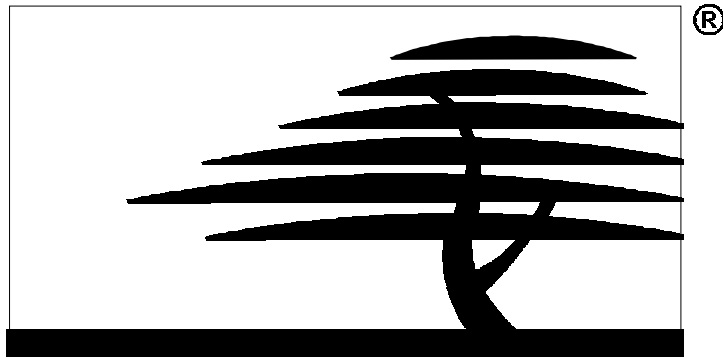




SCO

Shell Programming for System Administrators

Instructor Guide

Course Release: 3.0.2

For Software: UnixWare® 7 and
SCO OpenServer™ Release 5

© 1987–1999 The Santa Cruz Operation, Inc. All rights reserved.

This publication is protected under copyright laws and international treaties.

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

SCO, The Santa Cruz Operation, the SCO logos, VP/ix, SCO OpenServer, UnixWare, dbXtra, Panner, SCO Global Access, SCO MPX, SCO Doctor, SCO Doctor for Networks, SCO Doctor Lite, SCO Premier Motif, SCO Visual Tcl, SCO VisionFS, SCO ToolWare, SCO TermLite, SCO Vision 97, SCO CIFS Bridge, SCO WebTop, The Internet Way of Computing, and IWoC are trademarks or registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries. IXI, X.desktop, Deskworks, Wintif, IXI Panorama, Deskterm, and IXI Desktop are trademarks or registered trademarks of IXI Limited, a subsidiary of The Santa Cruz Operation, Inc. Visionware, Devkit.Vision, Esprit, Kodon, Super.Vision, Visionware Super.Vision, Term.Vision, Visionware Direction, the Visionware logo, Visionware SQL-Retriever, and XVision are registered trademarks of Visionware Limited, a subsidiary of The Santa Cruz Operation, Inc. SuperVision, TermVision, X-Visionware, SQL-Retriever, and Vision Builder are trademarks of Visionware Limited. UNIX is a registered trademark of The Open Group in the United States and other countries. Cheyenne and ARCServe are registered trademarks of Cheyenne Software, Inc. Netscape, Netscape Navigator, Netscape Communications Server, Netscape Commerce Server, Netscape Proxy Server, Netscape FastTrack Server, Netscape Enterprise Server, Netscape SuiteSpot, Netscape Catalog Server, Netscape News Server, Netscape Mail Server, and Netscape Navigator Gold are trademarks of Netscape Communications Corporation. NFS was developed by Legent Corporation (formerly Lachman Associates, Inc.) based on LACHMAN SYSTEM V NFS. LACHMAN is a trademark of Legent Corporation. NFS is a trademark of Sun Microsystems, Inc. TCP/IP was developed by Legent Corporation (formerly Lachman Associates, Inc.) based on LACHMAN SYSTEM V STREAMS TCP, a joint development of Lachman Associates and Convergent Technologies. MPX was developed by Corollary, Inc. VP/ix is a product developed and licensed by Phoenix Technologies, Ltd/INTERACTIVE Systems Corporation. XRemote is a registered trademark of Network Computing Devices, Inc. Oracle is a registered trademark of Oracle Corporation, Redwood City, California. Java is a trademark of Sun Microsystems, Inc. All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

The SCO software that accompanies this publication is commercial computer software and, together with any related documentation, is subject to the restrictions on US Government use as set forth below. If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the US Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If this procurement is for a civilian government agency, this FAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: This computer software is submitted with restricted rights under Government Contract No. _____ (and Subcontract No. _____, if appropriate). It may not be used, reproduced, or disclosed by the Government except as provided in paragraph (g)(3)(i) of FAR Clause 52.227-14 alt III or as otherwise expressly stated in the contract. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

The copyrighted software that accompanies this publication is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software.



The SCO® Authorized Education class you are about to take may form part of the recommended preparation for those seeking the SCO ACE™ certification.

The SCO ACE certification program uses a series of exams to measure your ability to support and administer SCO installations with the SCO recommended skills that maximize system performance. As a certified SCO ACE graduate, you earn all of the respect that the industry accords to each certified member of the extended SCO support team. When professionals in the computer industry see your SCO ACE certification, they know that you have passed demanding SCO testing requirements. This certification is the foundation for mastering real-world UNIX® systems, including those from other vendors.

Important!

When registering for any SCO ACE exam at a Sylvan Prometric™ testing center, please quote the course release number on the front of the ***Student Guide*** that the exam covers. This number corresponds to the version number of the SCO ACE exam for that course.

Ask your SCO Authorized Instructor, or call SCO for more information about SCO Certification.

Table of Contents

Lesson 1: Course Introduction	1-1
Introduction.....	1-3
Course goal and purpose.....	1-3
Target audience	1-4
Student Materials	1-7
The <i>Student Guide</i>	1-7
Job aids	1-7
Typographical Conventions	1-9
Lesson 2: Shell Overview	2-1
UNIX Architecture	2-7
The kernel	2-9
The shell.....	2-10
Shell Features.....	2-11
Command-line interpreter.....	2-11
Filename Expansion.....	2-15
I/O Redirection	2-19
Output redirection	2-21
Input redirection.....	2-22
Pipelines.....	2-25
Environment control	2-26
Programming language	2-26
Lesson Summary.....	2-28
Review Questions	2-29
Relevant Commands, Files, and Syntax	2-30
Answers to Review Questions	2-32
Lesson 3: Command Execution	3-1
Command Types	3-6
Binary programs	3-6
Shell built-in commands	3-7
Shell scripts.....	3-7
Processes	3-9
Creating processes	3-11
Finding commands.....	3-15
Executing commands.....	3-19
Execution Techniques	3-20
Background execution	3-20
Command grouping	3-20
The dot (.) command.....	3-24
Lesson Summary.....	3-25

