var` are equal to the string `string_value`, the loop continues.

```
while [ "$var" = "string_value" ]
while [[ "$var" == "string_value" ]]
```

The while Loop

In the following example, while the value of \$num is less than or equal to 10, the loop continues.

```
while [ $num -le 10 ]
while (( num <= 10 ))
```

Just as in the `if` statement, the controlling command of a `while` loop is often a `((...))` or `[[...]]` or `[...]` command. Frequently, a variable within the test changes value in the `while` loop so that the loop eventually terminates.

The following two examples show how you can alter numeric values in the loop body to cause the `while` loop to terminate.

```
$ cat whiletest.sh
#!/bin/sh

# Script name: whiletest.sh

num=5

while [ $num -le 10 ]
do
    echo $num
    num=`expr $num + 1`
done

$ cat whiletest.ksh
#!/bin/ksh

# Script name: whiletest.ksh

num=5

while (( num <= 10 ))
do
    echo $num
    (( num = num + 1 ))      # let num=num+1
done
```

There are two commands in the shell that always return the same exit status. These commands are `true` and `false`. The `true` command always causes the test to succeed (with zero errors), and the `false` command always causes the test to fail (with some number of errors). The following example causes an infinite loop.

```
while true
```

To debug your script by forcing the condition to be false, use the following:

```
while false
```

Note – The *null* statement (:) always evaluates to true; therefore, the statements `while :` and `while ((1))` are interchangeable.

Using the `((...))` syntax of the Korn shell, which evaluates numeric expressions, makes it possible for a single numeric variable or value to be present in the `while` test. In this case, if the value of the numeric expression is nonzero, the test succeeds, and the loop executes. If the value of the numeric expression is 0, the test fails, and the code following the loop executes.

```
$ cat whilettest2.ksh
#!/bin/ksh

# Script name: whilettest2.ksh

num=5
while (( num ))
do
    echo $num
    let num=num-1          # let "num = num - 1"
done
```

The statement: `while ((num))` evaluates to the logical negation of `num`. Therefore, the program remains in the loop as long as the value of `num` is not 0.

Example of Using a `while` Loop

In the following example, the variable `num` is initialized to 1, and a `while` loop is entered.

This `while` loop uses `num` instead of `$num` because the `((...))` command automatically does variable expansion.

Using `num` forces integer arithmetic, whereas using `$num` performs arithmetic on strings. Integer arithmetic is faster.

Within the `while` loop, the current value of `num` is printed to `stdout`. The variable `num` is incremented and the condition in the `while` statement is checked again. If the value of the variable `num` did not change within the `while` loop, the program would be in an *infinite* loop (that is, a loop that never ends).

The while Loop

```
$ cat while.ksh
#!/bin/ksh

# Script name: while.ksh

num=1

while (( num < 6 ))
do
    print "The value of num is: $num"
    (( num = num + 1 ))                      # let num=num+1
done

print "Done."

$ ./while.ksh
Value of num is: 1
Value of num is: 2
Value of num is: 3
Value of num is: 4
Value of num is: 5
Done.
```

Keyboard Input

You can place the `read` statement in the `while` condition. Each time a user inputs something from a keyboard, a loop executes. When a user presses Control-D, this tells the `read` statement that input is complete, the condition becomes false, and execution jumps to after the `done` statement.

```
$ cat readinput.ksh
#!/bin/ksh

# Script name: readinput.ksh

print -n "Enter a string: "

while read var
do
    print "Keyboard input is: $var"
    print -n "\nEnter a string: "
done

print "End of input.

$ ./readinput.ksh
Enter a string: OK
Keyboard input is: OK

Enter a string: This is fun
Keyboard input is: This is fun

Enter a string: I'm finished.
Keyboard input is: I'm finished.

Enter a string: ^d End of input.
```

The while Loop

Redirecting Input for a while Loop

Instead of input coming from the keyboard or `stdin`, you can specify a file from which to read the input. Consider the following example:

```
$ cat internalredir.ksh
#!/bin/ksh

# Script name: internalredir.ksh

# set the Internal Field Separator to a colon
IFS=:
while read name number
do
    print "The phone number for $name is $number"
done < phonelist
```

In the example above, the redirection is performed at the end of the `while` block:

```
done < phonelist
```

By redirecting on `stdin`, the file is opened only once. Each time around the loop, `read` returns the next line from the file. Lines are read into the loop while there are lines to read in from the file. After the end of the file is reached, a true state no longer exists, and the loop terminates. After the loop terminates, the statement that follows the `done` statement executes.

 **Note** – Input redirection occurs on the line containing the word `done`. A loop is a complete command structure and any input redirection must come after the complete command structure.

```
$ cat phonelist
Claude Rains:214-555-5107
Agnes Moorehead:710-555-6538
Rosalind Russel:710-555-0482
Loretta Young:409-555-9327
James Mason:212-555-2189

$ ./internalredir.ksh
The phone number for Claude Rains is 214-555-5107
The phone number for Agnes Moorehead is 710-555-6538
The phone number for Rosalind Russel is 710-555-0482
The phone number for Loretta Young is 409-555-9327
The phone number for James Mason is 212-555-2189
```

The until Loop

The until loop is very similar to the while loop, except that the until loop executes as long as the command fails. After the command succeeds, the loop exits, and execution of the script continues with the statement following the done statement. The syntax for the until loop is:

```
until control_command
do
    statement1
    ...
    statementN
done
```

The *control_command* can be any command that exits with a success or failure status. The statements can be any utility commands, user programs, shell scripts, or shell statements. If the *control_command* fails, the body of the loop (all the statements between do and done) execute, and the *control_command* executes again. As long as the *control_command* continues to fail, the body of the loop continues to execute. As soon as the *control_command* succeeds, the statement following the done statement executes; for example:

```
$ cat until.ksh
#!/bin/ksh

# Script name: until.ksh

num=1

until (( num == 6 ))
do
    print "The value of num is: $num"
    (( num = num + 1 ))
done

print "Done."

$ ./until.ksh
The value of num is: 1
The value of num is: 2
The value of num is: 3
The value of num is: 4
The value of num is: 5
Done.
```

The break Statement

The break Statement

The **break** statement allows you to exit the current loop. It is often used in an **if** statement that is contained within a **while** loop, with the condition in the **while** loop always evaluating to true. This is useful if the number of times the loop is executed depends on input from the user and not some predetermined number.

The **break** statement exits out of the *innermost* loop in which it is contained.

```
$ cat break.ksh
#!/bin/ksh

# Script name: break.ksh

typeset -i num=0

while true
do
    print -n "Enter any number (0 to exit): "
    read num junk

    if (( num == 0 ))
    then
        break
    else
        print "Square of $num is $(( num * num )) . \n"
    fi
done

print "script has ended"

$ ./break.ksh
Enter any number (0 to exit): 5
Square of 5 is 25.

Enter any number (0 to exit): -5
Square of -5 is 25.

Enter any number (0 to exit): 259
Square of 259 is 67081.

Enter any number (0 to exit): 0
script has ended
```

The continue Statement

Use the `continue` statement within a loop to force the shell to skip the statements in the loop that occur below the `continue` statement and return to the top of the loop for the next iteration. When you use the `continue` statement in a `for` loop, the variable `var` takes on the value of the next element in the list. When you use the `continue` statement in a `while` or an `until` loop, execution resumes with the test of the `control_command` at the top of the loop.

Example of Using the continue Statement

The following `continue.ksh` example script renames files in the current directory whose names contain a capital letter to lowercase letters.

First, the script prints the name of the file on which it is currently working. If this file name contains no capital letters, the script executes the `continue` statement, forcing the shell to go to the beginning of the `for` loop and get the next file name.

If the file name contains a capital letter, the original name is saved in the `orig` variable. The name is also copied into the `new` variable; however, the line `typeset -l new` at the beginning of the script causes all letters stored in the variable to be converted to lowercase during the assignment. When this is done, the script performs a `mv` command using the `orig` and `new` variables as arguments.

The script then prints a message concerning the name change for the file and gets the next file to be worked on.

When the `for` loop is finished, the `Done` message prints to let the user know the script finished.

The continue Statement

```
$ cat continue.ksh
#!/bin/ksh

# Script name: continue.ksh

typeset -l new # Changes the variable's contents
# to lowercase characters

for file in *
do
    print "Working on file $file..."

    if [[ $file != *[A-Z]* ]]
    then
        continue
    fi

    orig=$file
    new=$file
    mv $orig $new
    print "New file name for $orig is $new."
done

print "Done.

$ cd test.dir
$ ls
Als           a           s0rt.dAtA       slAlk
Data.File     recreate_names  scR1          teXtfile
```

```
$ ./continue.ksh
Working on file Als...
New file name for Als is als.
Working on file Data.File...
New file name for Data.File is data.file.
Working on file a...
Working on file recreate_names...
Working on file s0rt.dAtA...
New file name for s0rt.dAtA is sort.data.
Working on file scR1...
New file name for scR1 is scr1.
Working on file slAlk...
New file name for slAlk is slalk.
Working on file teXtfile...
New file name for teXtfile is textfile.
Done.
$ ls
a           data.file      scr1          sort.data
als         recreate_names  slalk        textfile
```



Discussion – The script does not check if a lowercase version of a file name containing capital letters already exists before executing the mv command. Therefore, it is possible to overwrite an existing file. How could this script be improved to avoid that problem?

The Korn Shell select Loop

The Korn Shell select Loop

The `select` statement in the Korn shell creates a menu. This construct is for the Korn shell only. The syntax for creating a menu is:

```
select var in list
do
    statement1
    ...
    statementN
done
```

The variables `var` and `list` follow the same syntactic rules used in the `for` loop (although the operation of the `select` loop is substantially different). If `list` contains metacharacters, the metacharacters are expanded into file names.

The elements in `list` are the menu choices presented to the user. The Korn shell automatically numbers each menu choice (starting with 1). The user must enter the number corresponding to the user's choice for the input to be valid.

Although you can use any statements within the `select` loop, the `case` statement is most often used to match the choice selected by the user and to take certain actions depending on that choice.

The input the user provides is saved in the `REPLY` variable.

If the user does not enter any input except for pressing the Return key, then the `REPLY` variable is set to the null string, and the shell redisplays the menu choices.

The `select` loop is exited when the user types the end-of-file (EOF) character, which is Control-D on most Sun systems.

The PS3 Reserved Variable

After displaying the list of menu choices for a user, the shell prints a prompt and waits for user input. The prompt value is the value of the `PS3` variable. You can set this variable to any value, but it helps if the value makes sense with regard to the choices that are presented to the user.

Sometimes a script produces so much output — between the time the menu list first displays to when the menu prompt is displayed — that a user is no longer able to see (or remember) the menu choices. To redisplay menu choices, a user presses the Return key in response to the menu prompt.

The `select` syntax creates a loop. A user is repeatedly presented with menu choices followed by a prompt (the value of the `PS3` variable). The `select` loop is exited when a user types the EOF character for the system as the response to the menu prompt. A more graceful way to exit the `select` loop is shown in the “Exiting the `select` Loop” on page 9-29.

The default value of the `PS3` variable in the Korn shell is `#?`. Therefore, if you fail to set the `PS3` variable, the system uses the string `#?` as the prompt and then waits for user input. If this happens, the user might not know that the script is waiting for input.

Example of Using a `select` Loop

The following script gives users five fruit choices and then prompts for their fruit choice.

A `case` statement is used in the `select` loop to take appropriate action on the choice selected by the user. The default case, the `*` pattern, matches any input by the user that is not one of the numbers of the choices on the menu.

The script then executes. Several valid choices and some invalid choices are shown to illustrate how the `select` loop works.

The Korn Shell select Loop

```
$ cat menu.ksh
#!/bin/ksh

# Script name: menu.ksh

PS3="Enter the number for your fruit choice: "

select fruit in apple orange banana peach pear
do
    case $fruit in
        apple)
            print "An apple has 80 calories."
            ;;

        orange)
            print "An orange has 65 calories."
            ;;

        banana)
            print "A banana has 100 calories."
            ;;

        peach)
            print "A peach has 38 calories."
            ;;

        pear)
            print "A pear has 100 calories."
            ;;

        *)
            print "Please try again. Use '1'-'5'"
            ;;

    esac
done
```

```
$ ./menu.ksh
1) apple
2) orange
3) banana
4) peach
5) pear
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 7
Please try again. Use '1'-'5'
Enter the number for your fruit choice: apple
Please try again. Use '1'-'5'
Enter the number for your fruit choice: 1
An apple has 80 calories.
Enter the number for your fruit choice: ^d
```

Exiting the select Loop

Users can exit the select loop at any time by entering the EOF character on their system; however, unless users know this or you include a message explaining this in your menu prompt, users can get stuck in the select loop.

Include a string in your menu choices to help users exit the menu. The string could simply say *Quit Menu*. Then the action in the case statement for this menu choice would be the *break* statement. The *break* statement exits the select loop, and execution of the script moves to the first statement following the *done* statement in the *select/do/done* syntax. If there are no statements following the *done* statement, the script terminates.

The Korn Shell select Loop

```
$ cat menu1.ksh
#!/bin/ksh

# Script name: menu1.ksh

PS3="Enter the number for your fruit choice: "

select fruit in apple orange banana peach pear "Quit Menu"
do
    case $fruit in
        apple)
            print "An apple has 80 calories."
            ;;

        orange)
            print "An orange has 65 calories."
            ;;

        banana)
            print "A banana has 100 calories."
            ;;

        peach)
            print "A peach has 38 calories."
            ;;

        pear)
            print "A pear has 100 calories."
            ;;

        "Quit Menu")
            break
            ;;

        *)
            print "You did not enter a correct choice."
            ;;

    esac
done
```

```
$ ./menu1.ksh
1) apple
2) orange
3) banana
4) peach
5) pear
6) Quit Menu
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 6
```

Submenus

The shell allows any statement within the `select` loop to be another `select` loop. This enables you to create a menu that has one or more submenus.

You usually include a `break` statement in an action for one or more choices on a submenu to exit a submenu loop and return to an outer menu loop.

Because the `PS3` variable is also used for the prompt in any submenu, be careful when moving from one menu to another to reset this variable to the prompt for the menu to which you are moving.

An example illustrating a submenu and the resetting of the `PS3` variable is provided next.

Example of Using Submenus

The following `submenu.ksh` example script first creates two variables, `main_prompt` and `dessert_prompt`, to hold the prompts for the main menu and the dessert submenu, respectively.

The `select` loop for the main menu is exited when the user chooses `Order Completed`, at which time the script prints the message `Enjoy your meal.`

When the user selects the main menu choice `dessert`, a submenu appears, asking the user to input the number for the dessert choice.

The Korn Shell select Loop

In the case statement for the select loop for the dessert menu, the break statement is used for each menu choice. (The default choice does not use the break statement because you want the user to reenter a dessert choice.) This exits the dessert menu. Then the menu prompt resets to the value of the *main_prompt* variable, and control returns to the main (outer) select loop.

All print statements use single quotes so that the \$ in the prices is printed. Also, because of formatting in this course book, several lines in the following script wrap.

```
$ cat submenu.ksh
#!/bin/ksh

# Script name: submenu.ksh

main_prompt="Main Menu: What would you like to order? "
dessert_menu="Enter number for dessert choice: "
PS3=$main_prompt

select order in "broasted chicken" "prime rib" "stuffed lobster" dessert
"Order Completed"
do
case $order in
"broasted chicken") print 'Broasted chicken with baked potato, rolls, and
salad is $14.95.';;
"prime rib") print 'Prime rib with baked potato, rolls, and fresh
vegetable is $17.95.';;
"stuffed lobster") print 'Stuffed lobster with rice pilaf, rolls, and
salad is $15.95.';;
dessert)
PS3=$dessert_menu
select dessert in "apple pie" "sherbet" "fudge cake" "carrot cake"
do
case $dessert in
"apple pie") print 'Fresh baked apple pie is $2.95.'
break;;
"sherbet") print 'Orange sherbet is $1.25.'
break;;
"fudge cake") print 'Triple layer fudge cake is $3.95.'
break;;
"carrot cake") print 'Carrot cake is $2.95.'
break;;
*) print 'Not a dessert choice.';;
esac
done
PS3=$main_prompt;;
"Order Completed") break;;
*) print 'Not a main entree choice.';;
esac
done
print 'Enjoy your meal.'
```

The Korn Shell select Loop

```
$ ./submenu.ksh
1) broasted chicken
2) prime rib
3) stuffed lobster
4) dessert
5) Order Completed
Main Menu: What would you like to order? 3
Stuffed lobster with rice pilaf, rolls, and salad is $15.95.
Main Menu: What would you like to order? 4
1) apple pie
2) sherbet
3) fudge cake
4) carrot cake
Enter number for dessert choice: 3
Triple layer fudge cake is $3.95.
Main Menu: What would you like to order? 5
Enjoy your meal.
```

The `for` and `select` Statements Revisited

The `for` and `select` statements do not have to explicitly supply a list of values. If a list is not provided, the shell assumes the list to be the values of the positional parameters.

If the positional parameters do not have any values, then the `for` loop terminates without executing any statements within its loop. It does not cause an error from the shell.

The following `pospara.ksh` example script is similar to the earlier `pos_par.scr` script up through the `for` loop, although the list in the `for` loop is omitted here.

The omission of the list of values in the `for` loop syntax means that the values of the positional parameters are used for the list. Thus, the result of executing the `for` loop in this script is identical to the execution of the `for` loop in the `pos_par.scr` script (where the values of the positional parameters were explicitly used as the list with the syntax `in $*`).

```
$ cat pospara.ksh
#!/bin/ksh

# Script name: pospara.ksh

set uno duo tres      # resets the value of the positional parameters
print "Executing script $0\n"

print "One, two, three in Latin is:"
for x                  # defaults to "in $*"
do
    print $x
done

$ ./pospara.ksh
Executing script pospara.ksh

One, two, three in Latin is:
uno
duo
tres
```

The shift Statement

The shift Statement

Sometimes a script does not require a specific number of arguments from users. Users are allowed to give as many arguments to the script as they want.

In these situations, the arguments of the script are usually processed in a while loop with the condition being `($#)`. This condition is true as long as the number of arguments is greater than zero. Within the script `$1` and the `shift` statement process each argument.

Because doing a `shift` reduces the argument number, eventually the condition in the while loop becomes false and the loop terminates.

Note – If no number is supplied as an argument to the `shift` statement, the number is assumed to be 1.

The syntax for the `shift` statement is:

```
shift [ num ]
```

All positional parameters are shifted to the left the number of positions indicated by `num` (or by 1 if `num` is omitted).