Relevant Commands and Files	3-28
Answers to Review Questions	3-29
Lesson 4: Basic Scripts	4-1
Overview	4-6
Shell Script Execution	4-9
Explicit execution	4-9
Implicit execution	4-9
Explicit vs. implicit execution	4-13
Comments	4-15
Which Shell Executes a Script?	4-16
Traditional shell determination	4-17
Hashpling	4-17
The echo Command	4-19
echo escape sequences	4-21
Error messages	4-21
Debugging Techniques	4-23
-v option	4-23
-x option	4-23
The set command	4-23
Lesson Summary	4-25
Lab: Working with Batch Files	4-27
Your login environment	4-27
System status	4-28
Relevant Commands	4-29
Answers to Review Questions	4-30
Lesson 5: Shell Variables	5-1
User Variables	5-6
Variable identifiers	5-6
Setting variables	5-7
Unsetting variables	5-8
Accessing variables	5-8
\${ name } notation	5-9
Exporting variables	5-13
Readonly variables	5-14
Displaying variables	5-14
Shell Environment Variables	5-17
The read Command	5-19
Syntax	5-19
Usage	5-21
Positional Parameters	5-22
Syntax	5-25
The shift command	5-27
The set command	5-31

Special Shell Variables.....	5-35
Lesson Summary.....	5-37
Lab: Using Shell Variables	5-39
Login environment.....	5-39
Phone database.....	5-39
A friendly find	5-40
Relevant Commands	5-41
Answers to Review Questions	5-42
Lesson 6: Quoting	6-1
Shell Metacharacters	6-7
Quoting Mechanisms	6-9
Single quotes	6-9
Double quotes	6-11
Backslash	6-13
Command Substitution	6-17
Syntax	6-17
The line command	6-17
Command-Line Parsing Sequence.....	6-19
Examples.....	6-23
The eval Command.....	6-27
eval syntax	6-27
eval Examples.....	6-29
Lesson Summary.....	6-32
Lab: Quoting	6-34
Command-line prompt.....	6-34
Phone database: revisited.....	6-34
Shell escape.....	6-34
Recursive grep	6-34
Relevant Commands	6-35
Answers to Review Questions	6-36
Lesson 7: Decision Constructs	7-1
Exit Codes	7-6
Communication mechanism	7-6
Conventional exit codes.....	7-6
The exit command	7-7
Specialized commands with constant exit codes	7-8
The test Command.....	7-9
File status	7-11
Character strings	7-13
Integer expressions	7-15
Logical operators	7-15
The if Statement.....	7-19
Syntax	7-19

Execution	7-23
The else statement.....	7-23
The elif statement.....	7-25
Nested if statements	7-27
The case Statement	7-31
Syntax	7-31
Execution	7-32
Special Constructs.....	7-35
The && construct	7-35
The construct.....	7-35
Combining the && and constructs	7-39
Lesson Summary.....	7-40
Lab: Decision Constructs.....	7-42
A friendly find: revisited.....	7-42
Phone database: revisited.....	7-42
Relevant Commands and Files	7-44
Answers to Review Questions	7-45

Lesson 8: Looping Constructs

8-1

The expr Command.....	8-7
General syntax	8-7
Arithmetic operators	8-7
Operator precedence	8-11
Variable arithmetic	8-11
The while Statement	8-15
Syntax	8-15
Execution	8-19
The until Statement	8-23
Syntax	8-23
Execution	8-23
The for Statement	8-27
Syntax	8-27
Overview	8-27
Execution	8-28
The \$@ variable: revisited.....	8-31
Loop Commands	8-35
The break statement	8-35
The continue statement	8-39
Lesson Summary.....	8-41
Lab: Looping Constructs	8-43
Phone database: revisited.....	8-43
File tally	8-43
Relevant Commands	8-45
Answers to Review Questions	8-46

Lesson 9: Shell Functions	9-1
----------------------------------	------------

Shell Functions	9-7
Overview	9-7
Syntax	9-7
Viewing and removing functions	9-9
Exporting functions	9-9
Shell functions and the dot (.) command	9-10
Passing Arguments	9-13
The return Statement	9-15
Overview	9-15
Syntax	9-15
Lesson Summary	9-17
Lab: Shell Functions	9-19
Everyday functions	9-19
Phone database: revisited	9-19
File tally: revisited	9-19
Relevant Commands	9-20
Answers to Review Questions	9-21

Lesson 10: Shell Signal Handling	10-1
---	-------------

Asynchronous Events	10-6
Overview	10-6
Signals	10-6
Generating Signals	10-9
Keyboard signals	10-9
The kill command	10-9
The trap Command	10-11
Syntax	10-11
Catching signals	10-13
Ignoring signals	10-13
Resetting signals	10-13
Lesson Summary	10-14
Lab: Shell Signal Handling	10-16
Logoff cleanup	10-16
Phone database: revisited	10-16
Relevant Commands and Files	10-17
Answers to Review Questions	10-18

Lesson 11: Efficiency Considerations	11-1
---	-------------

The time Command	11-7
Syntax	11-7
Overview	11-7
Redefining the PATH Variable	11-9
Absolute vs. Relative Pathnames	11-11

Built-in Shell Commands.....	11-15
Pipeline Efficiency	11-19
Data reduction	11-19
Unnecessary pipelines.....	11-20
Lesson Summary.....	11-22
Relevant Commands	11-24
Answers to Review Questions	11-25

Appendix A: Answers to Lab Exercises	A-1
---	------------

Lesson 1: Course Introduction	A-4
Lesson 2: Shell Overview	A-5
Lesson 3: Command Execution	A-6
Lesson 4: Basic Scripts	A-7
Lesson 5: Shell Variables	A-9
Lesson 6: Quoting	A-13
Lesson 7: Decision Constructs.....	A-16
Lesson 8: Loop Commands	A-19
Lesson 9: Shell Functions	A-23
Lesson 10: Shell Signal Handling.....	A-27
Lesson 11: Efficiency Considerations.....	A-30

Appendix B: Programming Tools	B-1
--------------------------------------	------------

Extracting Data from Files.....	B-7
The head Command	B-7
The tail Command	B-7
The cut Command	B-9
The uniq command.....	B-10
The grep command.....	B-12
Combining Files.....	B-19
The cat command	B-19
The paste command.....	B-20
Sorting File Contents	B-23
Syntax	B-23
Examples.....	B-24
Editing Files with sed	B-26
Syntax	B-26
Using sed scripts.....	B-29
Using sed Functions	B-31
awk Basics	B-32
Syntax	B-32
Field variables.....	B-33
Built-in Variables	B-33
User-defined variables	B-34
Operators.....	B-34
Regular expressions	B-35

Formatting awk output	B-36
Simple awk programs	B-37
Appendix Summary	B-38
Relevant Commands	B-39

1

Course Introduction

Instructor Guide

Instructor Guide

Lesson 1

Course Introduction

1.1 Introduction

This ***Shell Programming for System Administrators*** course is designed for technically-oriented individuals who need to enhance the functionality of their SCO® UNIX® operating system. This course is invaluable for any user of the system who wants to use the full power of the shell to execute commands on the system. The target audience includes support technicians, developers, engineers, system administrators, and other technical users who require in-depth knowledge on the use and programming aspects of the Bourne shell.

1.1.1 Course goal and purpose

At the conclusion of this course, students will be able to use the Bourne shell to accomplish a variety of tasks on the system, either by writing a shell script or by entering commands interactively. By the end of the course, the student will:

- Identify and use the basic command-line features of the Bourne shell
- Identify and execute the different command types on the system
- Write a basic “batch-file” shell script
- Use shell variables in a shell script
- Use quotes to protect shell metacharacters in a shell script
- Conditionally execute lines of code in a shell script
- Use looping statements to repeatedly execute lines of code in a shell script
- Write shell functions for use within a shell script
- Handle signals when they are received in shell scripts
- Identify efficiency considerations for a shell script

1.1.2 Target audience

This class is aimed at developers, engineers, support technicians, system administrators, and other technical users who operate in the SCO UNIX environment on a regular basis. Students are likely to have a wide variety of backgrounds in XENIX® or UNIX, and must have prerequisite skills equal to or exceeding those provided in the ***Introduction to SCO UNIX Systems*** course. Completion of ***SCO System Administration*** courses is not required, but provide useful background information for the student.

Course schedule

This course takes three days to complete. A suggested lesson duration is specified at the beginning of each lesson in this ***Instructor Guide***.

The labs included in the course must be completed in order to meet course objectives.

Suggestions for practice exercises are also included throughout the course. Ample time has been allotted to each lesson to allow for the completion of these exercises.

Equipment

The following equipment should be installed in a classroom to support 14 students:

- 14 (7 if shared) industry standard Intel Pentium or Intel Pentium Pro microprocessors containing:
 - A 540MB (minimum) primary disk
 - A minimum of 32MB of RAM in each system
 - A minimum of two unused serial ports for each system
 - A serial, bus, or PS/2-compatible mouse
- One liquid crystal display (LCD) unit, VGA projector, or large-screen display monitor
- White board with markers
- Overhead projector

Training materials

These training materials are used in this course:

- **Instructor Guide** and transparency masters
- **Student Guide**
- Course disks containing course graphics (transparencies) in GIF file format

In addition, the instructor should have these printed materials in the classroom:

- Kochan, Stephen G. and Patrick H. Wood. **UNIX Shell Programming**, Revised Edition. Carmel, Indiana: Hayden Books, 1990.
- vi Quick Reference Card
- **Introduction to SCO Visual Tcl Programming**, Student handout
- **Introduction to SCO Visual Tcl Programming**, Instructor Guide and transparency masters

Software

SCO OpenServer Release 5 or UnixWare 7.

The *Instructor Guide*

The **Instructor Guide (IG)** is organized as follows:

- Instructor notes are intermingled with text in the **Student Guide (SG)** to allow the instructor to easily refer to section content.
- Page numbers in the **IG** do not correspond to those in the **SG**; however, the heading numbers do.
- Cross references to other sections within the course materials are included within the text. These cross references generally include the section number or AV number, the section name, and page number. The page number will almost always be different in the **SG** than the **IG**; however, the other components will be the same.

The **IG** also contains notes which are instructional aids, including:

AV 1-1: Slide Title

These notes explain the instructional objective of every transparency. The AV number cross references to the number that appears above every slide in the workbook.

GuruNotes

This is extra information not directly presented in the **SG** and is intended for more advanced students.

GIF files

You may opt to present graphics to the students with GIF files rather than by projecting slides made with the transparency masters. Benefits of the GIF files include:

- They can be projected in full color rather than greyscale, as they appear in the workbook.
- There is no added cost of creating acetate transparencies.
- The GIF files do not deteriorate over time as acetate slides can.

One or more disks containing Graphics Interchange Format (GIF) files are included with the course materials instructors use. The disks are UNIX **tar** format disks and contain files that use this naming convention:

nameLLNN.gif

where:

- *name* is the course name abbreviation
- *LL* is the lesson number
- *NN* is the graphic number
- The *.gif* filename extension indicates this is a GIF file.

A footer at the bottom left of every graphic shows the GIF filename without its *.gif* filename extension.