The number of left-most parameters are eliminated after the `shift`. For example, the statement:

```
shift 4
```

eliminates the first four positional parameters. The value of the fifth parameter becomes the value of `$1`, the value of the sixth parameter becomes the value of `$2`, and so on.

It is an error for `num` to be larger than the number of positional parameters.

Example of Using the shift Statement

The following `shift.ksh` example script contains a **USAGE** message showing that the script should be run with arguments, although the number of arguments given is variable:

```
$ cat shift.ksh
#!/bin/ksh

# Script name: shift.ksh

USAGE="usage: $0 arg1 arg2 ... argN"

if (( $# == 0 ))
then
    print $USAGE
    exit 1
fi

print "The arguments to the script are:"
while (( $# ))
do
    print $1
    shift
done

print 'The value of $* is now:' $*
```

If the user does not provide arguments, the **USAGE** message is printed and the script exits; otherwise, the `while` loop is entered. Each time through the loop, the current value of the positional parameter, `$1`, is printed and the `shift` statement executes.

The `shift` statement reduces the number of arguments by one as it shifts the values of the positional parameters one position to the left. (`$1` is assigned the value of `$2`, `$2` is assigned the value of `$3`, and so on, while the value of `$1` is discarded).

After each argument is printed, the condition of `(($#))` is false, and the `while` loop terminates.

The statement `print $*` shows that nothing is printed after the execution of the `while` loop because the value of `$#` is 0.

The shift Statement

The values of the positional parameters are set to letters of the alphabet using the `set` statement and printed using the statement `print $*` (to show the new values of the positional parameters).

The last two statements in the script, `shift 4` and `print $*`, illustrate what occurs when a larger shift is made.

```
$ ./shift.ksh one two three four  
The arguments to the script are:  
one  
two  
three  
four  
The value of $* is now:
```

Exercise: Using Loops and Menus

Exercise objective – Write scripts that use loops and menus.

Preparation

Two of the tasks involve modifying scripts from previous lab exercises: the `adduser` and `filetype` scripts. If you have not written one of these and want to do the lab in this module, familiarize yourself with the solution.

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change the directory to `mod9/lab` before beginning the exercise.

When writing the scripts on paper you might want to execute commands on the system to ensure you have the correct commands and options to give you the prescribed output.

Tasks

1. Make a copy of the `filetype` script completed in the Module 7, Conditionals, and name the copy `filetype2`.
2. Modify the code so that it accepts one or more path names on the command line and prints a **USAGE** message if it receives no arguments.
3. Use a loop construct to test each path name entered on the command line to determine if it is a file or directory.
4. Copy the `adduser` script from the `mod8/lab/solutions` directory, and modify the script to do the following.
 - a. Create a menu for the list of available shells you want to offer as login shells.
 - b. When the user selects a value not in the menu, print a message stating that is an unknown choice, and then force the user to reenter a value until the user enters a correct menu choice.
 - c. When the user selects a valid shell from the menu, print a verification of which shell was selected. Then allow the program to continue prompting the user for the comment-field information (which was written in a previous lab).

Exercise: Using Loops and Menus

5. Copy the `filetype2` script to `filetype3`, and modify the `filetype3` script as follows:
 - a. Where it uses a `while` loop, change the code to use an `until` loop.
 - b. Where it uses an `until` loop, change the code to use a `while` loop.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

Task Solutions

1. Make a copy of the `filetype` script completed in Module 7, Conditionals, and name the copy `filetype2`.

```
$ cp ../../mod7/lab/solutions/filetype.sh filetype2.sh
```

2. Modify the code so it accepts one or more path names on the command line and prints a **USAGE** message if it receives no arguments.

```
if [ $# -eq 0 ]; then
    echo "Usage: filetype2 pathname [pathname2...]"
    exit 1
fi
```

3. Use a loop construct to test each path name entered on the command line to determine if it is a file or directory.

```
while [ $# -ne 0 ]
do
    if [ -d $1 ]; then
    ...
    else
        echo "$1 is a file."
    ...
done
```

The following is the complete solution:

```
#!/bin/sh

# Purpose: This script will determine whether the pathnames passed
#           on the command line are a file or directory. Then it will
#           print information about the pathname.
#
# Scriptname: filetype2.sh

if [ $# -eq 0 ]; then
    echo "Usage: filetype2.sh pathname [pathname2...]"
    exit 1
fi
while [ $# -ne 0 ]
do
    if [ -d $1 ]; then
        echo "$1 is a directory."
        echo "Following is a listing of the $1 directory."
        ls $1
        echo " "
        echo "The number of kilobytes used by directory $1 is:
\c"
        du -s -k $1 | awk '{print $1}'
    else
        echo "$1 is a file."
        echo "The size of $1 in bytes is: \c"
        ls -l $1 | awk '{print $5}'
        echo
        if [ -b $1 ]; then
            echo "$1 is a block special file."
        elif [ -c $1 ]; then
            echo "$1 is a character special file."
        elif [ -u $1 ]; then
            echo "$1 has the set-user-id permission."
        else
            echo "There is nothing special to say about file
\$1
."
        fi
        shift
        echo "\n\n"
    done
    echo "Finished"
```

4. Copy the adduser script from the mod8/lab/solutions directory, and modify the script to do the following.

```
$ cp ../../mod8/lab/solutions/adduser.sh .  
a. Create a menu for the list of available shells you want to offer as login  
shells.  
select ushell in csh ksh sh  
b. When the user selects a value not in the menu, print a message stating  
that is an unknown choice, and then force the user to reenter a value  
until the user enters a correct menu choice.  
*)  
    print "Unknown choice, try again."  
    ;;  
esac
```

- c. When the user selects a valid shell from the menu, print a verification of which shell was selected. Then allow the program to continue prompting the user for the comment-field information (which was written in a previous lab).

```
select ushell in csh ksh sh
do
    case $ushell in
        csh)
            print "$ushell was selected."
            break
            ;;
        ksh)
            print "$ushell was selected."
            break
            ;;
        sh)
            print "$ushell was selected."
            break
            ;;
        *)
            print "Unknown choice, try again."
            ;;
    esac
done
```

Task Solutions

The following is the complete solution:

```
#!/bin/ksh

# Purpose: To write a script to add user to the system. Generate a
#           menu for the login shell choices.
#
# Name: adduser.ksh

# Check for only a single command line argument
if [ $# -ne 1 ]
then
    echo "Usage: adduser.ksh newusername"
    exit 1
fi

# Assign the value of $1 to the variable name
name=$1

if grep "^\$name:" ./mypasswd > /dev/null 2>&1
then
    echo "The username $1 is already in use, here "
    echo "is the entry from the ./mypasswd file."
    echo
    grep "^\$name:" ./mypasswd
    exit 2
fi

# Use awk to extract all the UIDs into the temp file currentuid
awk -F: '{print $3}' ./mypasswd > ./currentuid

# Use sed to remove all 1, 2, and 3 digit UIDs from currentuid, and
# place the output in the temp file currentuid2
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2

# Use sed to remove UIDs 60001, 60002, and 65534 from currentuid2,
# and place the output in the temp file currentuid3
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3

# Sort the UIDs in currentuid3 numerically
sort -n ./currentuid3 > ./currentuid4

# Calculate the next available UID
lastuid='sed -n '$p' ./currentuid4'

uid='expr $lastuid + 1'
```

```
echo "The uid for $name is: $uid"

# Prompt for a group name and verify whether it exists in /etc/group

echo "What primary group should $a belong to? "
echo "Please specify the group name: \c"
read group junk

grep "^\$group:" /etc/group > /dev/null 2>&1
if [ $? -ne 0 ]
then
    echo "The group $group is not valid."
    echo
    exit 2
fi

PS3="Enter the number: "
select ushell in csh ksh sh
do
    case $ushell in
        csh)
            print "$ushell was selected."
            break
            ;;
        ksh)
            print "$ushell was selected."
            break
            ;;
        sh)
            print "$ushell was selected."
            break
            ;;
        *)
            print "Unknown choice, try again."
            ;;
    esac
done

if [ $ushell = "csh" -o $ushell = "ksh" -o $ushell = "sh" ]
then
    echo "Please enter the comment information for the user:"
    read comment
```

Task Solutions

```
echo "The new user entry looks like this:"  
echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell"  
echo "  
echo "New user $1 has been added to the ./mypasswd file"  
echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell" >> ./mypasswd  
else  
    echo "Sorry, that is not a valid shell, exiting..."  
fi
```

5. Copy the `filetype2` script to `filetype3`, and modify the `filetype3` script as follows:

- a. Where it uses a `while` loop, change the code to use an `until` loop.
- b. Where it uses an `until` loop, change the code to use a `while` loop.

Change:

while [\$# -ne 0]

To:

until [\$# = 0]

The following is the complete solution:

```
#!/bin/sh

# Purpose: This script will determine whether the pathnames passed
#           on the command line are a file or directory. Then it will
#           print information about the pathname.
#
# Scriptname: filetype3.sh

if [ $# -eq 0 ]; then
    echo "Usage: filetype3.sh pathname [pathname2...]"
    exit 1
fi

until [ $# = 0 ]
do
    if [ -d $1 ]; then
        echo "$1 is a directory."
        echo "Following is a listing of the $1 directory."
        ls $1
        echo ""
        echo "The number of kilobytes used by directory $1 is:
\c"
        du -s -k $1 | awk '{print $1}'
    else
        echo "$1 is a file."
        echo "The size of $1 in bytes is: \c"
        ls -l $1 | awk '{print $5}'
        echo
        if [ -b $1 ]; then
            echo "$1 is a block special file."
        elif [ -c $1 ]; then
            echo "$1 is a character special file."
        elif [ -u $1 ]; then
            echo "$1 has the set-user-id permission."
        else
            echo "There is nothing special to say about file
\$1."
        fi
        shift
        echo "\n\n"
    done
    echo "Finished"
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 10

The getopt Statement

Objective

Upon completion of this module, you should be able to process script options using the getopt statement.

Processing Script Options With the getopt Statement

Options or switches are single-letter characters preceded by a + sign or a - sign. By convention, a - sign preceding a character means to turn some flag on, while a + sign means to turn some flag off.

The Bourne shell supports only script options with a - sign preceding the character.

The syntax of getopt is:

```
getopt options [opt_args] var
```

where:

- *options* represents the valid single-letter characters the script expects as options from the command line. If any single-letter character in *options* is followed by a colon (:), then the shell expects that option to have an argument.
- *var* represents any variable name whose value is an option from the command line if the option is one of the valid single-letter characters specified in the getopt statement.

The getopt statement does not always process all options on the command line. It quits processing options if an argument that is not an option or an argument to an option is placed between two options. For example, suppose *scriptname* is a script with valid options x and y, neither of which requires an argument. Then the command causes the script to process the x option but not the y option:

```
scriptname -x filex -y
```

Arguments to options and the use of colons in the options are described in the “Specifying Arguments to Options” on page 10-7.

Note – When using the print \$* or print \$@ statements, you can get an error message if the first argument to the script is an option beginning with a - sign. Thus, it is safest to use

```
print -- $* or print -- $@.
```

When using the echo \$* or echo \$@ statements, you do not get an error message.

Using the getopt Statement

The `getopt` statement is most often used as the condition in a `while` loop, and the `case` statement is usually the command used to specify the actions to be taken for the various options that can appear on the command line.

The variable name used in the `case` statement is the name of the variable specified in the `getopt` statement. In the following code fragment, the variable name is `opt_char`.

```
while getopt xy opt_char
do
  case $opt_char in
    x) print "Option is -x";;
    y) print "Option is -y";;
    +x) print "Option is +x";;
    +y) print "Option is +y";;
  esac
done
```

If `-x` was given as an option to a script, the value of `opt_char` is `x`. If `+x` was given as an option to a script, the value of `opt_char` is `+x`.

The correct ways to invoke the script (if you have execute permission on the script) are:

scriptname `[+-]x [+-]y`

or

scriptname `[+-]xy`

The `while` loop is exited when there are no more options to process.

Note – Don't use the `shift` statement on command-line arguments until all options to a script are processed. If you use the `shift` statement within the `while` loop that contains the `getopt` statement, the `getopt` statement does not process correctly.

Handling Invalid Options

The options on the command line can be given in different ways. The following are some valid possibilities for a script that accepts x and y as options:

```
scriptname -x -y  
scriptname -xy  
scriptname +x -y  
scriptname +x +y  
scriptname +xy
```

Handling Invalid Options

Unless you program the `getopts` statement to look for invalid options and to handle an invalid option when it encounters one, the shell gives its own error message.

To process invalid options given on the command line, precede the list of single-character options with a colon; for example:

```
while getopts :xy opt_char
```

The beginning colon causes `getopts` to:

- Set the value of the `opt_char` variable to ?
- Set the value of the `OPTARG` reserved variable to the name of the invalid option

If the script uses the case statement to match the options from the command line, then the \? pattern is used to match the value of *opt_char* when an invalid option is encountered. The backslash is necessary to escape the meaning of the ? metacharacter.

```
$ cat getoptsex.ksh
#!/bin/ksh

# Script name: getoptsex.ksh

USAGE="usage: $0 -x -y"

while getopts :xy opt_char
do
    case $opt_char in
        x)
            echo "You entered the x option"
            ;;
        y)
            echo "You entered the y option"
            ;;
        \?)
            echo "$OPTARG is not a valid option."
            echo "$USAGE"
            ;;
    esac
done

$ ./getoptsex.ksh -y -x -z
You entered the y option
You entered the x option
z is not a valid option.
usage: ./getoptsex.ksh -x -y
```

Note – The Bourne shell does not assign the invalid option to the OPTARG variable .

Handling Invalid Options

```
$ cat getoptsex.sh
#!/bin/sh

# Script name: getoptsex.sh

USAGE="usage: $0 -x -y"

while getopts :xy opt_char
do
    case $opt_char in
        x)
            echo "You entered the x option"
            ;;
        y)
            echo "You entered the y option"
            ;;
        \?)
            echo "$OPTARG is not a valid option."
            echo "$USAGE"
            ;;
    esac
done

$ ./getoptsex.sh -y -x -z
You entered the y option
You entered the x option
    is not a valid option.
usage: ./getoptsex.sh -x -y
```

Specifying Arguments to Options

If an option requires an argument, then place a colon (:) immediately after the option character in the `getopts` statement.

```
while getopts :x:y opt_char
```

The colon after the `x` tells the `getopts` statement that an argument must immediately follow, with or without a space between the `x` and the argument.

After it is executed, the required argument to the `x` option is assigned to the `OPTARG` variable.

When the script is executed, the options that require an argument must be followed by their arguments before another option can be given. For example, if `x` and `y` are valid options in a script, with `x` requiring an argument, then the following commands are all valid ways of invoking the script:

```
scriptname -x x_arg -y  
scriptname -xx_arg -y  
scriptname -yx x_arg  
scriptname -yxx_arg
```

The argument to an option is placed in the `OPTARG` variable so that it can be accessed by the statements in the `case` statement for the corresponding option.

Example of Using the getopt Statement

Example of Using the getopt Statement

Consider the following example:

```
$ cat getopt1.ksh
#!/bin/ksh

# Script name: getopt1.ksh

USAGE="usage: $0 [-d] [-m month]"

year=$(date +%Y)

while getopt :dm: opt_char
do
    case $opt_char in
        d)
            print -n "Date: "          # -d option given
            date
            ;;
        m)
            cal $OPTARG $year        # -m option given with an arg
            ;;
        \?)
            print "$OPTARG is not a valid option."
            print "$USAGE"
            ;;
    esac
done
```

The `USAGE` variable holds the value of how the script is to be invoked.

The `year` variable holds the value of the current year (2000).

The `getopt` statement within the `while` loop specifies that this script has two valid options: `d` and `m`. The `m` option is followed by a colon (`:`), so it is required to have an argument supplied to it on the command line. The beginning colon in `:dm:` tells the `getopt` statement to watch for invalid switches. (It also tells `getopt` to watch for options on the command line that require an argument but have no argument supplied to them.)

```
$ ./getopts1.ksh -dk
Date: Wed Nov 25 18:42:35 IST 2009
k is not a valid option.
usage: ./getopts1.ksh [-d] [-m month]
```

In the preceding example, the `-d` option is correctly processed. Then the invalid `-k` option is encountered and, therefore, the `getopts` statement sets the value of `opt_char` to a `?`. Then the `\?)` case matches the value in `opt_char` and the corresponding statements are executed.

```
$ ./getopts1.ksh -d
Date: Wed Nov 25 18:43:11 IST 2009
$ ./getopts1.ksh -m
```

In the preceding example, you might have expected output because the `-m` option is valid, but the option requires an argument to perform its action. Remember that the colon after the `m` in `getopts :dm:` states that it requires an argument following it on the command line. If there is no argument, the `m)` case is not matched and processed.

```
$ ./getopts1.ksh -m 6
June 2009
S M Tu W Th F S
 1 2 3 4 5 6
 7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

$ ./getopts1.ksh -d filex
Date: Wed Nov 25 18:44:49 IST 2009
```

The last time the `getopts1.ksh` script is executed, two arguments are given to it: `-d` and `filex`. Because `filex` is not preceded with a `-` or a `+` sign, it is not processed as an option to the script.

Note – If `filex` had been the first argument given to the script, *no* options would have been processed.

Forgetting an Argument to an Option

Forgetting an Argument to an Option

The following `getopts2.ksh` example script is the `getopts1.ksh` script with an added pattern `(:)` in the `case` statement. This added pattern is matched when an option that requires an argument is given on the command line but is not followed with its argument.

In the example output, the `getopts2.ksh` script is executed and given the `-m` option without an argument so that you can see what is printed; otherwise, the script runs the same as the `getopts1.ksh` script.