You can then use a variety of UNIX or Microsoft Windows utilities to view the GIF files, including, but not limited to:

- **xv**—a UNIX-based shareware graphics viewer, available on SCO® Skunkware™
- Microsoft **Powerpoint**—a Windows-based presentation product

1.2 Student Materials

The contents of the box containing your student materials are yours to keep and use for taking notes. Your instructor may have other reference materials in the classroom for your use while attending this class. Please do not take or write in these materials as they will be needed again for later classes.

1.2.1 The *Student Guide*

The *Student Guide* contains:

- Outline of presentation
- Note-taking space
- Copies of all transparencies

1.2.2 Job aids

Job aids are designed to assist students in performing on-the-job administration and as references to other helpful materials. The job aids included with this course include:

vi Quick Reference Guide

This card provides a list of the most commonly used **vi** commands and their functions

Lesson plan

Each lesson follows a similar approach, or methodology, in the delivery of instruction:

- The suggested lesson duration and instructor preparation notes are provided at the beginning of each lesson when applicable.
- The lesson begins with an overview transparency of the lesson objectives.
- The lesson continues with the presentation of lesson topics and the demonstration of lesson tasks.
- Exercises, which allow the students to practice discrete tasks, appear throughout the presentation.
- References appear where appropriate.
- The lesson content concludes with a summary. Use this to reinforce the topics and tasks covered. You should also refer back to the objectives that were presented at the beginning of the lesson.

- Following the content of most lessons are summary exercises, which consist of review questions and labs. The review questions allow the instructor to determine how well the students understood the content, and the labs allow students to apply the material that they learned.

1.3 Typographical Conventions

This publication presents commands, filenames, keystrokes, and other special elements as shown here:

lp	Commands, device drivers, programs, and utilities (names, icons, or windows); the letter in parentheses indicates the reference manual section in which the command, driver, program, or utility is documented
<i>/new/client.list</i>	Files, directories, desktops (names, icons, or windows), system, network, and printer names
root	User names
<i>arg</i> or <i>arg</i>	Placeholders (replace with appropriate name or value)
[<i>arg</i>]	Surrounds an <i>option</i> or <i>cmdarg</i> that is not mandatory
<i>arg ...</i>	Indicates multiple occurrences of the <i>option</i> or <i>cmdarg</i>
\$	Command-line prompt for commands executable from the Bourne shell by any user
#	Command-line prompt for commands only executable from the root account
	Keyboard keys
 	Press  and  keys simultaneously
  	Press all three keys,  ,  , and  , simultaneously
 d	Press  key. With  still depressed, also press  .
 0240	Press  key. With  still depressed, also press and release in sequence  ,  ,  , and  .
 Q + i	Press  key. With  still depressed, also press  . Release both keys and press  .
 Q + I	Press  key. With  still depressed, also press  . Release both keys and press   .
Exit program?	System output (prompts, messages)

tar tv	User input, such as on a command line
“Description”	Field names or column headings (on screen or in database)
Cancel	Button names
Edit	Menu names
Copy	Menu items
File ⇌ Find ⇌ Text	Sequences of menus and menu items
open or open(S)	Library routines, system calls, kernel functions, C keywords; the letter in parentheses indicates the reference manual section in which the file is documented
\$HOME	Environment or shell variables
SIGHUP	Named constants, signals, and kernel parameters
“adm3a”	Data values
<i>employees</i>	Database names and tables
buf	C program structures
b_b.errno	Structure members

Using the root account in the classroom

When teaching *Shell Programming for System Administrators*, the instructor might refer to the use of the **root** account. However, allowing students to use the **root** account risks damaging the system. The power of the **root** account in a UNIX system makes it both important and dangerous in the classroom.

When, then, should students be allowed to use **root**? There is no hard and fast rule—only a number of helpful practices used by instructors. Here are the results of their experience, which you may find helpful in deciding what approach you will take.

NOTE: None of the exercises in this course require the use of the **root** account.

Why should students use root?

- They will be system administrators, and should learn to use **root** in the classroom before they use it on the job.

- They paid for the course, and often feel **root** privileges are part of what they paid for.
- Many procedures central to system administration are best learned with a hands-on approach.
- Many system administration students have already been system administrators, and expect to use **root**.

Why shouldn't students use root?

- Pranks and tampering can affect the classroom experience.
- Students should be discouraged from using **root** except when absolutely necessary.
- Reinstalling a corrupted system wastes class time.
- Some instructors believe that **root** procedures and use can be taught just as well using the LCD display.

What to say to students

- Explain the above concerns.
- Explain your approach, along with its pros and cons.
- Make clear to students their responsibilities if they use the **root** account.

Classroom setup

Install the necessary software, including the operating system and the class software on all the machines. The course disk is a **tar** disk that contains solutions to all lab exercises; these answers are also in Appendix A: "Answers to Exercises and Review Questions."

- Install the course disk software by following these steps:

- Step 1. Login as **root**.
- Step 2. Insert the diskette into the disk drive.
- Step 3. Enter:

```
# cd /usr
# tar xvf /dev/fd0
```

Your disk drive may have a different filename, in which case you should specify the appropriate name in place of `/dev/fd0`.

The above command will place the files in the `/usr/shellprg` directory (using a relative pathname); the `shellprg` subdirectory is created by `tar` if necessary. Students can copy these files into their own directories if needed. If you need to place the files elsewhere, `cd` to a different directory than `/usr`.

- Create user accounts for the students if you choose to not have them write and test shell programs as **root**. (This is a wise precaution!).
- Configure classroom equipment, your own display, and the LCD.

Lesson preparation

Before class:

- Read the pre-lesson preparation prior to each lesson. Complete any tasks listed in the preparation.
- Read the lesson thoroughly before teaching. Add any additional notes or exercises.
- Read through the appropriate sections of the SCO documentation in support of the lesson topics.
- Review the system files in each lesson to which students need access, and change the file permissions to make the files readable if necessary

2

Shell Overview

Instructor Guide

Instructor Guide

Lesson 2

Shell Overview

Suggested lesson duration

This lesson should take approximately 2 hours to complete. This lesson has no lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 2-1

Instructor Guide

Shell Overview

- Differentiate kernel and shell features
- Use wildcard metacharacters for filename expansion
- Use shell I/O redirection mechanisms
- Use shell pipes
- Identify shell environment variables

sp0201 © The Santa Cruz Operation, Inc.

Introduction

The UNIX operating system is a sophisticated multiuser, multitasking system. To work with it, users typically use an interface known as a shell or command-line interpreter. Several such shells are available with SCO UNIX, including the Bourne shell (*/bin/sh*), C shell (*/bin/csh*), and Korn shell (*/bin/ksh*), to name three.

This course focuses on using the Bourne shell and automating tasks using the programming language that it provides. You are probably already familiar with many of the command-line interface features of the shell, such as the use of filename wildcards and I/O redirection. This lesson will review these basic shell capabilities before proceeding onto the more sophisticated features presented later in this course.

Objectives

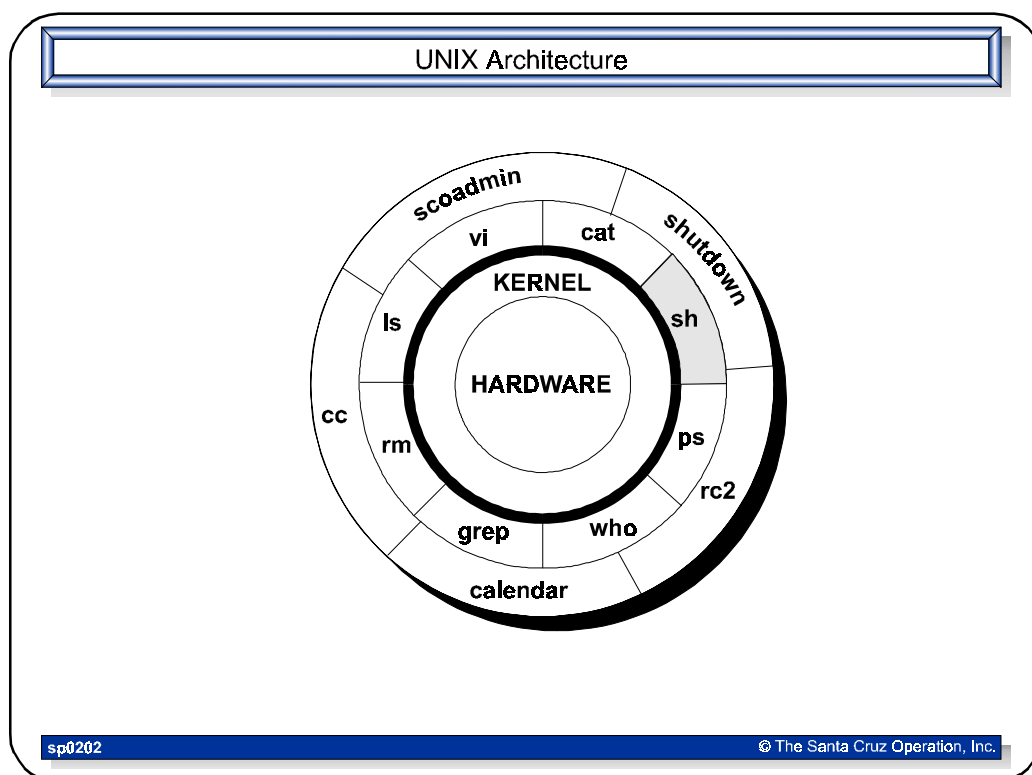
This lesson reviews the basic command-line interface features of the UNIX Bourne shell. By the end of this lesson, you will be able to:

- Differentiate kernel features from shell features
- Define the term *command-line interpreter*
- Use wildcard metacharacters for filename expansion on a command line
- Use shell I/O redirection mechanisms
- Use shell pipes in a command line
- Identify items in the working environment that the shell controls
- Define the term *interpreted programming language*

AV 2-1: Shell Overview

This first slide describes the objectives covered in this lesson.

AV 2-2



2.1 UNIX Architecture

The purpose of this section is review the major components of the UNIX operating system and lead into more detailed coverage of role of the shell.

The UNIX operating system, like all other operating systems, is a software package that provides users with a flexible means of operating and controlling hardware and software resources of a computer. On UNIX systems, this large package of software is composed of smaller software components that interact through well-defined interfaces.

UNIX components are categorized as:

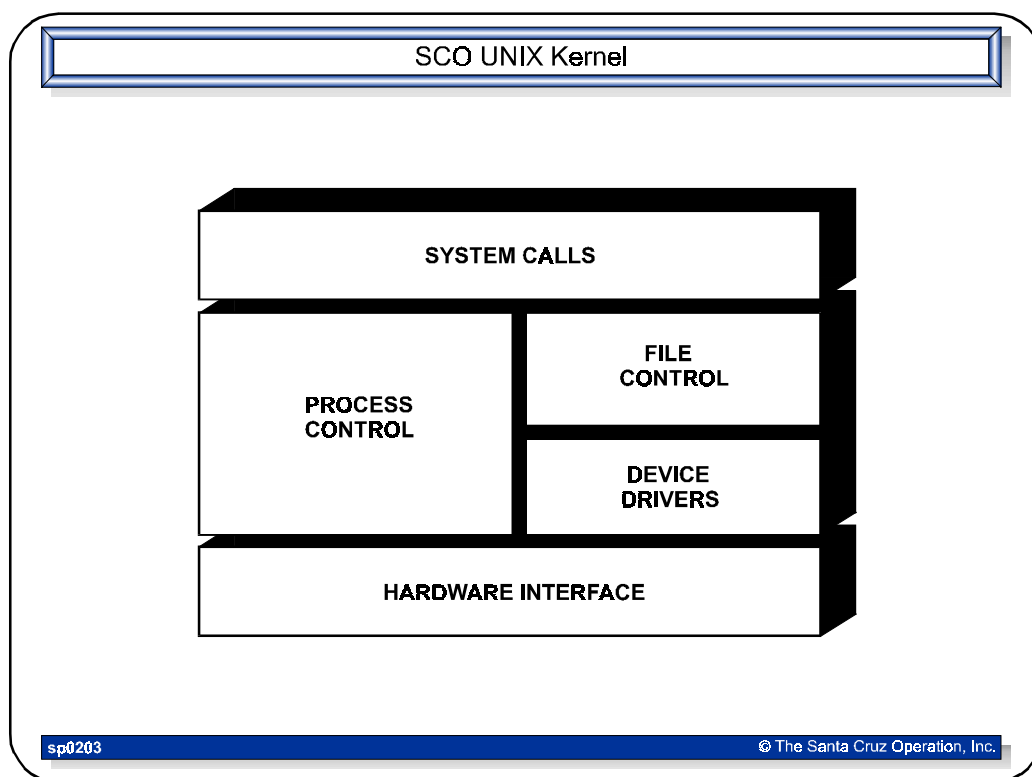
- System software, such as the kernel, which provides interfaces between user applications and hardware
- Application software, such as the Bourne shell, which provide interfaces that allow users to access system services