```
$ cat getopt2.ksh
#!/bin/ksh

# Script name: getopt2.ksh

USAGE="usage: $0 [-d] [-m month]"

year=$(date +%Y)

while getopts :dm: opt_char
do
    case $opt_char in
        d)
            print -n "Date: "           # -d option given
            date
            ;;
        m)
            cal $OPTARG $year         # -m option given with an arg
            ;;
        \?)
            print "$OPTARG is not a valid option."
            print "$USAGE"
            ;;
        :)
            print "The $OPTARG option requires an argument."
            print "$USAGE"
            ;;
    esac
done

$ ./getopt2.ksh -m
The m option requires an argument.
usage: ./getopt2.ksh [-d] [-m month]
```

If you rewrite the script for the Bourne shell, the script looks much the same except the syntax for the following changes:

- The command substitution for the year variable
- The print statements are changed to echo
- If the OPTARG variable is used, it would be null

Forgetting an Argument to an Option

Therefore, the code resembles:

```
$ cat getopt2.sh
#!/bin/sh

# Script name: getopt2.ksh

USAGE="usage: $0 [-d] [-m month]"

year='date +%Y'

while getopt :dm: opt_char
do
    case $opt_char in
        d)
            echo "Date: \$c"          # -d option given
            date
            ;;
        m)
            cal $OPTARG $year      # -m option given with an arg
            ;;
        \?)
            echo "\$OPTARG is not a valid option."
            echo $USAGE
            ;;
        :)
            echo "The \$OPTARG option requires an argument."
            echo "$USAGE"
            ;;
    esac
done

$ ./getopt2.sh -m
The option requires an argument.
usage: ./getopt2sh [-d] [-m month]
```

In the preceding Bourne shell example, the reference to \$OPTARG evaluated to null in the echo statement; otherwise, this script executes the same as the Korn shell version.

Exercise: Using the getopt Statement

Exercise objective – Modify a shell script to process a command-line argument with the getopt statement.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Tasks

1. Copy the getopt2.ksh (or getopt2.sh) program from the mod10/examples directory, and rename the file to getopt3.ksh (or getopt3.sh).
2. Add another option, c, that prints a custom date; use the statement:
`date "+%a %b %e %Y"`
that produces output similar to:
`Thu Nov 25 2009`
The new option does not require an argument.

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Copy the getopt2.ksh (or getopt2.sh) program from the mod10/examples directory, and rename the file getopt3.ksh (or getopt3.sh).

```
cp ..mod10/examples/getopts2.ksh getopt3.ksh
```

```
cp ..mod10/examples/getopts2.sh getopt3.sh
```

2. Add another option, c, that prints a custom date; use the statement:

```
date "+%a %b %e %Y"
```

that produces output similar to:

```
Thu Nov 25 2009
```

The new option does not require an argument.

```
while getopt :dcm: opt_char
do
...
c) print -n "Custom date: "
    date "+%a %b %e %Y" ;;
...
done
```

Task Solutions

The following is the complete solution:

```
$ cat getopt3.ksh
#!/bin/ksh

# Script name: getopt3.ksh

USAGE="usage: $0 [-d] [-m month]"

year=$(date +%Y)

while getopt :d:m: opt_char
do
    case $opt_char in
        d) print -n "Date: "           # -d option given
            date ;;
        m) cal $OPTARG $year ;;       # -m option given with an arg
        \?) print "$OPTARG is not a valid option."
            print "$USAGE" ;;
        :) print "The $OPTARG option requires an argument."
            print "$USAGE" ;;
        c) print -n "Custom date: "
            date "+%a %b %e %Y" ;;
    esac
done
```

```
$ cat getopt3.sh
#!/bin/sh

# Script name: getopt3.sh

USAGE="usage: $0 [-d] [-m month]"

year='date +%Y'

while getopt :d:m: opt_char
do
    case $opt_char in
        d) echo "Date: \$c"           # -d option given
            date ;;
        m) cal $OPTARG $year ;;      # -m option given with an arg
        \?) echo "\$OPTARG is not a valid option."
            echo $USAGE ;;
        :) echo "The \$OPTARG option requires an argument."
            echo "\$USAGE" ;;
        c) echo "Custom date: \$c"
            date "+%a %b %e %Y" ;;
    esac
done
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 11

Advanced Variables, Parameters, and Argument Lists

Objectives

Upon completion of this module, you should be able to:

- Declare strings, integers, and array variables
- Use the Korn shell built-in `let`, `print`, `set`, and `typeset` statements
- Manipulate string variables
- Change the values of the positional parameters using the `set` statement within a script
- Use Korn shell arrays
- Set default values for parameters

Variable Types

Variable Types

In the Bourne shell there are two variable types: *strings* and *constants*.

In the Korn shell there are four variable types: *strings*, *integers*, *arrays*, and *constants*.

- All variables are of type string unless explicitly declared otherwise.
- You can specify constants in a shell. A constant is a read-only variable that was assigned a value when it is declared and the value cannot be changed.
- A variable data type determines the values that can be assigned to the variable.

You can assign any value to a string variable. Integer variables can hold only number values, and arrays might be restrictive, depending on whether an array is an array of integers or strings.

Assessing Variable Values

When you access the value of a variable by preceding its name with a \$, you might need to isolate the variable name from the characters that immediately follow it; for example:

```
$ flower=rose  
  
$ print "$flower $flowers $flowerbush"  
rose  
  
$ print "${flower} ${flower}s ${flower}bush"  
rose roses rosebush
```

The Korn Shell typeset Statement

The typeset statement sets variable attributes. In some cases, it changes a variable value (for example, a right- or left-justification). The typeset statement options are shown in Table 11-1.

Table 11-1 typeset Statement Options

Syntax	Description
<code>typeset -u var</code>	Converts <code>var</code> to all uppercase characters
<code>typeset -l var</code>	Converts <code>var</code> to all lowercase characters
<code>typeset -LZ var</code>	Strips leading zeros from the string <code>var</code>
<code>typeset -Lnum var</code>	Left-justifies <code>var</code> within the field width specified by <code>num</code>
<code>typeset -Rnum var</code>	Right-justifies <code>var</code> within the field width specified by <code>num</code>
<code>typeset -i var</code>	Specifies that <code>var</code> can only contain integer values
<code>typeset -r var</code>	Specifies that <code>var</code> is read-only. Its value can't be changed by subsequent assignment

Example of Using String Manipulations

Quote the variable in the output statement to see a right-justified string with leading spaces (see example below):

```
$ cat strman1.ksh
#!/bin/ksh

# Script name: strman1.ksh

typeset -R8 word="happy"

typeset -L5 word1="depressed"

print "123456789"
print "$word"
print

print "123456789"
print "$word1"
```

The Korn Shell typeset Statement

```
$ ./strman1.ksh  
123456789  
happy
```

```
123456789  
depre
```

Without quotes, leading spaces aren't printed (see the following example):

```
$ cat strman2.ksh  
#!/bin/ksh  
  
# Script name: strman2.ksh  
  
typeset -R8 word="happy"  
  
typeset -L5 word1="depressed"  
  
  
print "123456789"  
print $word  
print  
  
print "123456789"  
print $word1  
  
  
$ ./strman2.ksh  
123456789  
happy  
  
123456789  
depre
```

The following script further demonstrates string manipulations. The script converts all characters in a string to uppercase and then lowercase characters. The script then left- and right-justifies the characters within the string.

The line that prints 123456789 is included so you can check the output with regard to the field widths assigned to the variables ljust and rjust.

```
$ cat strman3.ksh
#!/bin/ksh

# Script name: strman3.ksh

string1="manipulation"
print "Length of string1 is ${#string1} characters \n"

string2="CaSes"
print "string2 is $string2"

typeset -u string2
print "string2 in upper case: $string2"

typeset -l string2
print "string2 in lower case: $string2 \n"

typeset -L7 ljust
ljust="hi there"

typeset -R5 rjust
rjust="farewell"

print "           123456789"
print "Value of ljust: $ljust"
print "Value of rjust: $rjust \n"

lzero="00034;lsl"
print "Value of lzero: $lzero"

typeset -LZ lzero
print "New value of lzero: $lzero"

$ ./strman3.ksh
Length of string1 is 12 characters
string2 is CaSes
string2 in upper case: CASES
string2 in lower case: cases
           123456789
Value of ljust: hi ther
Value of rjust: ewell
Value of lzero: 00034;lsl
New value of lzero: 34;lsl
```

Declaring an Integer Variable

Declare an integer variable in two ways by using either typeset -i or integer in front of the variable name. The first way is more flexible because it also allows you to specify the base in which the variable is represented.

```
typeset -i int_var1[=value] int_var2[=value] ... int_varn[=value]
```

```
integer int_var1[=value] int_var2[=value] ... int_varn[=value]
```

An integer variable can have only integer numbers assigned to it. An assignment of a number with a decimal part results in the decimal being truncated.

An assignment containing nonnumeric characters to an integer variable results in an error message.

```
$ typeset -i num # base 10 integer
$ num=5
$ print $num
5

$ typeset -i num # base 10 integer
$ num=25.34      # truncation will occur on the decimal portion
$ print $num
25

$ typeset -i num # base 10 integer
$ num=27
$ print $num
27

$ typeset -i8 num # change to base 8
$ print $num
8#33

$ num=two
/usr/bin/ksh: two: bad number
$ print $num
8#33
```

Creating Bourne Shell Constants

A variable can be made *read-only* with the following syntax:

```
readonly var [=value]
```

The square brackets around `=value` mean that the assignment of a value is not always necessary. For instance, if the variable had previously been created and assigned a value, and you now want to make it read-only (and not change its current value), do not use `=value`.

If the variable did not previously exist, and you make it read-only, you may never assign a value to the variable.

The value of a read-only variable cannot be changed. This is why read-only variables are referred to as *constants*:

```
$ sh  
$ var=constant  
$ readonly var  
$ unset var  
var: is read only  
$ var=new_value  
var: is read only
```

Creating Korn Shell Constants

Creating Korn Shell Constants

A variable can be made *read-only* by using either of the following syntaxes:

```
typeset -r var[=value]
readonly var[=value]
```

The following is a Korn shell example:

```
$ ksh

$ typeset -r cvar=constant

$ unset cvar
ksh: cvar: is read only

$ cvar=new_value
ksh: cvar: is read only
```

Removing Portions of a String

Table 11-2 summarizes the syntax used to remove portions of a string.

Table 11-2 Removing Portions of a String

Syntax	Description
<code> \${str_var%pattern}</code>	Removes the smallest right-most substring of string <i>str_var</i> that matches <i>pattern</i>
<code> \${str_var%%pattern}</code>	Removes the largest right-most substring of string <i>str_var</i> that matches <i>pattern</i>
<code> \${str_var#+pattern}</code>	Removes the smallest left-most substring of string <i>str_var</i> that matches <i>pattern</i>
<code> \${str_var##pattern}</code>	Removes the largest left-most substring of string <i>str_var</i> that matches <i>pattern</i>

You can use any of the shell metacharacters in a pattern following the %, %% , #, and ## symbols.

Note – When the pattern does not contain any metacharacters, then `${str_var%pattern}` and `${str_var%%pattern}` have the same value, as do `${str_var#+pattern}` and `${str_var##pattern}`.

Examples of Removing Portions of a String

Removing portions of a string is useful when working with path names. Sometimes only the name of a file is needed. You must strip everything in the path name except for the *last component* (the group of characters following the last / in the path name).

Sometimes the parent directory is required for a file for which you have the absolute path name. You must remove the last / and the characters that follow it.

The following example sets the value of variable *stringx* to /usr/bin/local/bin:

```
$ stringx=/usr/bin/local/bin
```

Removing Portions of a String

The *smallest* right-most substring that matches /bin and /bin* in both of these cases is /bin. Thus, the result of the following example is /usr/bin/local:

```
$ print ${stringx%/bin}  
/usr/bin/local
```

```
$ print ${stringx%/*bin}  
/usr/bin/local
```

The *largest* right-most substring that matches /bin is /bin. Thus, the result of the following example is the same as for the preceding two cases.

```
$ print ${stringx%%bin}  
/usr/bin/local
```

The *largest* right-most substring that matches /bin* is /bin/local/bin. Thus, the result of the following example is the string /usr:

```
$ print ${stringx%*/bin*}  
/usr
```

The *smallest* left-most substring that matches /usr/bin and */bin is /usr/bin in both of these cases. Thus, the result of the following example is /local/bin:

```
$ print ${stringx#/usr/bin}  
/local/bin
```

```
$ print ${stringx##*/bin}  
/local/bin
```

The *largest* left-most substring that matches /usr/bin is /usr/bin. Thus, the result of the following example is the same as for the proceeding two cases:

```
$ print ${stringx##/usr/bin}  
/local/bin
```

The *largest* left-most substring that matches */bin is /usr/bin/local/bin. Thus, the result of the following example is the blank (null) string:

```
$ print ${stringx##*/bin}
```

The *largest* left-most substring that matches */ is /usr/bin/local/. Thus, the result of the following example is the string bin:

```
$ print ${stringx##*/}  
bin
```

Korn Shell Arrays

Arrays are variables that contain more than one value. Don't declare arrays explicitly or explicitly set an array size. An array is created when you first use it. It is treated as an array when you first assign a value to one element of the array variable. Each array can hold up to 1024 values.

You can create arrays of strings or integers. By default, an array contains strings. To create an array of integers, declare a variable as an integer, and then use it as an array by assigning integer values to the elements; for example:

```
integer my_array  
my_array[1]=5  
my_array[12]=16
```

The first element of an array is indexed by 0, the second element is indexed by 1, and so on. Therefore, the largest index value that is valid is 4095.

All arrays in the Korn shell are one-dimensional arrays. Enclose an array reference in braces for the Korn shell to recognize it as an array. For example use \${arr[1]} instead of \$arr[1].

The syntax for accessing the value of the *i*th array element is the following, where *i* is an integer:

```
{array_name [i]}
```

To print the values of all array elements, use the syntax:

```
{array_name [*]}
```

To print the number of elements (assigned values) in the array, use the syntax:

```
{#array_name [*]}
```

You don't have to assign array elements in order or assign values to any elements. Skipped array elements are not assigned a value and are treated as though they do not exist.

```
arr[2]=two  
arr[4]=4  
arr[8]=eight  
arr[16]=16
```

Note – You cannot use the export statement with arrays in the Korn shell.

Examples of Using Arrays

Examples of Using Arrays

The first two of the following examples illustrate how to create arrays of strings. You do not need to enclose the values assigned to the various array elements within double quotes unless the string value contains special characters, such as spaces or tabs.

To create an array of three strings:

```
$ arr[0]=big  
$ arr[1]=small  
$ arr[2]="medium sized"
```

To create an array of three strings using the set statement:

```
$ set -A arr big small "medium sized"
```

Note – Not all Korn shell implementations support the `set -A` statement.

In the following examples, note the creation of the array of integers. The variable num is first declared as an integer variable and then it is used as the name of an array to assign integers to the first five elements of the array. If you try to assign a value other than an integer to any of the elements of the array, you get a bad number error message. To create an array of five integers:

```
$ integer num  
$ num[0]=0  
$ num[1]=100  
$ num[2]=200  
$ num[3]=300  
$ num[4]=400
```

To print the number of array elements in the num array:

```
$ print ${#num[*]}\n5
```

To print the values of all array elements in the arr array:

```
$ print ${arr[*]}\nbig small medium sized
```

To unset the arr array:

```
$ unset arr\nClosure (*)4-13\nClosure (*)4-13
```

Examples of Using Arrays

Using the shift Statement With Positional Parameters

By default, the `shift` statement shifts the values held in the positional parameter. The `shift` statement drops the first value (`$1`) and shifts all other values to the left. So the value in `$2` is shifted or copied into `$1`, then the value in `$3` is shifted or copied into `$2`, and so on. This is shown in the following example.

```
$ cat argtest.sh
#!/bin/sh

# Script name: argtest.sh

echo '$#: $#'
echo '$@: $@"
echo '$*: $*
echo
echo '$1 $2 $9 $10 are: ' $1 $2 $9 $10
echo

shift
echo '$#: $#'
echo '$@: $@"
echo '$*: $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

shift 2
echo '$#: $#'
echo '$@: $@"
echo '$*: $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

echo '${10}: ${10}
```

```
$ ./argtest.sh a b c d e f g h i j k l m n
$#: 14
$@: a b c d e f g h i j k l m n
$*: a b c d e f g h i j k l m n

$1 $2 $9 $10 are: a b i a0

$#: 13
$@: b c d e f g h i j k l m n
$*: b c d e f g h i j k l m n

$1 $2 $9 are: b c j
$#: 11
$@: d e f g h i j k l m n
$*: d e f g h i j k l m n

$1 $2 $9 are: d e l
./argtest.sh: bad substitution
```

The echo of `${10}` causes the error message at the end of the output because the Bourne shell cannot access command-line parameters beyond `$9`.

In the Korn shell, braces are needed around the argument number when referring to all command-line arguments beyond the ninth one. For example, to display the contents of the tenth argument, use the syntax `${10}` .

To view the difference between the Bourne shell and the Korn shell, copy the previous script and change the interpreter to the Korn shell, as in the following example:

Examples of Using Arrays

```
$ cat argtest.ksh
#!/bin/ksh

# Script name: argtest

echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 $10 are: ' $1 $2 $9 $10
echo

shift
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

shift 2
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

echo '${10}: ' ${10}

$ ./argtest.ksh a b c d e f g h i j k l m n
$#: 14
$@: a b c d e f g h i j k l m n
$*: a b c d e f g h i j k l m n

$1 $2 $9 $10 are: a b i a0

$#: 13
$@: b c d e f g h i j k l m n
$*: b c d e f g h i j k l m n

$1 $2 $9 are: b c j
$#: 11
$@: d e f g h i j k l m n
$*: d e f g h i j k l m n

$1 $2 $9 are: d e l
${10}: m
```

The Values of the "\$@" and "\$*" Positional Parameters

The values of \$@ and \$* are identical, but the values of "\$@" and "\$*" are different.

The expansion of "\$@" is a list of strings consisting of the values of the positional parameters. The expansion of "\$*" is one long string containing the positional parameter values separated by the first character in the set of delimiters of the IFS variable; for example:

```
$ cat posparatest1.ksh
#!/bin/ksh

# Script name: posparatest1.ksh

set This is only a test

print 'Here is the $* loop.'
for var in "$*"
do
    print "$var"
done

print \n"
print 'Here is the $@ loop.'
for var in "$@"
do
    print "$var"
done

$ ./posparatest1.ksh
Here is the $* loop.
This is only a test

Here is the $@ loop.
This
is
only
a
test
```

Examples of Using Arrays

"\$@" expands to "\$1" "\$2" "\$3" ... "\$n"; that is, *n* separate strings.

"\$*" expands to "\$1x\$2x\$3x...\$n", where *x* is the first character in the set of delimiters for the IFS variable. This means that "\$*" is one long string.

When you use quotes to group tokens into a string, the string sets the value of a single positional parameter.

```
$ cat posparatest2.sh

#!/bin/sh

# Script name: posparatest2.sh

set "This is a test" and only a test

echo 'Here is the $* loop output: '
for var in "$*"
do
    echo "$var"
done

echo "\n"
echo 'Here is the $@ loop output: '
for var in "$@"
do
    echo "$var"
done

$ ./posparatest2.sh
Here is the $* loop output:
This is a test and only a test

Here is the $@ loop output:
This is a test
and
only
a
test
```

Assigning Positional Parameter Values Using the `set` Statement

Although you cannot use any positional parameter name on the left side of an assignment statement, you can still assign values to the positional parameters if you use the `set` statement:

```
set value1 value2 ... valueN
```

With the preceding syntax, `$1` has value `value1`, `$2` has value `value2`, and so on. The value of the positional parameter `$0` is the name of the script.