AV 2-2: UNIX Architecture

The slide is one representation of the UNIX operating system architecture. The key thing to emphasize is where the shell fits into the big picture. Stress that it is just another application on the system, intrinsically no different than **vi** or **grep**—the shell is also just a big C program.

The heavier line surrounding the kernel represents the system call interface. Think of it as a barrier which prevents user processes from directly accessing the kernel. Access is via a system call, which is a request for a specific service from the kernel.

AV 2-3



Instructor Guide

2.1.1 The kernel

AV 2-3: SCO UNIX Kernel

This slide shows the major components of the kernel and how they are accessed through system calls.

The kernel represents the core of the operating system and is responsible for the computer's basic behavior. Its functions include:

System call interface	Receives requests through a well-defined set of system calls. These calls are made by users from a shell or some higher-level application.
Process control	Controls the SCO UNIX multitasking environment. It creates the processes in which commands execute. These processes are allowed to run for a small amount of time (called a time slice), and then preempted, giving another process a chance to use the CPU. Eventually, each process gets enough time slices to finish its job.
Memory management	Allocates and frees up memory so that processes have the memory they require to execute
Filesystem management	Manages the storage environment by maintaining the organization, integrity, and security of user data
I/O management	Controls and communicates directly with the underlying computer hardware and makes important resources available for user processes

The kernel is loaded at boot time, typically from the file *unix*, providing services for processes when they make system call requests and when asynchronous events (such as external interrupts) occur.

The kernel does not directly interact with users, but instead interacts with user applications (via system calls).

Explain that the UNIX kernel is actually a program named *unix*. (Note: the kernel resides on the *root* filesystem in SCO UNIX Release 3.2v4 and before, and on a separate *stand* filesystem in SCO OpenServer Release 5 and UnixWare 7. The reason for this change was the introduction of a new filesystem type for the *root* filesystem which is not bootable.)

2.1.2 The shell

A shell is any interactive computer program that provides a user interface to the operating system kernel. The shell assigned to your account becomes the active program when you log into the system and performs these functions:

- Provides a user interface through which users interact with the system
- Prompts users for commands to be executed and then executes them on behalf of the user
- Provides a configurable environment allowing users to modify commands, prompts, and other system behavior

Shells are often referred to as command-line interfaces. Those supported in UNIX systems include:

Bourne shell (sh)	The standard shell found on UNIX systems
Korn shell (ksh)	An extended version of the Bourne shell
C shell (csh)	The shell popularized by versions of UNIX developed at Berkeley (BSD UNIX)
Restricted shell (rsh, rksh)	Subsets of the Bourne and Korn shells designed for accounts with limited access to the system

Point out that the shell being used in this course is the Bourne Shell. The remainder of this lesson should act primarily as review material. Students should already know the basic mechanisms, but may require more instruction on some of the finer points.

2.2 Shell Features

The remainder of this lesson reviews the features provided by the Bourne shell:

Command-line interpreter	Responds to typed commands
Filename expansion	Interprets wildcard characters
I/O redirection	Allows specification of files or devices used for command input and output; constructs pipelines to stream data between programs
Environment control	Allows customization of your environment
Programming language	Processes shell scripts

2.2.1 Command-line interpreter

Note that *shell*, *command-line interface*, and *command-line interpreter* are often used synonymously. In this context, command-line interpreter is one (of the many) functions of the shell.

A command-line interpreter is a service provided by the shell to allow the user to interact with the operating system. It serves to:

- Prompt for, and read, command lines entered by users
- Read from command files which are called shell scripts
- Parse the command line into individual arguments
- Execute commands on behalf of the user

Introduce students to the concept of the command-line parsing sequence here. Mention there are many parsing steps. The first step is to read the command line; the last is to finally execute it. There are many things which occur between these two steps. Prompt students for examples of what these could be.

The shell performs a sequence of steps *before* the command is actually executed. Some of these steps include:

- Read the command line
- Perform variable substitution
- Expand filenames with wildcards
- Redirect input and output (I/O)

The complete command-line parsing sequence is covered in Section 6.4 on page 6-19.

AV 2-4

Shell Wildcard Metacharacters

*	Matches zero or more of any character
?	Matches exactly one of any character
[<i>string</i>]	Matches exactly one of any character in <i>string</i>
[<i>a-z</i>]	Matches exactly one of any character in the range <i>a-z</i>
[!<i>string</i>]	Matches exactly one of any character <i>not</i> in <i>string</i>

sp0204

© The Santa Cruz Operation, Inc.

2.2.2 Filename Expansion

This section should only contain review information, but inevitably, many students will need more assistance with the bracket notations—particularly the use of the exclamation mark.

Emphasize that the filename expansion occurs *before* the command line is executed. The command itself will never see, for example, a * on the command line unless the shell cannot find a filename to match it.

The shell provides a service of looking for filenames to match *patterns*. If found, it expands the pattern to one or more alphabetically sorted filenames.

- A *pattern* is a word on a command line that contains a shell metacharacter (wildcard) for filename expansion.
- The shell expands filenames before the command line is executed.
- If the shell cannot find a filename to match a pattern, then the shell leaves the word unchanged on the command line.
- The chart on the opposite page shows the metacharacters which are interpreted by the Bourne shell for filename expansion.

AV 2-4: Shell Wildcard Metacharacters

The slide reviews the syntax of the wildcard metacharacters.

Take the time to go over this exercise with the class. It will help you gauge the students' familiarity with the basic user-level workings of the shell.

Exercise: Using filename wildcards

Given a directory with these files:

report	report_feb	report_jun
report_apr	report_jan	reports

What is the output of each of these command lines?

```
echo *
```

```
echo report_*
```

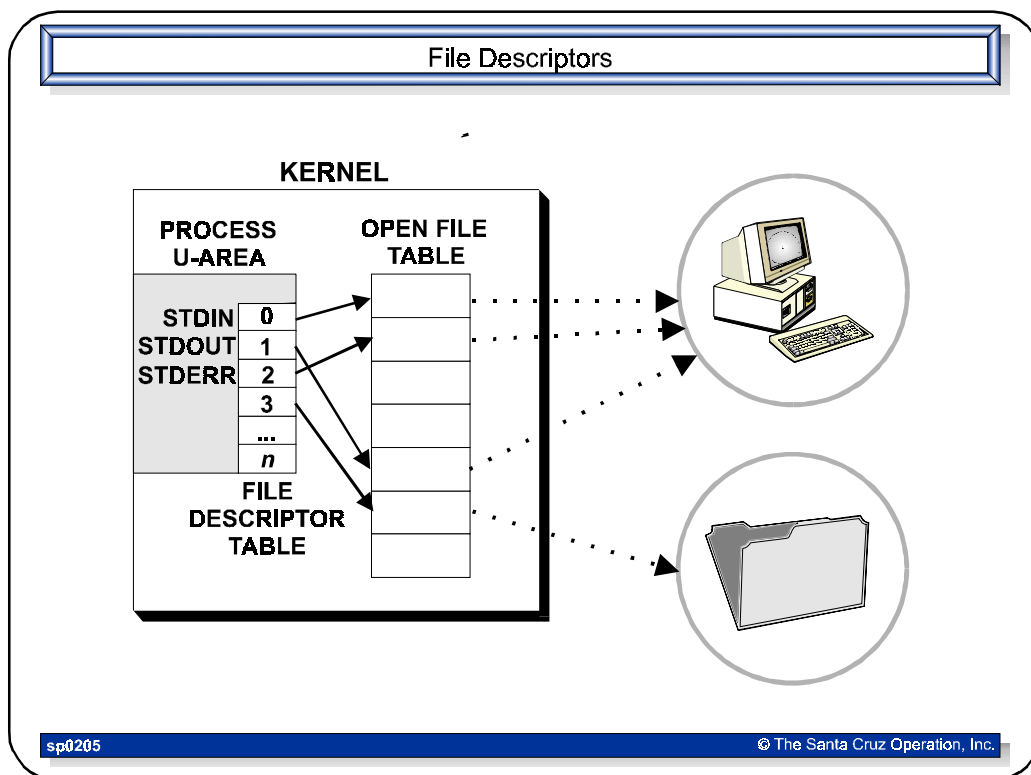
```
echo report_*
```

```
echo report?
```

```
echo re*[nl]
```

```
echo r*x
```


AV 2-5



2.2.3 I/O Redirection

An executing program (called a *process*) can read and write to many different files as it executes. Each process has a file descriptor table which the kernel uses to identify those files that the process currently has open. The process identifies to the kernel which file it wants to read or write by specifying a *file descriptor* number in its system call requests.

AV 2-5: File Descriptors

This slide shows the kernel data structures that manage the I/O of a process. Information in the file descriptor table is used to access other kernel tables (in fact, many others, hence the dotted line in the diagram), which ultimately allows I/O on a file.

2.2.3.1 File descriptors

A file descriptor describes the file upon which I/O will take place when a process is executing. In fact, it is a pointer to an entry in the systems Open File Table:

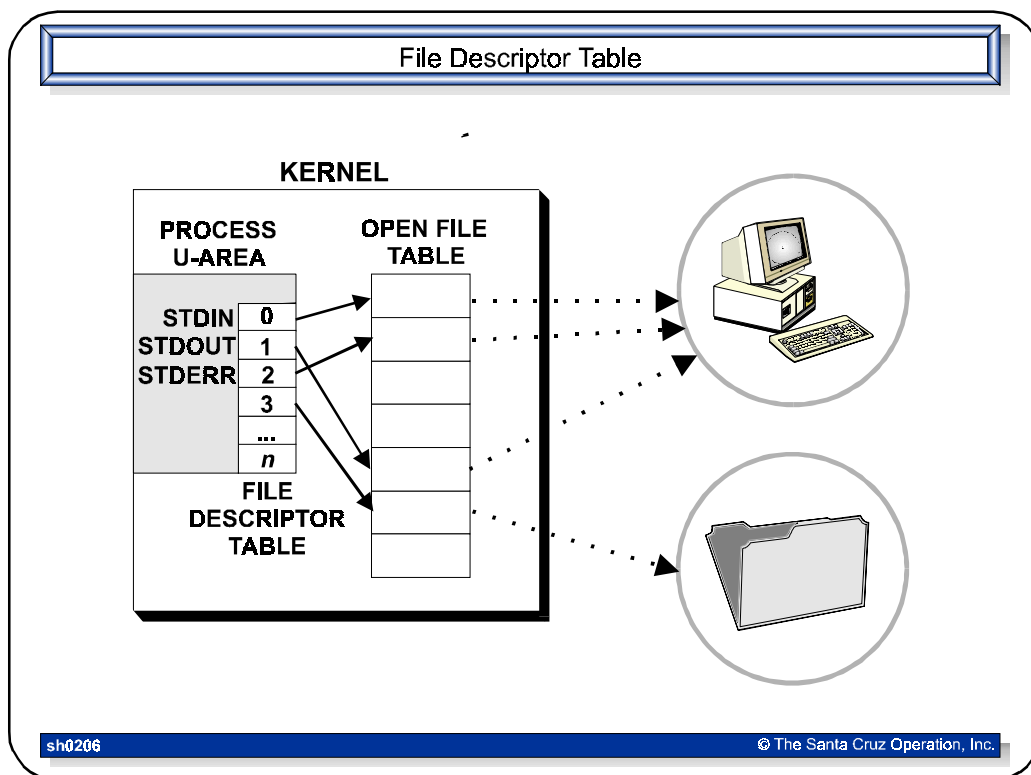
- The file descriptor number selects an entry in the file descriptor table.
- When a process starts, at least three of its file descriptors are set:
 - File descriptor 0, the standard input (*stdin*), typically refers to the control terminal, and is used for input.