Use the `set` statement to create a parameter list using the statement or variable substitution.

```
set $(cal)  
set $var1
```

To sort the positional parameters lexicographically, use the `set -s` statement. If you sort the list of values in lexical order, the first value on the list is assigned to `$1`, the second to `$2`, and so forth.

```
set -s
```

The statement `set --` unsets all the positional parameters. Thus, `$1`, `$2`, and so on, have no values. The value of `$0`, is still the script name.

```
set --
```

The `set -s` and `set --` statements work on the positional parameters regardless of how the positional parameter names were assigned their values. These statements work whether values were assigned from command-line arguments or by using the `set` statement.

Statement or variable substitutions, combined with the `set` statement, is useful. For example, to find out how many days are in the current month, use the `cal` statement. The `cal` statement outputs a value for each day in the month plus an additional nine values: the month, the year, and the day-of-week values.

Examples of Using Arrays

Example of Using the set Statement

The following `pospar2.scr` example script sets the values of the positional parameters using the `set` statement.

The values `a`, `b`, and `c` are passed into the script and assigned to positional parameters `$1`, `$2`, and `$3` respectively. The values of the positional parameters are displayed using the `print` statement and then given new values with the `set` statement.

The `textline` variable is assigned a value, which is a string consisting of several words. The statement `set $textline` assigns new values to the positional parameters. Because `$textline` is replaced by its value (a string of several words), the words become the positional parameter values.

The script then prints the positional parameter values using the `print $*` statement.

Then the script prints the values of the `$1` and `$4` positional parameters.

Next, the positional parameters are sorted by the statement `set -s`, and then `print $*` is used to print the sorted list. Lastly, the `set --` statement unsets the positional parameters so that when the `print $0 $*` statement is executed, only the name of the script (which is held in `$0`) is printed.

```
$ cat pospara2.ksh
#!/bin/ksh

# Script name: pospara2.ksh

print "Executing script $0 \n"
print "$1 $2 $3"

set uno duo tres
print "One two three in Latin is:"
print "$1"
print "$2"
print "$3 \n"

textline="name phone address birthdate salary"
set $textline
print "$*"
print 'At this time $1 =' $1 'and $4 =' $4 "\n"

set -s
print "$* \n"

set --
print "$0 $*"

$ ./pospara2.ksh a b c
Executing script ./pospara2.ksh

a b c
One two three in Latin is:
uno
duo
tres

name phone address birthdate salary
At this time $1 = name and $4 = birthdate

address birthdate name phone salary
./pospara2.ksh
```

Exercise: Using Advanced Variables, Parameters, and Argument Lists

Exercise: Using Advanced Variables, Parameters, and Argument Lists

Exercise objective – Write scripts that manipulate the positional parameters, use a loop construct, and perform integer arithmetic.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change the directory to mod11/lab before beginning the exercise.

When writing the script on paper, you might want to execute commands on the system to ensure you have the correct commands and options to give you the prescribed output.

Tasks

1. Write a Korn script called `arginfo.ksh` that displays the following information:
 - a. Display the name of the script being executed (`$0`).
 - b. Display the first, second, and tenth parameters that have been passed to the script (`$1`, `$2` and `${10}`).
 - c. Display the number of parameters that were passed to the script (`$#`).
 - d. If there were five or more positional parameters, use the `shift` statement to move all the values of the positional parameters over by five places.
 - e. Print all the values remaining in the positional parameters.
 - f. Print the number of positional parameters.
2. After you have written the script, make it executable, and then run it. Make sure that you test it thoroughly by running it with no parameters, just one parameter, and with several (more than 10) parameters.
3. Write a Korn script called `parsepath.ksh` that parses information from a long path name. Assume the path name is the only item passed in on the command line.

Exercise: Using Advanced Variables, Parameters, and Argument Lists

A good path name is: /usr/share/lib/terminfo/s/sun.

An example execution is:

```
$ ./parsepath.ksh /usr/share/lib/terminfo/s/sun
```

```
The pathname is: /usr/share/lib/terminfo/s/sun  
/usr/share/lib/terminfo/s/sun  
/usr/share/lib/terminfo/s  
/usr/share/lib/terminfo  
/usr/share/lib  
/usr/share  
/usr
```

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Solutions

1. Write a Korn script called `arginfo.ksh` that displays the following information:

- a. Display the name of the script being executed (`$0`).

```
print "The name of the script is: $0"
```

- b. Display the first, second and tenth parameters that were passed to the script (`$1`, `$2` and `${10}`).

```
print "Positional Parameter      Value"  
print "      \$1                  \$1"  
print "      \$2                  \$2"  
print "      \${10}                ${10}"
```

- c. Display the number of parameters that were passed to the script (`$#`).

```
print "The number of positional parameters is: $"#"
```

- d. If there are five or more positional parameters, use the `shift` statement to move all the values of the positional parameters over by five values.

```
if (( $# >= 5 )); then  
...  
    shift 5  
...  
fi
```

- e. Print all the values remaining in the positional parameters.

```
print "Positional parameters remaining:  
print $*
```

- f. Print all the remaining positional parameter values.

```
print "The number of positional parameters is: $"#"
```

Exercise Solutions

2. After you write the script, make it executable, and then run it. Make sure that you test it thoroughly by running it with no parameters, just one parameter, and with several (more than 10) parameters.

The following is the complete solution:

```
$ cat arginfo.ksh
#!/bin/ksh

# Purpose: Print out information about the positional parameters.
#
# Name: arginfo.ksh

print "The name of the script is: $0"

print "Positional Parameter      Value"
print "      \$1                  \$1"
print "      \$2                  \$2"
print "      \${10}                \${10}"

print "The number of positional parameters is: $#"
print

if (( $# >= 5 )); then
    print "Now shift all values over by 5 places."
    shift 5

    print "Positional parameters remaining:"
    print $*
    print

    print "The number of positional parameters is: $#"
fi
```

3. Write a Korn script called `parsepath.ksh` that parses information from a long path name. Assume the path name is the only item passed in on the command line.

A good path name is: `/usr/share/lib/terminfo/s/sun`.

An example execution is:

```
$ ./parsepath.ksh /usr/share/lib/terminfo/s/sun
```

```
The pathname is: /usr/share/lib/terminfo/s/sun
/usr/share/lib/terminfo/s/sun
/usr/share/lib/terminfo/s
/usr/share/lib/terminfo
/usr/share/lib
/usr/share
/usr
```

```
$ cat parsepath.ksh
#!/bin/ksh

# Purpose: To parse a long pathname.
#
# Name: parsepath.ksh

if [ $# -ne 1 ]; then
    echo "Usage: parsepath.ksh /pathname"
    exit 1
fi

name=$1

print
print "The pathname is: $name"

while [[ -n $name ]]
do
    print $name

    name=${name%/*}
done

print
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Module 12

Functions

Objectives

Upon completion of this module, you should be able to:

- Create user-defined functions in a shell script
- Create, invoke, and display functions from the command line
- Pass arguments into a function
- Call functions from special (function) files that are saved in one or more function directories
- Describe where functions are available for use

Functions in the Shell

A function is a set of one or more statements that act as a complete routine. Each function must have a unique name *within that shell or shell script*.

Syntax

The following is the function syntax in the Bourne shell:

```
function_name ()  
{  
    block_of_statement_lines  
}
```

The following is the function syntax in the Korn shell:

```
function function_name  
{  
    block_of_statement_lines  
}
```

The following is the function syntax in the Bash shell:

```
function function_name [ block_of_statement_lines  
]
```

Function Execution

A function is executed within the shell in which it has been declared; the function is executed as a subroutine of the shell. It is not executed as a subprocess of the current shell. After a function has been loaded into the current shell, it is retained.

Positional Parameters and Functions

Functions act like mini-scripts. They can accept parameters that are passed to them, they can use local variables, and they can return values back to the calling shell command line.

Positional parameters passed to the function are not the same positional parameters that are passed to the script; for example:

```
$ cat funparas.ksh
#!/bin/ksh

# Script name: funparas.ksh

function hello
{
    print '$1 in the function is: ' $1
}

print 'Input passed and stored in $1 is: ' $1
hello John          # execute the function hello
print
print 'After the function $1 is still ' $1

$ ./funparas.ksh Susan
Input passed and stored in $1 is: Susan
$1 in the function is: John

After the function $1 is still Susan
```

Positional Parameters and Functions

The following customized .kshrc file shows three functions: killit, rcgrep, and rgrep. The following examples show the execution of each of the three functions.

```
$ cat /.kshrc

function killit          # Korn shell syntax
{
    pkill -u $1
    print -n "Had to kill process for user: $1 "
    print "on $(date +%D) at $(date +%T)"

    # The previous print statement may be appended to a log file.
}

function rcgrep          # Korn shell syntax
{
    grep $1 /etc/init.d/* | more
}

rgrep ()                 # Bourne shell syntax
{
    find $2 -type file -exec grep $1 {} \; | more
}
```

The following example calls the killit function, which calls pkill to kill all processes owned by the user specified as the first argument to the function. It then prints a message that is logged to a file.

```
$ killit annette
Had to kill process for user: annette on 05/23/00 at 21:38:32
```

The following example executes a grep search through all the files in the /etc/init.d directory for the string passed in as the first argument to the rcgrep function.

```
$ rcgrep sed
/etc/init.d/autoinstall:# are available, then the default profiles in are used.
/etc/init.d/dtlogin:# This version of the dtlogin.rc script can be used on the Solaris(TM)
/etc/init.d/inetinit:# tcp used when it loaded.

<Output truncated>
```

The following example performs a recursive grep search. Two arguments are passed to the `rcgrep` function: the first is the search string, and the second is the directory from which to begin the search. The `find` statement recursively finds all files under that directory, and passes each file name to the `grep` statement, which then searches for the search string within each file. Therefore, you can apply `grep` to the contents of the files, rather than to the file names.

```
$ rgrep root /etc/default
/etc/default/login:# If CONSOLE is set, root can only login on that
device.
/etc/default/login:# Comment this line out to allow remote login by root.
/etc/default/login:# SUPATH sets the initial shell PATH variable for root
/etc/default/login:# to log all root logins at level LOG_NOTICE and
multiple failed login
/etc/default/su:# CONSOLE determines whether attempts to su to root
should be logged
/etc/default/su:# SUPATH sets the initial shell PATH variable for root
/etc/default/su:# root, LOG_INFO messages are generated for su's to other
users, and LOG_CRIT
```

Return Values

A value can pass from a function back to the shell or script that called that function.

The `return` statement terminates the function and passes a value back to the calling shell or script. The `return` statement returns any designated value between 0 (zero) and 255. By default, the value passed by the `return` statement is the current value of the `? exit status` variable.

The typeset and unset Statements

The typeset and unset Statements

The following shows the syntax for using typeset and unset with functions:

- typeset -f lists the known functions and their definitions
- typeset +f lists the known function names
- functions is an alias for typeset -f
- unset -f *name* unsets the value of the function

The following are examples of using typeset and unset with functions:

```
$ typeset -f
function killit
{
    pkill -u $1
    print -n "Had to kill process for user: $1 "
    print "on $(date +%D) at $(date +%T)"

    # The previous print statement may be appended to a log file.
}
function rcpgrep
{
    grep $1 /etc/init.d/* |more
}
function rgrep
{
    find $2 -type file -exec grep $1 {} \; | more
}
```

The following function will echo its purpose and the line it will execute:

```
# System call count by process
function syscall_count_by_process
{
    echo "Syscall count by program"
    echo "dtrace -n 'syscall:::entry { @num[execname] = count(); }'"
    dtrace -n 'syscall:::entry { @num[execname] = count(); }'

$ typeset +f
killit
rcpgrep
rgrep
```

```
$ alias | grep fun
functions='typeset -f'
$ unset -f rcgrep
$ typeset +f
killit
rgrep
```

Function Files

You can create functions within a shell script, or they can be external to the shell script. Functions created within a shell script exist only within the shell interpreting that script. You can put functions in a file. When you create a function, it must contain the definition of only one function. The name of a function and function file must be the same. Make a function file an executable. A function file can be autoloaded into a shell script and used by that script. Only one function can be in each function file; for example:

```
$ cat holder

function holder
{
    print
    print -n "Type some text to continue: "
    read var1
    print "In function holder var1 is: $var1"
}
```

Autoloading Korn Shell Functions With the FPATH Variable

A function is treated in the same way as a Korn shell built-in statement. When the function is invoked, it is invoked in the current shell and not as a subprocess.

Before a function is invoked, the shell must know that the function exists. Therefore, you must define functions at the start of the shell script before any command-line command attempts to invoke the function.

If a function was created as a function file, you must create or modify the FPATH variable to include the directory name that contains the function file. Declare the FPATH variable before any command-line command attempts to invoke a function from one of the function files.

Note – The FPATH variable is similar to the PATH variable. It can contain the names of one or more directories, each separated by colons.

Directories that are listed in the FPATH variable should contain only function files.

The FPATH variable is an environment variable; for example:

```
$ FPATH=$HOME/function_dir ; export FPATH
```

The files that exist in the \$HOME/function_dir directory should all be function files. Ensure that no other type of files reside in that directory.

By using the FPATH variable, the functions can be autoloaded into a shell script and do not need to be declared in every script.

Autoloading Korn Shell Functions With the FPATH Variable

Also, by updating a function file (stored in a directory that is pointed to by the FPATH variable), any scripts that use that function autoload the updated version of the function file when the scripts are next invoked.

```
$ cat holdertest.ksh
#!/bin/ksh

# Script name: holdertest.ksh

FPATH=./funcs
export FPATH

print "Calling holder..."
holder

print
print "After the function var1 is: $var1"

$ ./holdertest.ksh
Calling holder...

Type some text to continue: shell scripts
In function holder var1 is: shell scripts

After the function var1 is: shell scripts
```

Exercise: Using Functions

Exercise: Using Functions

Exercise objective – Write shell scripts that include or use functions.

Preparation

Think about creating a function directory to store only your functions and consider what you would name this directory and where you can create it.

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change the directory to mod12/lab before beginning the exercise.

When writing the script on paper, you might want to execute statements on the system to ensure you have the correct statements and options to give you the prescribed output.

Tasks

1. Copy the parsepath.ksh file from the mod11/lab directory, and name the copy firstfunc.
2. Take the portion of the code that actually does the parsing and printing of the path names, and place it all in a function named parsepath.
3. Modify the code so that it does not take arguments from the command line, and it prints a USAGE message if any arguments are received.
4. Prompt the user for a path name, read the path name, call the function, and pass the path name to the function. Do this two or three times using a loop construct.
5. Create a directory named funcs.
6. Copy the firstfunc script to a /funcs/parsepath file.
7. Modify the /funcs/parsepath file so that it contains only the function code.
8. Copy the firstfunc script to a secondfunc file.

9. Modify the `secondfunc` file to remove the `parsepath` function.
10. Modify the `secondfunc` file to add a declaration for the `F PATH` variable and export the variable; for example:

```
F PATH= ./funcs  
export F PATH
```
11. Execute the `secondfunc` file, which should go to the directory specified in the `F PATH` variable. It should also search and find the `parsepath` function in the `parsepath` file.

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Copy the parsepath.ksh file from the mod08/lab directory, and name the copy firstfunc.

```
$ cp ../../mod11/lab/solutions/parsepath.ksh firstfunc.ksh
```

2. Take the portion of the code that actually does the parsing and printing of the path names, and place it all in a function named parsepath.

```
function parsepath
{
    name=$1

    print
    print "The pathname is: $name"

    while [[ -n $name ]]
    do
        print $name
        name=${name%/*}
    done

    print
}
```

3. Modify the code so that it does not take arguments from the command line, and it prints a USAGE message if any arguments are received.

```
# Purpose: To parse a long pathname.
#
# Name: firstfunc.ksh

if [ $# -ne 0 ]; then
    echo "Usage: firstfunc.ksh"
    exit 1
fi
```

Task Solutions

4. Prompt the user for a path name, read the path name, call the function, and pass the path name to the function. Do this two or three times using a loop construct.

```
print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
read pathname junk

while [[ -n $pathname ]]
do
    parsepath $pathname

    # Enter a control-D to quit
    print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
    read pathname junk
done
```

The following is the complete solution:

```
#!/bin/ksh

# Purpose: To parse a long pathname.
#
# Name: firstfunc.ksh

function parsepath
{
    name=$1

    print
    print "The pathname is: $name"

    while [[ -n $name ]]
    do
        print $name
        name=${name%/*}
    done

    print
}

if [ $# -ne 0 ]; then
    echo "Usage: firstfunc.ksh"
    exit 1
fi

print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
read pathname junk

while [[ -n $pathname ]]
do
    parsepath $pathname

    # Enter a control-D to quit
    print -n "Enter a pathname and I will parse it: [Control-D to quit]: "

    read pathname junk
done
```

Task Solutions

5. Create a directory named funcs.

```
$ mkdir funcs
```

6. Copy the firstfunc script to a /funcs/parsepath file.

```
$ cp firstfunc ./funcs/parsepath
```

7. Modify the /funcs/parsepath file so that it contains only the function code.

```
function parsepath
{
    name=$1

    print
    print "The pathname is: $name"

    while [[ -n $name ]]
    do
        print $name
        name=${name%/*}
    done

    print
}
```

8. Copy the firstfunc script to a secondfunc file.

```
$ cp firstfunc secondfunc
```

9. Modify the secondfunc file to remove the parsepath function.

```
#!/bin/ksh

# Purpose: To parse a long pathname.
#
# Name: secondfunc.ksh

if [ $# -ne 0 ] ; then
    echo "Usage: secondfunc.ksh"
    exit 1
fi

print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
read pathname junk

while [[ -n $pathname ]]
do
    parsepath $pathname

    # Enter a control-D to quit
    print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
    read pathname junk
done
```

10. Modify the secondfunc file to add a declaration for the FPATH variable and export the variable; for example:

```
FPATH=./funcs
export FPATH
```

11. Execute the secondfunc file, which should go to the directory specified in the FPATH variable. It should also search and find the parsepath function in the parsepath file.

Task Solutions

The following is the complete solution:

```
$ cat secondfunc.ksh
#!/bin/ksh

# Purpose: Use a function from FPATH directory to parse a long pathname.
#
# Name: secondfunc.ksh

FPATH=./funcs
export FPATH

if [ $# -ne 0 ]; then
    echo "Usage: secondfunc.ksh"
    exit 1
fi

print -n "Enter a pathname and I will parse it: [Control-D to quit]: "
read pathname junk

while [[ -n $pathname ]]
do
    parsepath $pathname

    # Enter a control-D to quit
    print -n "Enter a pathname and I will parse it: [Control-D to quit]: "

    read pathname junk
done

$ cat ./funcs/parsepath
function parsepath
{
    name=$1

    print
    print "The pathname is: $name"

    while [[ -n $name ]]
    do
        print $name
        name=${name%/*}
    done

    print
}
```

Module 13

Traps

Objectives

Upon completion of this module, you should be able to:

- Describe how the `trap` statement works
- Include `trap` statements in a script
- Use the `trap` statement to catch signals and handle errors

Shell Signal Values

Shell Signal Values

A signal is a message that some abnormal event has taken place or a message requesting another process do something. The signal is sent from one process to another process. Typically, a process sends a signal to one of its own subprocesses. You can obtain more information on signals from the `signal` and `kill` man pages.

You use the `kill` command to send signals to processes. The `root` user can send any signal to any process. Other users can only send signals to processes they own.

The following are some signals that can be sent from the keyboard:

- Send Signal 2 (`INT`) by typing Control-C.
- Send Signal 3 (`QUIT`) by typing Control-\.
- Send Signal 23 (`STOP`) by typing Control-S.
- Send Signal 24 (`TSTP`) by typing Control-Z.
- Send Signal 25 (`CONT`) by typing Control-Q.

The `INT` and `QUIT` signals cause the currently running process associated with the device (console or window) to terminate. The `TSTP` signal causes the process to stop or suspend execution. The job is put into the background; the process can be resumed by putting the job into the foreground.

Because most signals sent to a process executing a shell script cause the script to terminate, the next topic describes how to have a script avoid termination from specified signals.

The Korn shell has 46 defined signals. Each signal has a name and a number associated with it. Typing the `kill -l` command displays this list:

EXIT	HUP	INT	QUIT	ILL
TRAP	ABRT	EMT	FPE	KILL
BUS	SEGV	SYS	PIPE	ALRM
TERM	USR1	USR2	CLD	PWR
WINCH	URG	POLL	STOP	TSTP
CONT	TTIN	TTOU	VTALRM	PROF
XCPU	XFSZ	WAITING	LWP	FREEZE
THAW	CANCEL	LOST	RTMIN	RTMIN+1
RTMIN+2	RTMIN+3	RTMAX-3	RTMAX-2	RTMAX-1
RTMAX				

The numeric signal values range from 0 (EXIT) through 45 (RTMAX).

You can confirm the numeric value of a signal by executing the `kill` command with the `-l` option, followed by the signal name; for example:

```
$ kill -l EXIT  
0  
  
$ kill -l RTMAX  
45  
  
$ kill -l KILL  
9  
  
$ kill -l TSTP  
24
```

Shell Signal Values

If you know the number and want to learn or confirm the name of the signal, execute the `kill` command with the `-l` option, followed by the signal number.

```
$ kill -l 0  
EXIT
```

```
$ kill -l 45  
RTMAX
```

```
$ kill -l 9  
KILL
```

```
$ kill -l 24  
TSTP
```

You can send a signal to processes running on the system by using the `kill` command:

```
kill -signal pid
```

where *signal* is the signal number or the signal name, and *pid* is the process identification number for the process being sent the signal.

You can terminate a process by executing the `kill` command and using the `-9` option. The `-9` option sends the `KILL` signal to the process. You can use either the numeric value or the signal name as shown here.

```
kill -9 pid  
kill -KILL pid
```

When you do not specify a signal name or number, the `TERM` signal, Signal 15, is sent to the process.

Catching Signals With the `trap` Statement

You might want to write a script that you do not want the user to immediately terminate using Control-C, Control-\, or Control-Z. For example, the script might need to do some clean-up before exiting. To avoid having the script exit before the clean-up is done, you use the `trap` statement in your script to catch these signals.

The syntax for using the `trap` statement is:

```
trap 'action' signal [ signal2 ... signalx ]
```

where:

- *action* is a statement or several statements separated by semicolons. If an action is not provided, then no action is performed, but the signal is still “trapped.” Enclose the *action* within a pair of single quotes.
- *signal* is the name or number for the signal to be caught.

For example, if you want to trap the use of Control-C, use the following:

```
trap 'echo "Control-C not available"' INT
```

The statements to be executed can be more than one line as long as the closing quote is followed by the signal value or values to be trapped; for example:

```
trap 'echo "Control-C not available"  
echo "Core dumps not allowed"  
sleep 1  
continue' INT QUIT
```

The trap does not work if you attempt to trap the KILL or STOP signals. The shell does not let you catch these two signals, thereby ensuring that you can always terminate or stop a process. This means that shell scripts can still be terminated using the following command:

```
kill -9 script_PID  
kill -KILL script_PID
```

Also, the execution of the shell script can be suspended using the Control-S character because both signals can never be trapped within a Korn shell script.

The following statement tells the Korn shell to restore the original actions for the signal.

```
trap - signal
```

Example of Using the trap Statement

Example of Using the trap Statement

The following trapsig.ksh example script sets up a `trap` statement for the signals `INT` (Control-C), `QUIT` (Control-\) and `TSTP` (Control-Z). The script does not illustrate any practical application; it just shows how to install signal handling for these signals.

After the `trap` statement, a `while` loop is entered that prints the string `rolling ...` and waits for user input. The `while` loop and the script terminate when the user types in the `dough` string.

The script cannot be terminated by typing Control-C, Control-\, or Control-Z.

The example output shows the script being executed. The user presses the Return key after the first `rolling ...` prompt. The user then types a `d` to the second `rolling ...` prompt and an `s` to the third.

The user types a Control-C, a Control-\, and a Control-Z to the next `rolling ...` prompts, which causes the appropriate `trap` statements to execute.

Finally, the user types the string `dough`, and the script terminates.

```
$ cat trapsig.ksh
#!/bin/ksh

# Script name: trapsig.ksh

trap 'print "Control-C cannot terminate this script."' INT
trap 'print "Control-\ cannot terminate this script."' QUIT
trap 'print "Control-Z cannot terminate this script."' TSTP

print "Enter any string (type 'dough' to exit)."
while (( 1 ))
do
    print -n "Rolling..."
    read string

    if [[ "$string" = "dough" ]]
    then
        break
    fi
done

print "Exiting normally"
```

```
$ trapsig.ksh
Enter any string (type 'dough' to exit).
Rolling...<CR>
Rolling...d
Rolling...s
Rolling...^C
Control-C cannot terminate this script.
Rolling...^\
Control-\ cannot terminate this script.
Rolling...^Z
Control-Z cannot terminate this script.
Rolling...dough
Exiting normally
```

Example of Using the trap Statement

The following is another example of using the trap statement. This example uses two scripts, parent and child. The parent script passes the signal to the child script and the messages are from the script that is called.

```
$ cat parent
# trap - SIGNAL - cancel redefinitions
# trap "cmd" SIGNAL - "cmd" is executed, signal is inherited
to child
# trap : SIGNAL - nothing is executed, signal is inherited
to child
# trap "" SIGNAL - nothing is executed, signal is not
inherited to child

#!/bin/ksh
#
# Script name: parent

trap "" 3 # ignore, no forward to children
trap : 2 # ignore, forward to children

./child

$ cat child
#
# Script name: child

trap 'print "Control-C cannot terminate this script."' INT
trap 'print "Control-\\" cannot terminate this script."' QUIT

while true
do
    echo "Type ^D to exit..."
    read || break
done

$ ./parent
Type ^D to exit...
^\
Type ^D to exit...
^C
Control-C cannot terminate this script.
Type ^D to exit...
^\
Type ^D to exit...
$
```

Catching User Errors With the `trap` Statement

In addition to catching signals, you can use the `trap` statement to take specified actions when an error occurs in the execution of a script.

The syntax for this type of `trap` statement is:

```
trap 'action' ERR
```

The value of the `$?` variable indicates when the `trap` statement is to be executed. It holds the exit (error) status of the previously executed command or statement, so any nonzero value indicates an error. Thus the `trap` statement is executed whenever `$?` becomes nonzero.

The following `traperr1.ksh` example script requires users to enter a negative integer number (`-1`) if they want to exit.

If a user enters an integer that is not a `-1`, the script prints the square of the number. The script then requests another integer until the user quits the script by entering a `-1`.

The example output shows the script being executed. The user enters the letter `x`. This is not an integer. The user's input is read into the variable `num`, which is declared to only hold integer data types. Without a `trap` statement the shell prints an error message and exits the script. You can avoid this problem by using a `trap` statement, which is shown in the next section.

Catching User Errors With the trap Statement

```
$ cat traperr1.ksh
#!/bin/ksh

# Script name: traperr1.ksh

integer num=2

while (( 1 ))
do
    read num?"Enter any number ( -1 to exit ): "

    if (( num == -1 ))
    then
        break
    else
        print "Square of $num is $(( num * num )).\n"
    fi
done

print "Exiting normally"

$ ./traperr1.ksh
Enter any number (-1 to exit): r
traperr1.ksh[9]: r: bad number
Square of 2 is 4.

Enter any number (-1 to exit): -1
```

Example of Using the trap Statement With the ERR Signal

The following `trapsig2.ksh` script is the `traperr1.ksh` script rewritten with a `trap` statement, a test of the exit status, and an `exec` statement to redirect standard error messages to `/dev/null`.

The redirection of standard error messages is done so that the user does not see the error messages the shell would otherwise print to the screen. You want the user to see just the message you set up with the `trap` statement if an error occurs.

The value of `$?` is saved in the `status` variable immediately after the `read` statement is executed so that you can check later whether the `read` statement successfully read an integer.

The `if` statement checks to see if the user entered a `-1`. If so, the script breaks out of the `while` loop and terminates. If the user did not enter a `-1` and the value of `status` is `0` (indicating the `read` statement read an integer), the square of the integer entered by the user is printed and the user is again prompted for a number.

If the user does not enter an integer, the value of `$?` is nonzero and it is trapped. The `print` statement within the `trap` statement is then executed. Control then returns to the `while` loop, prompting the user for another number.

The example output shows the script being executed, with the user typing in an integer, then the letter `x`, another integer, and finally `-1` to exit the script. The messages output by the script for the corresponding input from the user are what is expected, based on the preceding description.

Example of Using the trap Statement With the ERR Signal

```
$ cat trapsig2.ksh
#!/bin/ksh

# Script name: trapsig2.ksh

integer num

exec 2> /dev/null
trap 'print "You did not enter an integer.\n"' ERR

while (( 1 ))
do
    print -n "Enter any number ( -1 to exit ): "
    read num

    status=$?

    if (( num == -1 ))
    then
        break
    elif (( status == 0 ))
    then
        print "Square of $num is $(( num * num )) . \n"
    fi
done

print "Exiting normally"
```

```
$ ./trapsig2.ksh
```

```
Enter any number ( -1 to exit ): 3
Square of 3 is 9.
```

```
Enter any number ( -1 to exit ): r
You did not enter an integer.
```

```
Enter any number ( -1 to exit ): 8
Square of 8 is 64.
```

```
Enter any number ( -1 to exit ): -1
Exiting normally
```

When to Declare a trap Statement

To trap a signal any time during the execution of a shell script, define a `trap` statement at the start of the script.

Alternatively, to trap a signal only when certain command lines are to be executed, you can turn on the `trap` statement before the appropriate command lines and turn off the `trap` statement after the command lines have been executed.

If a loop is being used, a `trap` statement can include the `continue` statement to make the loop start again from its beginning.

You can also trap the `EXIT` signal so that certain statements are executed only when the shell script is being terminated with no errors. For example, if a shell script created temporary files, you could ensure that these are removed using a trap of the `EXIT` signal value.

```
trap 'rm -f /tmp/tfile* ; exit 0' EXIT
```

Note – The preceding example includes an `exit` statement within the list of statements to be executed when the trap occurs. This is necessary to exit from the script.

The following example is a copy of the `/etc/profile` file with some added comments to explain the various `trap` statements.

```
#ident  "@(#)profile  1.18  98/10/03 SMI  /* SVr4.0 1.3  */  
  
# The profile that all logins get before using their own .profile.  
  
trap "" 2 3      # trap INT (Control-C) and QUIT (Control-\)  
                  # and give no feedback  
export LOGNAME PATH  
  
if [ "$TERM" = "" ]  
then  
    if /bin/i386  
    then  
        TERM=sun-color  
    else  
        TERM=sun  
    fi  
    export TERM
```

When to Declare a trap Statement

```
fi

# Login and -su shells get /etc/profile services.
# -rsh is given its environment in its .profile.

case "$0" in
  -sh | -ksh | -jsh)

    if [ ! -f .hushlogin ]
    then
      /usr/sbin/quota
        #      Allow the user to break the Message-Of-The-Day only.
        #      The user does this by using Control-C (INT).
        #      Note: QUIT (Control-\) is still trapped (disabled).
      trap "trap '' 2" 2
      /bin/cat -s /etc/motd
      trap "" 2      # trap Control-C (INT) and give no feedback.

      /bin/mail -E
      case $? in
        0)
          echo "You have new mail."
          ;;
        2)
          echo "You have mail."
          ;;
      esac
    fi
  esac

umask 022
trap 2 3      # Allow the user to terminate with Control-C (INT) or
               # Control-\(QUIT)
```

Exercise: Using Traps

Exercise objective – In this lab, you modify a shell script to trap the Control-C signal.

Preparation

Refer to the lecture notes as necessary to answer the following questions and perform the following tasks.

Change the directory to mod13/lab before beginning the exercise.

You need a copy of the adduser script from Module 9 lab exercise. If you did not complete this exercise, you can copy the solution to complete this lab exercise.

Tasks

1. Copy the adduser script from the mod9/lab/solutions directory.
2. Modify the script as follows:
 - a. Execute the adduser script with a new user name, and execute a Control-C at the shell menu prompt.
 - b. Add a trap to the adduser script to catch the INT signal so the user can not terminate with that signal. The trap should notify the user that Control-C is not available. Place the trap statement after the test for the command-line arguments.
 - c. Execute the adduser script with a new user name and press Control-C at the shell menu prompt, which triggers your notification that Control-C is disabled.

Exercise Summary



Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Task Solutions

1. Copy the adduser script from the mod9/lab/solutions directory.
2. Modify the script as follows:
 - a. Execute the adduser script with a new user name, and execute a Control-C at the shell menu prompt.

```
$ ./adduser michael
```

```
The uid for michael is: 1003
```

```
What primary group should belong to?
```

```
Please specify the group name: staff
```

- 1) csh
- 2) ksh
- 3) sh

```
Enter the number: ^C
```

- b. Add a trap to the adduser script to catch the INT signal so the user can not terminate with that signal. The trap should notify the user that Control-C is not available. Place the trap statement after the test for the command-line arguments.

```
trap 'banner "No Control-C"' 2
```

- c. Execute the adduser script with a new user name and press Control-C at the shell menu prompt, which triggers your notification that Control-C is disabled.

Task Solutions

The following is the complete solution:

```
#!/bin/ksh

# Purpose: To write a script to add user to the system. Generate a
#           menu for the login shell choices.
#
# Name: adduser.ksh

# Check for only a single command line argument
if [ $# -ne 1 ]
then
    echo "Usage: adduser.ksh newusername"
    exit 1
fi

# Add a trap for Control-C
trap 'banner "No Control-C"' 2

# Assign the value of $1 to the variable name
name=$1

if grep "^\$name:" ./mypasswd > /dev/null 2>&1
then
    echo "The username $1 is already in use, here "
    echo "is the entry from the ./mypasswd file."
    echo
    grep "^\$name:" ./mypasswd
    exit 2
fi

# Use awk to extract all the UIDs into the temp file currentuid
awk -F: '{print $3}' ./mypasswd > ./currentuid

# Use sed to remove all 1, 2, and 3 digit UIDs from currentuid, and
# place the output in the temp file currentuid2
sed -e '/^.$/d' -e '/^..$/d' -e '/^...$/d' ./currentuid > ./currentuid2

# Use sed to remove UIDs 60001, 60002, and 65534 from currentuid2,
```

```
# and place the output in the temp file currentuid3
sed -e '/^6000[12]$/d' -e '/^65534$/d' ./currentuid2 > ./currentuid3

# Sort the UIDs in currentuid3 numerically
sort -n ./currentuid3 > ./currentuid4

# Calculate the next available UID
lastuid='sed -n '$p' ./currentuid4'

uid='expr $lastuid + 1'
echo "The uid for $name is: $uid"

# Prompt for a group name and verify whether it exists in /etc/group
echo "What primary group should $a belong to? "
echo "Please specify the group name: \c"
read group junk

grep "^\$group:" /etc/group > /dev/null 2>&1
if [ $? -ne 0 ]
then
    echo "The group $group is not valid."
    echo
    exit 2
fi

PS3="Enter the number: "
select ushell in csh ksh sh
do
    case $ushell in
        csh)
            print "$ushell was selected."
            break
    ;;
    ksh)
        print "$ushell was selected."
        break
    ;;
    ;;
```

Task Solutions

```
sh)
    print "$ushell was selected."
    break
    ;;

*)
    print "Unknown choice, try again."
    ;;
esac

done

if [ $ushell = "csh" -o $ushell = "ksh" -o $ushell = "sh" ]
then
    echo "Please enter the comment information for the user:"
    read comment

    echo "The new user entry looks like this:"
    echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell"
    echo " "
    echo "New user $1 has been added to the ./mypasswd file"
    echo "$1:x:$uid:$group:$comment:/export/home/$1:$ushell" >>
./mypasswd
else
    echo "Sorry, that is not a valid shell, exiting..."
fi
```

Advanced nawk Programming

Appendix A

This appendix covers more advanced nawk programming concepts.

Programming Concepts

The awk programming language contains many of the programming concepts that were described throughout the course for scripting:

- Conditionals, such as the `if` statement
- Loops, such as the following
 - The `while` loop
 - The `do while` loop
 - The `for` loop

The `if` Statement

The `if` statement can have two branches: an `if` branch and an `else` branch. If the condition is true, the `if` branch is executed; if the condition is false, the `else` branch is executed.

```
if (condition) {  
    statements  
} else {  
    statements }
```

You can nest `if` statements. When examining nested `if` statements and one or more of the `if` statements has an `else` statement, it is difficult to know with which `if` the `else` is operating. The simple rule is: Each `else` works on the closest `if` that does not yet have its own `else`; for example:

```
if (condition) {  
    if (condition) {  
        statements  
    } else {  
        statements  
    }  
}
```

In the previous example, the `else` goes with the second `if` statement. Using indentation helps alleviate some of the potential confusion.

Conditional Printing With the nawk Language

In its simplest form, the `if` statement tests a condition and, if that condition is true, the statements are executed. To compare two numbers in the condition, use one of the following relational operators:

<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

For example:

```
$ nawk '{num = $5 / $6; if (num > 6) print $1, num}' data.file
southeast 7.14286
central 6.06383
```

```
$ nawk '{if ($6 > .92) print $2, $6, $4}' data.file | sort
CT .94 Watson
NE .94 Nichols
NW .98 Craig
SO .95 Chin
WE .97 Kelly
```

String Comparisons and Relational and Logical Operators

To compare two strings in a conditional, use one of the relational operators shown below:

==	Equal to
!=	Not equal to
~	Contains a /RE/
! ~	Does not contain a /RE/

The ~ and !~ operators require some explanation. The left operand is the region of text being searched, which can be the entire record (\$0) or a specific field of the record (\$1). The right operand is a regular expression in slashes (/RE/) that is being searched for. If the text contains the regular expression, then the condition is true. You can use all the regular expression characters, that have been described, in the conditional's regular expression.

The next examples show a pattern as a conditional statement (numeric or string). Each record is tested to see if the conditional is true for that record. If it is true, the ACTION (printing, in this case) is taken on that record. In the second example, an ACTION is not listed, so the default action is taken, to print the entire input record.

```
$ awk '{if ($2 ~ /E/) print $2, $1}' data.file
WE western
SE southeast
EA eastern
NE northeast
```

```
$ awk '$2 !~ /E/' data.file
northwest      NW      Joel Craig      3.0 .98 3      4
southwest      SW      Chris Foster    2.7 .8   2      18
southern        SO      May Chin       5.1 .95 4      15
north           NO      Val Shultz     4.5 .89 5      9
central         CT      Sheri Watson   5.7 .94 5     13
```

Logical Operators

You use two logical operators between any two conditional expressions to join the expressions. The conditional expressions can be either numeric comparisons or string comparisons or a mixture of numeric and string comparisons. The logical AND requires that both of the joined expressions be true before the combination expression is true. The OR operator requires that only one of them is true. A unary NOT operator exists (the exclamation point) and you can use it to invert the logical value of an expression.

&&	Logical AND
	Logical OR
!	Logical NOT

For example:

```
$ awk '{if ( $2 ~ /E/ && $6 > .92) print $2, $6, $4}' data.file
WE .97 Kelly
NE .94 Nichols
```

The while Loop in the nawk Language

When loops are included in a nawk script or command, the loop executes one time for each record that is processed. For each record, the loop executes until the condition fails.

The `while` loop is the simplest loop available. It tests the same types of conditions that were used by the `if` statement. In loop constructs, the single statement, or multiple statements enclosed in curly brackets, is known as the body of the loop. If the condition is true, the body of the loop is executed; if the condition is false, nawk continues with the nawk program. The syntax for a `while` loop is:

```
while (condition)
      statement

while (condition) {
    statements
}
```

The following example uses a `while` loop to print out each field of every record on a line by itself. It prints a blank line after each record is reported.

```
{ i = 1 }
{ while ( i <= NF )
    { print $i ; i++ } }

{ print "\n" }
```

The do while Loop

A `do while` loop follows the same philosophy of a `while` loop except that the condition is tested after the body of the loop is executed. The syntax is:

```
do
    { statements }
while (condition)
```

The while Loop in the nawk Language

The primary difference between a `while` loop and a `do while` loop is that the `do while` loop executes at least once. For example, suppose that the condition is false for a `while` loop and for a `do while` loop. In the `while` loop, the condition is tested first and because it is false, the body of the loop is not executed. In the `do while` loop, the body of the loop executes first, and then the condition is tested. Because the condition is false, the body of the loop is no longer executed, but it did execute once.

The following example uses a `do while` loop to print out each field of every record on a line by itself. It prints a blank line after each record is reported.

```
{ i = 1 }

{ do {
    { print $i ; i++ }
} while ( i <= NF )
}

{ print "\n" }
```

The for Loop in the nawk Language

The for loop has two varieties, one borrowed from the C programming language and the other to process nawk arrays.

```
for (setup_statement ; condition ; step_statement)
    {statements }
```

```
for (indexvar in arrayname)
    {statements }
```

The following example shows the first type of for loop. The loop executes one time for every field in a record. When the value of a reaches the value of NF (the number of fields in the current record), the loop terminates, and the next record is read from the file to begin the process again.

```
$ cat for.nawk
{ for (a = 1; a <= NF; a++) print $a }

$ nawk -f for.nawk data.file
northwest
NW
Joel
Craig
3.0
.98
3
4
western
WE
Sharon
Kelly
5.3
<output truncated>
```

Using Loops With Arrays

The second type of `for` loop is used explicitly with `nawk` arrays, and it executes once for each array element.

```
for (index_var in array_name) { statements }
```

The `index_var` variable contains each array subscript in turn.

With `nawk` arrays, it is not always possible to know how many indices there are. Therefore, the syntax of this `for` loop allows the processing of arrays by creating a special variable (the first word in the `for` loop parentheses) to hold the value of each of the array indices. The second word in the parentheses is always `in` and the third word is the name of the array to be processed. The `for` loop executes one time for each index of the array (each element of the array). Each time through, the value of the special variable (the first word in the parentheses) changes to the next index of the array.

Nonnumeric Array Indices

Arrays in awk differ from those in some programming languages in that they do not have to be declared before they can be used. Further, the index (the value in the square brackets) does not have to be numeric. The index value is associative, which means that it can be anything that relates to the data being stored.

```
$ cat bigcats
lions
tigers
leopards
tigers
leopards
leopards
tigers
leopards

$ nawk '{a[$1]++}
> END {for (c in a) print c, a[c]}' bigcats
tigers 3
lions 1
leopards 4
```

In the preceding example, the file being processed contains a list of big cats. The awk statement reads each record, creating an array called a. The index of the element is the first field of the record (the name of the cat). The value of each array element always starts at 0 (when the array element is created). That element is incremented (with the ++ operator). Thus, at the end of the first statement, a variable with the name:

a[lions]

is given the value of 1.

This continues through the entire file. When a cat that has already been processed before it is read, the array element for that index already exists. Therefore, the current value is incremented by 1. So, at the end of the file, one array element exists for each type of cat and the value of each element is the number of times that cat was encountered in the file.

Nonnumeric Array Indices

The `for` loop creates a variable `c` to represent each of the indices (it is not known how many there are going to be) and processes the `a` array. For each element, the value of `c` (a cat name] and the value of the element (the number of times the cat name has been encountered) prints.

Discussion – What would the following `nawk` statement do to a file?



```
$ nawk '{a[NR] = $0};  
> END {for (c = NR; c > 0; c--) print a[c]}' data.file
```

The break and continue Statements

You use the `break` statement to break out of a loop or stop its execution entirely. The statements in the loop body that follow the `break` statement are not executed, and execution resumes with the first line after the body of the loop.

```
while (condition) {  
    statements  
    break  
    statements  
}  
more_nawk_statements
```

The continue Statement

The `continue` statement stops the current iteration of the loop body, returning to the loop control expression.

```
while (condition) {  
    statements  
    continue  
    statements  
}
```

The `continue` statement stops the execution of the body of statements in a loop, and execution is sent to the control portion of the loop. In a `while` loop or a `do while` loop, the control portion is the condition expression. The condition is tested again, and if it is true, then the body of the loop is entered again, and the process continues.

In a `for` loop (not the kind explicitly used with arrays), execution moves to the step portion of the loop (the third expression in the parentheses). The step statement is executed and the condition is tested again. If the condition is true, the loop is executed again. In a `for` loop that is used with arrays (`for (indexvar in arrayname)`), execution begins at the top of the loop body again but with the next array element being processed. The old array element is forgotten and dropped.

The break and continue Statements

The effect of the `continue` statement for each of the types of looping statements is:

`while` and `do while`

The condition is tested again

`for (setup_statement; condition; step_statement)`

The step is taken, and then the condition is tested

`for (indexvar in arrayname)`

The next array element is processed

The next and exit Statements

Use the `next` and `exit` statements for program flow control.

The `next` statement stops processing the current record, reads the next record from the input file, and starts over at the beginning of the `nawk` program.

The `exit` statement terminates the `nawk` program. An exit status might be given, and it passes to `nawk`'s parent process.

It is possible to stop processing any record of a file. The next record is read in and the `nawk` program begins again with this new record. The statement that tells `nawk` to get the next record is the `next` statement. Typically, a test is performed and, if the test is true (or false, depending on what is appropriate), the current record is discarded and the next record is read.

The `exit` statement is used in a similar manner in that a test is usually made. If the test is true (or false, depending on what is appropriate) then the program terminates, which is accomplished with the `exit` statement.

When `exit` is called, the entire `nawk` process quits. The only argument allowed for `exit` is an optional exit status. As the `nawk` process terminates, it passes an exit status to its parent process. If an argument is used with `exit`, this is the exit status that is returned to `nawk`'s parent process.

User-Defined Functions

User-Defined Functions

You can create user-defined functions in a nawk script file using the `func` or `function` keywords. Placement in the file is not important; the function definition can occur anywhere in the nawk script.

The function can take arguments (local to the function only) or use any existing variable.

The syntax of a function definition is:

```
func function_name ([optional_arg] ...) { statements }
function function_name ([optional_arg] ...) { statements }
```

The `function_name` is how the function is called at some other point in the program. Arguments are allowed but not required. In the definition of the function, only the names of the arguments are required. These arguments can be accessed only within the body of the function (the statements inside the curly braces). You might, however, access any other variable that is being used in the nawk script from within a function.

For example:

```
$ cat func.nawk
func MAX(val1, val2) {
    if (val1 > val2)
        return val1
    else
        return val2
}
BEGIN {largest = 0}
{largest = MAX(largest, $5)}
END {print largest}

$ nawk -f func.nawk data.file
5.7
```

You can use the `return` statement to return a value from the function. Use this returned value to assign a value to a variable or to perform a test of some sort with an `if` statement. To perform such a test, the function is called within the `if` statement. If the function returns a nonzero value, the `if` statement considers the test to be true; if the function returns a zero value, the `if` statement considers the test to be false, and the `if` statements are not executed; for example:

```
if (function_call()) { statements }
```

Command-Line Arguments

When a `nawk` command is executed, the number of words on the command line is stored in the `ARGC` built-in variable. The words themselves are stored in the `ARGV` built-in array. Exceptions to this are options to `nawk`, such as the `-f awkscript` and the `-F<char>` options. These options are not counted by `ARGC` nor are they stored as elements in the `ARGV` variable.

For example:

```
$ cat args.nawk
BEGIN {
    print ( "The number of words entered is " ARGC )
    print ( "The value of var is " var )
    print ( "The first word is " ARGV[0] )
    print ( "The second word is " ARGV[1] )
}
{ print ( "The value of var is " var ) }
```



```
$ nawk -f args.nawk var=32 junkfile
The number of words entered is 3
The value of var is
The first word is nawk
The second word is var=32
The value of var is 32
```

Note – The distinction between `nawk` and `awk` is significant in regard to command-line variables. The built-in variables, `ARGC` and `ARGV`, are not acknowledged by the `awk` language.

Using Built-in Variables

As nawk is processing an input file, it uses several variables. You can provide a value to some of these variables, while other variables are set by the nawk language. Table 0-1 lists the nawk built-in variables.

Table A-1 nawk Built-in Variables

Name	Default Value	Description
OFS	space	The output field separator
FS	space or tab	The input field separator
ORS	\n (newline)	The output record separator
RS	\n (newline)	The input record separator
OFMT	% .6g	The output format for numbers
NF		The number of fields in current record
NR		The number of records from beginning of the first input file
FNR		The number of records from beginning of current file
ARGC		The number of command-line arguments
ARGV		An array of command-line arguments
FILENAME		The name of current input file
RSTART		The index into a string, set by <code>match()</code>
RLENGTH		The length of substring, set by <code>match()</code>

Built-in Arithmetic Functions

Table A-2 lists the awk built-in arithmetic functions.

Table A-2 Built-in awk Arithmetic Functions

Command	Syntax	Meaning
atan2	atan2 (<i>y</i> , <i>x</i>)	Returns the arc tangent of <i>y/x</i> in radians.
cos	cos (<i>x</i>)	Returns the cosine of <i>x</i> in radians.
exp	exp (<i>arg</i>)	Returns the natural exponent of <i>arg</i> .
int	int (<i>arg</i>)	Returns the integer value of <i>arg</i> .
log	log (<i>arg</i>)	Returns the natural logarithm of <i>arg</i> .
rand	rand()	Generates a random number between 0 and 1. This function returns the same series of numbers each time the script is executed, unless the random number generator is seeded using the srand() function.
sin	sin (<i>x</i>)	Returns the sine of <i>x</i> in radians.
sqrt	sqrt (<i>arg</i>)	Returns the square root of <i>arg</i> .
srand	srand (<i>expr</i>)	Sets a new seed for the random number generator. The default is the time of day. It returns the old seed.

Built-in String Functions

Table A-3 lists the awk built-in string functions.

Table A-3 Built-in awk String Functions

Command	Syntax	Meaning
gsub	gsub(<i>x, s [, t]</i>)	Globally substitutes <i>s</i> for each match of the regular expression <i>x</i> in the string <i>t</i> . Returns the number of substitutions. If <i>t</i> is not supplied, the default is \$0.
index	index(<i>str, substr</i>)	Returns position of first substring <i>substr</i> in string <i>str</i> , or 0 if not found.
length	length(<i>arg</i>)	Returns the length of <i>arg</i> .
match	match(<i>s, r</i>)	Matches the regular expression pattern <i>r</i> in the string <i>s</i> . It returns the position in <i>s</i> where the match begins or 0 if no occurrences are found. Sets the values of RSTART and RLENGTH.
split	split(<i>string, array [, sep]</i>)	Splits the string into elements of <i>array</i> . <i>string</i> at each occurrence of separator <i>sep</i> . If <i>sep</i> is not specified, FS is used. The number of array elements created is returned.
sub	sub(<i>r, s [, t]</i>)	Substitutes <i>s</i> for the first match of the regular expression <i>r</i> in the string <i>t</i> . It returns 1 if successful; 0 otherwise. If <i>t</i> is not supplied, the default is \$0.
substr	substr(<i>string, m [, n]</i>)	Returns a substring of <i>string</i> beginning at character position <i>m</i> and consisting of the next <i>n</i> characters. If <i>n</i> is not supplied, all characters to the end of the string are returned.
tolower	tolower(<i>str</i>)	Translates all uppercase characters in <i>str</i> to lowercase and returns the new string.

Built-in String Functions

Table A-3 Built-in nawk String Functions (Continued)

Command	Syntax	Meaning
toupper	toupper(<i>str</i>)	Translates all lowercase characters in <i>str</i> to uppercase and returns the new string.

Built-in I/O Processing Functions

Table A-4 lists the `nawk` input/output processing functions.

Table A-4 Built-in `nawk` I/O Processing Functions

Command	Syntax	Meaning
close	<code>close (filename-expr)</code> <code>close (command-expr)</code>	Closes a file or a pipe. The expression is the same argument that opened the file or pipe.
delete	<code>delete array[element]</code>	Deletes an element of <code>array</code> .
getline	<code>getline [var] [<file command getline [var]]</code>	Reads the next line of input from <code>stdin</code> , an input file, or a pipe.
next	<code>next</code>	Reads the next input line, and starts a new cycle through pattern or procedure statements.
print	<code>print [args] [destination]</code>	Prints <code>arg</code> to the <code>destination</code> . If <code>arg</code> is not specified, prints <code>\$0</code> . If <code>destination</code> is not specified, prints to <code>stdout</code> .
printf	<code>printf (format [, expression(s)] [destination])</code>	Prints according to the formatted expressions.
sprintf	<code>sprintf(format [, expression(s)])</code>	Returns the value of formatted expressions. The data is formatted but not printed.
system	<code>system(command)</code>	Executes the specified UNIX commands and returns the status.

The printf() Statement

The printf() Statement

The syntax of the printf() statement is:

```
printf ("string_of_characters" [ , data_values ] )
```

where the *data_values* are used to fill in placeholders in the *string_of_characters*. The placeholders are:

%d	Integer value
%f	Floating point value
%c	Character value
%s	String value

The following example uses a printf statement to output Fields 4, 6, and 7 of the data.file file.

```
$ awk '{ printf "%s %5.1f %d \n", $4, $6, $7 }' data.file
Craig    1.0 3
Kelly    1.0 5
Foster   0.8 2
Chin     0.9 4
Johnson  0.7 4
Beal     0.8 5
Nichols  0.9 3
Shultz   0.9 5
Watson   0.9 5
```

Built-in Operators

There are many built-in operators and combination assignment operators for the awk language. These are listed in Table A-5, Table A-6, and Table A-7 on page Appendix A-26.

Table A-5 Relational Operators

Symbol	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>~</code>	Contains /RE/
<code>!~</code>	Does not contain /RE/

Table A-6 Mathematical Operators

Symbol	Meaning
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulus
<code>^</code>	Exponentiation
<code>++</code>	Increment variable by 1
<code>--</code>	Decrement variable by 1

Built-in Operators

Table A-7 Combination Assignment Arithmetic Operators

Symbol	Meaning
=	Assignment
+=	Add to
-=	Subtract from
*=	Multiply by
/=	Divide by
%=	Modulus by
^=	Raise to the power

Additional grep Functionality

Appendix B

In addition to the functionality described in the course, there are more metacharacters and regular expressions. For a complete listing of all features, refer to the `grep` man page. This appendix describes a few more of the most commonly used features.

Regular Expression Metacharacters

The following table shows the additional regular expression metacharacters.

Metacharacter	Function
[^]	Matches any one character except one from the specified set
\(\)	A grouping of RE characters that can be repeated

Character Classes

Sometimes it is easier to specify what you do not want to match. In this case, specify a character class in which the first character within the brackets is a `^`. Be careful, because the results often surprise first-time users of this metacharacter.

```
$ grep '[^1-5]' /etc/group
root::0:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
<output truncated>
```

You might have been surprised by the output. The command specifies to find any line that contains at least one character that is not in the character class. Because the first character of the first line is an `r`, the line is printed even though there is a `0` in the line.

Consider the following example that outputs lines from the `/etc/group` file that contain more than one user listed in the final field. You need to be extremely familiar with the syntax of the file to realize that the user list is uniquely identified by a comma-separated list of user names. Otherwise, only characters, digits, and the colon are normally present in the entry.

```
$ grep '[^a-zA-Z:0-9]' /etc/group
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
uucp::5:root,uucp
tty::7:root,tty,adm
lp::8:root,lp,adm
nuucp::9:root,nuucp
daemon::12:root,daemon
```

If the first symbol following the `[` is a `^`, then a negative character class is specified. In this case, the string matches all characters except those enclosed in the brackets. For example, `[^a-z]` matches any character that is not a lowercase letter.

For example, to find all lines that contain a character that is not `a`, `b`, `c`, or `d`, use the following command:

```
$ grep '[^abcd]' data1
```

Character Classes

efgh
city
ijkl
aeiou
any

If you list only a, b, and c in the expression, the results are different. Because the line also contains a d, using a, b, and c in the search pattern does not sufficiently remove the line from the results.

```
$ grep '[^abc]' data1  
abcd  
efgh  
city  
ijkl  
aeiou  
any
```

Tagged Regular Expressions

A backslash and a left parenthesis and a backslash and a right parenthesis delimit a group of characters within a regular expression that can be repeated later in the same regular expression. Each tag created within a regular expression with the backslashes and parentheses is given a numerical value. The first tagged regular expression is automatically given the value 1, the second tagged regular expression is given the value 2, and so on.

You use the backslashes and parentheses when a pattern you are searching with is repeated within the expression, hence, within the line of text. So, if you are trying to find lines that contain more than one occurrence of the pattern the in the nfs.server script, you would use the following command:

```
$ grep '\(the\).*\1' /etc/init.d/nfs.server
# lines, then run shareall to export them, and then start up mountd
# logging enabled, or they were shared in the previous session
# When the system comes up umask is not set; so set the mode now
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Additional sed Functionality

Appendix C

In addition to the functionality shown in Module 5, there are more commands and features in `sed`. For a complete listing of all features, refer to the `sed` man page. This appendix describes a few more of the most commonly used features.

Editing Commands

Editing Commands

Additional `sed` editor commands are shown in the following table.

Command	Function
<code>a\</code>	Appends text
<code>c\</code>	Changes text in the current line with new text
<code>i\</code>	Inserts text above current line

The `sed` command has an additional option, `-e`, which allows multiple edits on a line.

Placing Multiple Edits in a Single `sed` Command

There are times when you want to perform several edits on the file. Rather than using multiple `sed` commands, you can use the `-e` option to place the edits in the same command line.

The edits are performed in the order that you specify.

```
$ sed -e '2,4d' -e 's/north/North/' data.file
Northwest      NW    Joel Craig     3.0 .98 3        4
southeast      SE    Derek Johnson  5.0 .70 4        17
eastern        EA    Susan Beal     4.4 .8 5        20
Northeast      NE    TJ Nichols    5.1 .94 3        13
North          NO    Val Shultz    4.5 .89 5        9
central        CT    Sheri Watson   5.7 .94 5        13
```

Delete lines carefully. You might perform edits on a line of text and subsequently remove that line of text from the output. This occurs in the following example with Line 1 of the input file.

```
$ sed -e 's/north/North/' -e '1,4d' data.file
southeast      SE    Derek Johnson  5.0 .70 4        17
eastern        EA    Susan Beal     4.4 .8 5        20
Northeast      NE    TJ Nichols    5.1 .94 3        13
North          NO    Val Shultz    4.5 .89 5        9
central        CT    Sheri Watson   5.7 .94 5        13
```

With each edit you add, the command line becomes more prone to errors. The following example requires three sets of quotes, one for each of the three edits.

```
$ sed -e '1,4d' -e 's/north/North/' -e 's/^east/East/' data.file
southeast      SE      Derek Johnson  5.0 .70 4      17
Eastern        EA      Susan Beal     4.4 .8   5      20
Northeast      NE      TJ Nichols    5.1 .94 3      13
North          NO      Val Shultz    4.5 .89 5      9
central        CT      Sheri Watson  5.7 .94 5      13
```

Multiple search and substitute commands might compound and lead to unexpected results.

```
$ cat animals
pig
cow
horse
```

```
$ sed -e 's/pig/cow/' -e 's/cow/horse/' animals
horse
horse
horse
```

Using sed to Append, Insert, or Change Text

The append command adds a line of text after the matched line. The append command is `a\``. The text to be appended begins on the next line and continues through the subsequent line that is not terminated with a backslash.

Because of the embedded backslash in the syntax for the append, insert, or change commands, these commands are normally put in script files that are accessed with the `-f` option.

In the following example, a new line is added after each line containing the string `central`.

Editing Commands

```
$ cat script2.sed
/central/a\
*** Currently there is 1 employee in the central region. ***
```

```
$ sed -f script2.sed data.file
northwest      NW      Joel Craig      3.0 .98 3      4
western        WE      Sharon Kelly     5.3 .97 5      23
southwest      SW      Chris Foster    2.7 .8 2      18
southern        SO      May Chin       5.1 .95 4      15
southeast       SE      Derek Johnson   5.0 .70 4      17
eastern         EA      Susan Beal      4.4 .8 5      20
northeast       NE      TJ Nichols     5.1 .94 3      13
north          NO      Val Shultz     4.5 .89 5      9
central         CT      Sheri Watson   5.7 .94 5      13
*** Currently there is 1 employee in the central region. ***
```

The `i\` (insert) command adds a line of text before the current line. In the following example, a new line is inserted before each line containing the string north.

```
$ cat script3.sed
/north/i\
*** The north region is the newest in the company. *****
```

```
$ sed -f script3.sed data.file
*** The north region is the newest in the company. *****
northwest      NW      Joel Craig      3.0 .98 3      4
western        WE      Sharon Kelly     5.3 .97 5      23
southwest      SW      Chris Foster    2.7 .8 2      18
southern        SO      May Chin       5.1 .95 4      15
southeast       SE      Derek Johnson   5.0 .70 4      17
eastern         EA      Susan Beal      4.4 .8 5      20
*** The north region is the newest in the company. *****
northeast       NE      TJ Nichols     5.1 .94 3      13
*** The north region is the newest in the company. *****
north          NO      Val Shultz     4.5 .89 5      9
central         CT      Sheri Watson   5.7 .94 5      13
```

The `c\` (change) command changes the matched line with a new line of text. In the following example, the lines containing the pattern western are changed with the change command text.

```
$ cat script4.sed
/western/c\
```

```
*** The western office employees will now report to the **\
*** manager of the central region. *****
```

```
$ sed -f script4.sed data.file
```

northwest	NW	Joel Craig	3.0	.98	3	4
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	5.0	.70	4	17
eastern	EA	Susan Beal	4.4	.8	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Shell Metacharacters for Pattern Matching for File Names

Appendix D

The following table lists the Bourne and Korn shell metacharacters that perform pattern matching for file names.

Metacharacter	Meaning
?	Matches any one character
[<i>char1 char2 ... charn</i>]	Matches any one character from the list
[! <i>char1 char2 ... charn</i>]	Matches any one character not in the list
[<i>char1</i> - <i>charn</i>]	Matches any one character from <i>char1</i> through <i>charn</i> inclusive
[! <i>char1</i> - <i>charn</i>]	Matches any one character except <i>char1</i> through <i>charn</i> inclusive
*	Matches any single character or group of characters

The following table lists the additional Korn shell metacharacters that perform pattern matching.

Metacharacter	Meaning
? (<i>pat1 pat2 ... patn</i>)	Matches zero or one of the specified patterns
@ (<i>pat1 pat2 ... patn</i>)	Matches exactly one of the specified patterns
* (<i>pat1 pat2 ... patn</i>)	Matches zero, one, or more of the specified patterns
+ (<i>pat1 pat2 ... patn</i>)	Matches one or more of the specified patterns
! (<i>pat1 pat2 ... patn</i>)	Matches any pattern except the specified patterns

Metacharacter Examples

Change the directory to /etc to use the following examples.

```
$ cd /etc
```

The following command lists all two-character file names.

```
$ ls -d ??  
dt ff fn fs lp tm
```

The following command lists all three-character file names.

```
$ ls -d ???  
ami dmi lib nca nfs rc0 rc2 rc5 rcs rpc tar  
dfs gss log net opt rc1 rc3 rc6 rmt saf
```

The following command lists all three-character file names that begin with rc.

```
$ ls -d rc?  
rc0 rc1 rc2 rc3 rc5 rc6 rcs
```

The following command lists all three-character file names that begin with rc and end in a 1, 3, or 5.

```
$ ls -d rc[135]  
rc1 rc3 rc5
```

The following command lists all three-character file names that begin with rc and end with a character other than a 1, 3, or 5.

```
$ ls -d rc[!135]  
rc0 rc2 rc6 rcs
```

The following command lists all three-character file names that begin with rc and end with a character in the range of 0 through 3 inclusive.

```
$ ls -d rc[0-3]  
rc0 rc1 rc2 rc3
```

The following command lists all three-character file names that begin with rc and end with a character other than a character in the range of 0 through 3 inclusive.

```
$ ls -d rc[!0-3]  
rc5 rc6 rcs
```

Metacharacter Examples

The following command lists file names that begin with `r`.

```
$ ls -d r*
rc0          rc2          rc5          reboot        rmt
rc0.d        rc2.d        rc6          release        rpc
rc1          rc3          rcs          remote         rpcsec
rc1.d        rc3.d        rcs.d        rmmount.conf  rpld.conf
```

Change the directory to `/usr/bin` to use the following examples.

```
$ cd /usr/bin
```

The following command lists all the file names that begin with `rm`.

```
$ ls rm*
rm           rmail        rmdir        rmic         rmid
rmiregistry
```

The following command lists all the file names that begin with `rm` and contain zero or any one of the specified patterns.

```
$ ls rm?(ic|id|ir|egistry)
rm      rmic      rmid
```

The following command lists all the file names that begin with `rm` and contain zero or more occurrences of the specified patterns.

```
$ ls rm*(ic|id|ir|egistry)
rm      rmic      rmid      rmiregistry
```

The following command lists all the file names that begin with `rm` and contain one or more occurrences of the specified patterns.

```
$ ls rm+(ic|id|ir|egistry)
rmic      rmid      rmiregistry
```

The following command lists all the file names that begin with `rm` and contain exactly one of the specified patterns.

```
$ ls rm@(ic|id|ir|egistry)
rmic      rmid
```

The following command lists all the file names that begin with `rm` and do not contain one of the specified patterns.

```
$ ls rm!(ic|id|ir|egistry)
rm      rmail      rmdir      rmiregistry
```

UNIX Commands and Utilities

Appendix E

This appendix briefly describes some frequently used UNIX commands and provides examples.

Status Commands

Status Commands

The status commands are UNIX commands, not specific shell commands.

date	Displays the current (system) date and time
ps	Displays information about system processes
who	Display which users are logged in to the system
rusers	Displays the users on local area network (LAN) systems
finger	Displays information about logged-in users
uptime	Displays how long the system has been up
rup	Displays how long LAN systems have been up (uptime for all network systems)
w	Displays the output of the uptime command and then displays a list of the current activity on the system and what each user is doing

The date Command

If you create a script that writes a report, you might want the date and time to appear as an entry in the file. The date command prints a one-line output of the current date and time (system date and time).

```
$ date  
Mon Feb 21 09:26:03 MST 2000
```

The ps Command

The ps command displays the current processes for the executing user.

```
$ ps  
 PID TTY      TIME CMD  
22373 pts/3    0:00 csh
```

The following table describes the meaning of the columns in the ps output.

Column Heading	Meaning
PID	The numerical process ID number
TTY	The terminal number from which the process was initiated
TIME	Shows how much central processing unit (CPU) time the process has consumed
CMD	The name of the command that the process is executing
UUID	The name of the user who started the process
PPID	The parent process ID
C	The processor utilization for scheduling (obsolete)
STIME	The day the process was started

As shown in the following example, some options to the ps command expand the information that is displayed. The -f option displays the full listing and includes such information as the user's login name, the process identification number (PID), the parent's PID, and so on. The -e option displays all the processes on the system. You can combine these options.

```
$ ps -ef
  UID  PID  PPID  C   STIME TTY      TIME CMD
root    0     0   0 Oct 15 ?        0:00  sched
root    1     0   0 Oct 15 ?        0:04 /etc/init -
root    2     0   0 Oct 15 ?        0:02 pageout
root    3     0   0 Oct 15 ?       8:46 fsflush
milner 22359 22306  0 Nov  1 pts/2   0:04 /usr/dt/bin/dtsession
root   619     1   0 Oct 15 ?       0:00 /usr/sbin/inetd -s
root   793     1   0 Oct 15 ?       0:00 /usr/lib/saf/sac -t 300
root   797   793   0 Oct 15 ?       0:00 /usr/lib/saf/ttymon
root   107     1   0 Oct 15 ?       0:22 /usr/sbin/rpcbind
root   109     1   0 Oct 15 ?       0:00 /usr/sbin/keyserv
root   626     1   0 Oct 15 ?       0:00 /usr/lib/nfs/lockd
daemon 624     1   0 Oct 15 ?       0:00 /usr/lib/nfs/statd
root   120     1   0 Oct 15 ?       0:01 /usr/lib/netsvc/yp/ypbind
-broadcast
root   643     1   0 Oct 15 ?       0:37 /usr/lib/autofs/automountd
-D CFS_RW=fstype=cachefs,cachedir=/cache/cachefs,bac
root   722     1   0 Oct 15 ?       0:00 /usr/lib/sendmail -bd -
q15m
```

Status Commands

```
root    647      1  0  Oct 15 ?          0:02 /usr/sbin/syslogd
root    660      1  0  Oct 15 ?          0:02 /usr/sbin/cron
root    725      1  0  Oct 15 ?          0:01 /usr/sbin/vold
root    677      1  0  Oct 15 ?          0:00 /usr/lib/lpsched
```

The who Command

In preparation for system administration tasks, it is important to know who is currently working on the system.

The **who** command displays the list of currently logged-on users, at which terminal they are logged in, and the date and time they logged in.

```
$ who
milner  console      Nov  1 08:36      (:0)
milner  pts/3        Nov  1 08:36      (:0.0)
milner  pts/4        Nov  1 08:36      (:0.0)
```

The rusers Command

The **rusers** command polls the LAN and prints a list of all currently logged-in users. If a system does not have any logged-in users, no users are displayed in the list. The **rusers** command, however, forces *all* systems to be displayed, whether they have current users or not. Options also control the order of the display (sorted by system, sorted by user, and so on). The **rusers** command sometimes takes a long time to terminate. The information is printed quickly, but then the command pauses as if it is still searching for more users. Nothing usually prints after this, so you can kill the command with a Control-C (interrupt character).

```
$ rusers
Sending broadcast for rusersd protocol version 3...
redvale      pdonohue
pinecliff     dnelson
mandan-qfe3   root jduvall
micmac-qfe3   rc80105 rc80938 rc81489 jduvall
thediver     cindyle
```

The finger Command

The finger command lists information about the users logged into the current system. The fields of information are:

- The login name of the user
- The real name of the user (as defined in the /etc/passwd file)
- The terminal to which the user is logged in
- The idle time (the amount of time since any keyboard activity was detected)
- When the user logged in
- Where the user logged in from (if the user remotely logged in from another system on the network)

```
$ finger miller
```

```
Login name: miller          In real life: Wendy Miller
Directory: /home/miller      Shell: /bin/csh
On since Nov  1 08:36:15 on console from :0
No unread mail
No Plan.
```

```
Login name: miller
Directory: /home/miller
On since Nov  1 08:36:29 on pts/3 from :0.0
2 minutes 46 seconds Idle Time
```

```
In real life: Wendy Miller
Shell: /bin/csh
```

```
Login name: miller
Directory: /home/miller
On since Nov  1 08:36:29 on pts/4 from :0.0
4 days 2 hours Idle Time
```

```
In real life: Wendy Miller
Shell: /bin/csh
```

Status Commands

The `uptime` Command

The `uptime` command displays information about the status of the system. The fields are:

- The current time
- How long the system has been up (time since the system was last booted)
- How many users are currently logged into the system
- The load average. The load average is the average number of jobs (processes) in the run queue over the last 1 minute, 5 minutes, and 15 minutes

```
$ uptime  
11:27am  up 21 day(s),  3:45,  1 user,  load average: 0.09, 0.08, 0.10
```

The `rup` Command

The `rup` command is similar to performing an `uptime` command for all systems on the LAN. The information is the same, except for these differences:

- The name of the system is listed first
- The date is not listed
- The number of users is not listed

```
$ rup  
mandan-qfe3      up  7 days,  6:47,      load average: 0.21, 0.21, 0.21  
timberland       up           4:20,      load average: 0.21, 0.26, 0.26  
kerala          up           15:53,     load average: 0.20, 0.16, 0.14  
jedi-mojo        up  2 days,  2:44,      load average: 0.00, 0.00, 0.01
```

The `w` Command

The `w` command first displays the output of the `uptime` command. Then the `w` command displays a list of the current activity on the system and what each user is doing. It gives the following information:

- The user's login name
- The terminal at which the user logged in
- The time the user logged in
- The amount of time the user has been idle

- The joint CPU (JCPU) time that all the processes of that terminal have consumed
- The process CPU (PCPU) time that the currently active processes have consumed
- The name and arguments of the current process

```
$ w
```

```
11:30am up 21 day(s), 3:48, 1 user, load average: 0.13, 0.11, 0.11
User     tty          login@  idle   JCPU   PCPU  what
milner  console      Mon 8am 4days  3:13    6 /usr/dt/bin/sdt_shell
-c unseten
milner  pts/3        Mon 8am      1       3           w
milner  pts/4        Mon 8am 4days                  -csh
```

File Access Commands

File Access Commands

The following file access commands are UNIX commands, not specific shell commands.

find	Finds the location of a file
sort	Reads a file and sort the output
head or tail	Looks at just the start or end of a file

The find Command

The `find` command allows you to search for files and directories and to execute commands on those files.

The syntax of `find` has three general argument sections. The *paths* section is a list of path names (directories) to search. The *search_criteria* section is a list of options that are considered to be a Boolean (true or false) expression. Each option (condition) is tested in turn and if the result is true, the indicated *action* is taken.

A few of the possible search criteria are:

<code>-name</code>	If the file name matches
<code>-user</code>	If the file belongs to a user (login name)
<code>-atime</code>	If the file was accessed the specified number of days ago
<code>-mtime</code>	If the file was last modified the specified number of days ago
<code>-size</code>	If the file is the specific block size

A few of the possible actions are:

<code>-print</code>	Displays the path name of the file
<code>-exec</code>	Executes the command
<code>-ok</code>	Executes the command only after receiving a <code>y</code> from <code>stdin</code>

Examples of Using the `find` Command

You do not need to know the entire name of the file to find it. For example, log files tend to grow rather rapidly. As a system administrator, you might want to watch the size of log files. To search for the administrative log files (files in the `/var/adm` directory), use the following command:

```
$ find /var/adm -name "*.log" -print  
/var/adm/log/asppp.log  
/var/adm/vold.log
```

As part of a file system clean-up, you might want to search for core files and delete them. The following two commands find any file named `core`. The first command executes the `rm` command without operator input. The second command prints the `rm` command and requires operator input.

```
$ find / -name core -exec rm -f {} \;  
$ find / -name core -ok rm -f {} \;
```

To find files that are larger than 1000 blocks, use the following command:

```
$ find / -size +1000 -print
```

The `sort` Command

A file, such as the password file, is frequently built by adding new users to the end of the file. The file is not in any particular order; however, when you are looking for something or someone in the file, it would be helpful if the file was sorted in some order. The `sort` command provides a way to sort the file.

Some of the more commonly used `sort` options are:

- r Sorts in reverse order (z to a, instead of a to z)
- n Sorts the fields numerically, instead of as just American Standard Code for Information Interchange (ASCII) strings
- t <ch> Sets the field separator character
- + num Sets the field number from which to begin sorting (fields are numbered from 0 [zero])
- num Sets the field number to stop sorting before (this is the first field that is *not* considered for sorting)
- b Ignores leading blanks (they are normally considered to be part of the fields specified during keyed sorts)

File Access Commands

- r Sorts in reverse order (z to a, instead of a to z)
- o *file* Sets the file to be used to hold the output of the sort (which might be the same file that contains the original input)

```
$ cat /etc/passwd
root:x:0:1:Super-User:/sbin/sh
daemon:x:1:1:::
smtp:x:0:0:Mail Daemon User:/:
bin:x:2:2::/usr/bin:
sys:x:3:3:::
adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
listen:x:37:4:Network Admin:/usr/net/nls:
```

To sort the file by the login name, use the following command:

```
$ sort /etc/passwd
adm:x:4:4:Admin:/var/adm:
bin:x:2:2::/usr/bin:
daemon:x:1:1:::
listen:x:37:4:Network Admin:/usr/net/nls:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
root:x:0:1:Super-User:/sbin/sh
smtp:x:0:0:Mail Daemon User:/:
sys:x:3:3:::
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
```

To sort the file by the third field (the UID), where fields are separated by the colon, use the following command:

```
$ sort -t: +2n /etc/passwd
root:x:0:1:Super-User:/sbin/sh
smtp:x:0:0:Mail Daemon User:/:
daemon:x:1:1:::
bin:x:2:2::/usr/bin:
sys:x:3:3:::
adm:x:4:4:Admin:/var/adm:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
listen:x:37:4:Network Admin:/usr/net/nls:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
```

You can put the output of a sort into a file. You can even output the sort into the original file; for example:

```
$ cat names
Big Ape 415
Roger Rabbit 408
Jessica Rabbit 510
Easter Rabbit 408
Doctor Doom 415
Easter Rabbit 408
Peter Wolf 510
Roger Rabbit 408
Peter Rabbit 510

$ sort names -o names
Big Ape 415
Doctor Doom 415
Easter Rabbit 408
Easter Rabbit 408
Jessica Rabbit 510
Peter Rabbit 510
Peter Wolf 510
Roger Rabbit 408
Roger Rabbit 408
```

If you have multiple files, already sorted, that you want to merge into a single file, use the following command:

```
$ sort file1 file2 file3 -o file4
```

Merging files might give you duplicate entries. To output only the unique lines of the file, use the following command:

```
$ sort -u names -o names
Big Ape 415
Doctor Doom 415
Easter Rabbit 408
Jessica Rabbit 510
Peter Rabbit 510
Peter Wolf 510
Roger Rabbit 408
```

Reading Part of a File

Frequently, you do not want to review an entire file. You might want to browse the beginning of the file to determine if it is the correct one, or look at the end of a file to review what was last entered in the file.

The head Command

To review the start of a file, use the `head` command:

```
$ head -5 /etc/passwd
root:x:0:1:Super-User:/sbin/sh
daemon:x:1:1:::
bin:x:2:2::/usr/bin:
sys:x:3:3:::
adm:x:4:4:Admin:/var/adm:
```

The tail Command

When reviewing log files, you are probably interested in only the last few entries of that file. You can use the `tail` command to look at the end of the file.

```
$ tail -5 /etc/passwd
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
listen:x:37:4:Network Admin:/usr/net/nls:
nobody:x:60001:60001:Nobody:/:
noaccess:x:60002:60002:No Access User:/:
nobody4:x:65534:65534:SunOS 4.x Nobody:/:
```

The tr Command

The `tr` command enables you to translate characters. It maps a set of input characters to a set of output characters. Both sets of characters should be of the same length, as the mapping is one-to-one.

```
$ cat tr.info
```

The `tr` utility copies the standard input to the standard output with substitution or deletion of selected characters. The options specified and the `string1` and `string2` operands control translations that occur while copying characters and single-character collating elements.

```
$ tr 'abc' 'xyz' < tr.info
```

The tr utility copies the standard input to the standard output with substitution or deletion of selected characters. The options specified and the string1 and string2 operands control translations that occur while copying characters and single-character collating elements.

```
$ tr '[a-z]' '[A-Z]' < tr.info
```

or

```
$ tr 'a-z' 'A-Z' < tr.info
```

or

```
$ tr a-z A-Z < tr.info
```

THE TR UTILITY COPIES THE STANDARD INPUT TO THE STANDARD OUTPUT WITH SUBSTITUTION OR DELETION OF SELECTED CHARACTERS. THE OPTIONS SPECIFIED AND THE STRING1 AND STRING2 OPERANDS CONTROL TRANSLATIONS THAT OCCUR WHILE COPYING CHARACTERS AND SINGLE-CHARACTER COLLATING ELEMENTS.

```
$ tr -d ' ' < tr.info
```

The tr utility copies the standard input to the standard output with substitution or deletion of selected characters.

The options specified and the string1 and string2 operands control translations that occur while copying characters and single-character collating elements.

The cut Command

You use the cut command to extract portions of each line of a file. The only tricky part is that what you specify in the command is the part to *keep*, not the part to discard. To help you specify the portions to keep, cut has three options: -c, -f, and -d.

-c <i>number</i>	Prints the character at <i>number</i> position
-f <i>number</i>	Prints the field numbered <i>number</i>
-d <i>character</i>	Uses <i>character</i> to separate fields (if it is a character other than a tab)
,	Lists multiple fields or characters
-	Lists a range of fields or character

File Access Commands

This following file is used in the following examples:

```
$ cat phonefile
John Robinson:Koren Inc.:978 Commonwealth Ave.:Boston:MA 01760:696-0987
Phyllis Chapman:GVE Associates:34 Seville Drive:Amesbury:MA 01881:879-0900
Jeffrey Willis:Burns Medical Group:22 Altair Drive:Yonkers:NY 10701:914-636-0000
Bill Gold:George Menkins:1100 Fan Drive:Santa Rosa:CA 95470:1-707-724-0000
Alice Gold:George Menkins:1100 Fan Drive:Santa Rosa:CA 95470: (707) 724-4568
Alice Silver:GTE Associates:123 Crescent:Artesia:NM 88210:505 746-2231
```

The following example shows printing the first character and the sixth through tenth characters:

```
$ cut -c1,6-10 phonefile
JRobin
Pis Ch
Jey Wi
BGold:
A Gold
A Silv
```

The following example shows using the colon as a field separator and printing the first and third through fifth fields:

```
$ cut -d: -f1,3-5 phonefile
John Robinson:978 Commonwealth Ave.:Boston:MA 01760
Phyllis Chapman:34 Seville Drive:Amesbury:MA 01881
Jeffrey Willis:22 Altair Drive:Yonkers:NY 10701
Bill Gold:1100 Fan Drive:Santa Rosa:CA 95470
Alice Gold:1100 Fan Drive:Santa Rosa:CA 95470
Alice Silver:123 Crescent:Artesia:NM 88210
```

The paste Command

The **paste** command appends lines to files, as is shown in the following examples:

```
$ paste data1 data2
abcd      1
efgh      2
ijkl      3
mnop      4
qrst      5
uvwxyz   6
          7
          8
          9

$ paste data1 data2 data1
abcd      1      abcd
efgh      2      efgh
ijkl      3      ijkl
mnop      4      mnop
qrst      5      qrst
uvwxyz   6      uvwx
          7
          8
          9

$ paste -d":-:" data1 data2 data1
abcd:1-abcd
efgh:2-efgh
ijkl:3-ijkl
mnop:4-mnop
qrst:5-qrst
uvwxyz:6-uvwxyz
:7-
:8-
:9-

$ paste -s -d":\n" data2
1:2
3:4
5:6
7:8
9
```

The **cmp**, **diff**, and **sdiff** Commands

The **cmp** command enables you to compares files byte-by-byte. The **diff** and **sdiff** commands compare the characters of a file.

```
$ cmp data1 data3  
data1 data3 differ: char 4, line 1
```

```
$ diff data1 data3  
1c1  
< abcd  
---  
> abc  
3c3  
< ijkl  
---  
> ijk  
6d5  
< uvwx
```

```
$ sdiff data1 data3  
abcd | abc  
efgh | efgh  
ijkl | ijk  
mnop | mnop  
qrst | qrst  
uvwxyz |
```

Combining Commands

You can take the output of one command and use that as the input to another command rather than having to put the output into a file first. There are several ways to do this.

Pipes

Pipes work when a command is expecting input from `stdin`. You run one command that outputs to `stdout` (the display) and pipe the output to another command that is expecting input from `stdin` (the keyboard). Frequently, a command accepts input from either a file or `stdin`.

For example, the `date` command outputs more data than just the date. To obtain just the date from the `date` command, you can *pipe* the output of `date` to the `cut` command and cut just the information you want.

```
$ date  
Thu Apr  4 17:00:36 PST 1996  
  
$ date | cut -d" " -f2,4,7  
Apr 4 1996
```

Do the following to output just the user name from the `who` command:

```
$ who  
milner      console      Nov  1  08:36      (:0)  
milner      pts/6        Nov  9  14:34      (:0.0)  
  
$ who | cut -d " " -f1  
milner  
milner
```

Redirection

Redirection was shown in the examples for the `tr` command in “The `tr` Command” on page E-12.

```
$ tr 'a-z' 'A-Z' < tr.info
```

The `<` character specifies that the input is coming from a file. You can also redirect a command’s output to a file.

```
$ ls > listing
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and BOS-it GmbH & Co.KG use only

Shell Comparison

Appendix F

This appendix compares the C, Bourne, and Korn shells.

Shell Comparison

Shell Comparison

The following table compares the syntax of various features in the three shells.

Feature	C	Bourne	Korn
Variables			
Shell variables	set num = 21	num=21	num=21
Environmental variables	setenv NUM 1	NUM=1 export NUM	export NUM=1
Variable manipulation	set NAME = Rob echo \$NAME echo \${NAME}ert	NAME=Rob echo \$NAME echo \${NAME}ert	NAME=Rob echo \$NAME or print \${NAME} echo \${NAME}ert or print \${NAME}ert typeset
Special shell variables			
PID of current process	\$\$	\$\$	\$\$
Exit status	\$status	\$?	\$?
Last background job		\$!	\$!
Arrays			
Assigning array values	set arr = (ksh sh csh)	N/A ^a	arr[0]=ksh; arr[1]=sh arr[2]=csh set-A arr1 ksh sh csh
Printing array elements	echo \${arr[0]} \${arr[1]}	N/A	print \${arr[0]} \${arr[1]}
Printing all elements	echo \$arr or \${arr[*]}	N/A	print \${arr[*]}
Number of elements	echo \$#arr	N/A	print \${#arr[*]}

Feature	C	Bourne	Korn
Command substitution			
Assigning to variables	set d = 'date'	d='date'	d=\$(date) or d='date'
Printing values	echo \$d echo \${d[1]} \${d[2]} echo \${#d}	echo \$d	print \$d
Command-line arguments			
Printing arguments	echo \$argv[1] \$argv[2] or echo \$1 \$2	echo \$1 \$2 ... \$9	print \$1 \$2 \${10} ...
Resetting positional parameters	N/A	set red white blue set 'date' echo \$1 \$2	set red white blue set \$(date) print \$1 \$2
Number of positional parameters	\$#argv	\$#	\$#
File name expansion			
Single character	?	?	?
Zero or more characters	*	*	*
One character from the set	[aeiou]	[aeiou]	[aeiou]]
One character from a range	[a-m]	[a-m]	[a-m]
One character not in the set	N/A	[!aeiou]	[!aeiou]

Shell Comparison

Feature	C	Bourne	Korn
Zero or one occurrence of the specified patterns	N/A	N/A	<code>?(pat1 pat2 pat3)</code>
One occurrence of patterns	N/A	N/A	<code>@(pat1 pat2 pat3)</code>
Zero, one, or more occurrences of the patterns	N/A	N/A	<code>*(pat1 pat2 pat3)</code>
One or more occurrences of the patterns	N/A	N/A	<code>+(pat1 pat2 pat3)</code>
Any pattern except the specified patterns	N/A	N/A	<code>!(pat1 pat2 pat3)</code>

I/O redirection and pipes

Input redirection	<code>cmd < filey</code>	<code>cmd < filey</code> <code>cmd 0< filey</code>	<code>cmd < filey</code> <code>cmd 0< filey</code>
Output redirection	<code>cmd > filex</code>	<code>cmd > filex</code> <code>cmd 1> filex</code>	<code>cmd > filex</code> <code>cmd 1> filex</code>
Error redirection	<code>(cmd > out) >& errors</code>	<code>cmd 2> errors</code>	<code>cmd 2> errors</code>
Append to a file	<code>cmd >> filex</code>	<code>cmd >> filex</code>	<code>cmd >> filex</code>

Feature	C	Bourne	Korn
Output and errors redirected to the same file	<code>cmd >& both</code>	<code>cmd > both 2>&1</code>	<code>cmd > both 2>&1</code>
Assign output and ignore noclobber	<code>cmd >! filex</code>	N/A	<code>cmd >! filex</code>
Output and errors to <code>cmd</code>	<code>cmd1 & cmd2</code>	<code>cmd1 2>&1 cmd2</code>	<code>cmd1 2>&1 cmd2</code>
“here” document	<code>cmd << EOF</code> <code>input_statements</code> EOF	<code>cmd << EOF</code> <code>input_statements</code> EOF	<code>cmd << EOF</code> <code>input_statements</code> EOF
Pipe the output	<code>cmd1 cmd2</code>	<code>cmd1 cmd2</code>	<code>cmd1 cmd2</code>
Conditional statement	<code>cmd && cmd</code> <code>cmd cmd</code>	<code>cmd && cmd</code> <code>cmd cmd</code>	<code>cmd && cmd</code> <code>cmd cmd</code>
Read from standard input	<code>set var = \$<</code>	<code>read var1</code> <code>read var1 var2</code>	<code>read var1</code> <code>read var1 var2</code> <code>read</code> <code>read var?"Enter value"</code>

Arithmetic

Perform calculation	<code>@ var =\$x + \$y</code>	<code>var='expr 5 + 1'</code>	<code>((var = 5 + 1))</code>
			<code>let var=5 + 1</code>

Relational condition test	<code>if (\$x < \$y)</code>	<code>if [\$x -le \$y]</code>	<code>((x < y))</code>
			<code>let var=x+y</code>

Boolean conditional test	<code>cmd && cmd</code> <code>cmd cmd</code>	<code>cmd && cmd</code> <code>cmd cmd</code>	<code>cmd && cmd</code> <code>cmd cmd</code>

Tilde expansion

Home directory of any user	<code>~username</code>	N/A	<code>~username</code>

Shell Comparison

Feature	C	Bourne	Korn
Home directory of current user	~	N/A	~
Current working directory	N/A	N/A	~+
Previous directory	N/A	N/A	~-
Aliases			
Create an alias	alias name value	N/A	alias name=value
List aliases	alias	N/A	alias
Remove an alias	unalias name	N/A	unalias name
History			
Set the history	set history = 25	N/A	automatic or HISTSIZE=25
Display a numbered history list	history	N/A	history fc -1
Display a portion of the list	history 5	N/A	history 5 10 history -5
Re-execute a command	!! (<i>last_command</i>) !5 (<i>fifth_command</i>) !v (<i>last_command starting with v</i>)	N/A	r (<i>last_command</i>) r 5 (<i>fifth_command</i>) r v (<i>last_command starting with v</i>)
Set the interactive editor	N/A	N/A	set -o vi set -o emacs
Signals			
Command	onintr	trap	trap

Feature	C	Bourne	Korn
Initialization files			
Executed at login	.login	.profile	.profile
Executed every time the shell is invoked	.cshrc	N/A	file specified in .profile with ENV variable ENV=\$HOME/.kshrc
Functions			
Define a function	N/A	func1() { commands; }	function func1 { commands; } func1() { commands; }
Use a function	N/A	func1 func1 param1 param2	func1 func1 param1 param2
Programming constructs			
if conditional	if (expression) then commands endif if { (command) } then commands endif	if [expression] then commands fi if command then commands fi	if [[stringexpression]] then commands fi if ((numericexpression)) then commands fi
if/else conditional	if (expression) then commands else commands endif	if command then commands else commands fi	if command then commands else commands fi

Feature	C	Bourne	Korn
if/else/elseif conditional	<pre>if (expression) then commands else if (expression) commands else commands endif</pre>	<pre>if command then commands elif command then commands else commands fi</pre>	<pre>if command then commands elif command then commands else commands fi</pre>

Feature	C	Bourne	Korn
Programming constructs (continued)			
goto	goto <i>label</i> <i>commands</i> <i>label</i> :	N/A	N/A
switch and case	switch (<i>value</i>) case <i>pattern1</i> : <i>commands</i> breaksw case <i>pattern2</i> : <i>commands</i> breaksw default: <i>commands</i> breaksw endsw	case <i>value</i> in <i>pattern1</i>) <i>commands</i> ;; <i>pattern2</i>) <i>commands</i> ;; *) <i>commands</i> ;; esac	case <i>value</i> in <i>pattern1</i>) <i>commands</i> ;; <i>pattern2</i>) <i>commands</i> ;; *) <i>commands</i> ;; esac
Loops			
while loop	while (<i>expression</i>) <i>commands</i> end	while <i>command</i> do <i>command</i> done	while <i>command</i> do <i>commands</i> done
for/foreach loop	foreach var (<i>wordlist</i>) <i>commands</i> end	for var in <i>wordlist</i> do <i>commands</i> done	for var in <i>wordlist</i> do <i>commands</i> done
until loop	N/A	until <i>command</i> do <i>commands</i> done	until <i>command</i> do <i>commands</i> done
repeat	repeat 3 <i>command</i>	N/A	N/A

Shell Comparison

Feature	C	Bourne	Korn
select	N/A	N/A	<pre>PS3="Select a menu item" select var in wordlist do commands done</pre>

a. "N/A" means "not applicable"