Recall that all I/O in UNIX uses a file interface, which is why we have special device files for terminals and disks, as well as ordinary files for arbitrary data. The user's control terminal is just another file.

Stress that *stdin* is *not* synonymous with "terminal." Rather, it is synonymous with "file descriptor 0." It is coincidental that Descriptor 0 usually does describe a terminal. (Likewise for the other "standard" file descriptors.)

- File descriptor 1, the standard output (*stdout*), typically refers to the control terminal, and is used for ordinary output.
- File descriptor 2, the standard error (*stderr*), typically refers to the control terminal, and is used for diagnostic output.
- When the shell starts a process, it can modify the file descriptor entries for that new process so that they do not refer to the typical files (namely, the terminal).
- Shell I/O redirection occurs *before* the command starts.

AV 2-6



2.2.4 Output redirection

This material should mostly be review material, except perhaps for the `fd1>&fd2` notation, which you might need to cover more fully.

The basic shell syntax for redirecting output is as follows:

<code>> file</code> (or <code>1> file</code>)	Use <i>file</i> as standard output (file descriptor 1). If <i>file</i> does not exist, it is created; otherwise, it is overwritten (truncated to zero length).
<code>2> file</code>	Use <i>file</i> as standard error (file descriptor 2). If <i>file</i> does not exist, it is created; otherwise, it is overwritten.
<code>>> file</code> (or <code>1>> file</code>)	Use <i>file</i> as standard output (file descriptor 1). If <i>file</i> does not exist, it is created; otherwise, output is appended to <i>file</i> .
<code>2>> file</code>	Use <i>file</i> as standard error (file descriptor 2). If <i>file</i> does not exist, it is created; otherwise, output is appended to <i>file</i> .

Both the *stdout* and *stderr* can be redirected on a single command line. For example:

```
ls -l /tmp /user/tmp > goodoutput 2> errorlog
```

Use the special `fd1>&fd2` syntax to redirect both the *stdout* and *stderr* to the *same* file. For example:

```
ls -l /tmp /user/tmp > goodoutput 2>&1
```

AV 2-6: Output redirection

This slide repeats AV 2-5 showing the Open File Table so that you can explain why the `fd1>&fd2` notation is sometimes needed. Without it, both the *stdout* and *stderr* have their own independent entries in the Open File Table, as in this example:

```
$ ls -l /unix /etc/password > data 2> data
```

Because these Open File Table entries indicate where in the file I/O will next take place (the file offset), one file descriptor can easily overwrite what the other one just wrote. In this next example, however, both the *stdout* and *stderr* shares the *same* entry in the Open File Table:

```
$ ls -l /unix /etc/password > data 2>&1
```

When one file descriptor writes to the file, it effectively increments the file offset for itself and also for the other file descriptor.

2.2.5 Input redirection

Two methods used for redirecting input to a command are discussed in the following sections:

- File redirection
- In-line documents

2.2.5.1 File redirection

Most commands are designed to open a named file for their input, which allocates a new file descriptor (not 0, 1, or 2).

Many commands will default to using the *stdin* when no filename is specified. The basic shell syntax for redirecting input is:

`< file` (or `0< file`)

When redirected, the shell uses *file* as the standard input (file descriptor 0).

All three standard file descriptors can be redirected on the same command line. For example:

```
sort < inputfile > outputfile 2> errorfile
```

The purpose of this exercise is for students to recognize which file descriptors are used for different commands. Novice users who don't understand the mechanism are known to do odd things, even with the best intentions! For example, note the command from the following exercise:

```
sort data10 < data11 > data12
```

Here, **sort** opens *data10* and writes to the *stdout*, which has been changed by the shell to refer to *data12*. The file *data11* is irrelevant on this command line—the shell did indeed set up the *stdin* to refer to it, but the command (**sort**) never bothered to use that file descriptor under this invocation.

Exercise: Commands and I/O redirection

For each of these command lines, indicate which file or file descriptor the **sort** command (not the shell) is opening for input and regular output:

```
sort
```

```
sort data1
```

```
sort data2 -o data3
```

```
sort < data4
```

```
sort > data5
```

```
sort < data6 > data7
```

```
sort < data8 -o data9
```

```
sort data10 < data11 > data12
```

```
sort data13 > data13
```

```
sort < data14 > data14
```

Note that in the last two examples, the output file will be truncated to zero length by the shell before the **sort** command executes. The input file is empty, hence the output is null.

2.2.5.2 In-line documents

An *in-line document* is also known as a *here document*. In-line documents are useful in shell scripts for embedding data for a command within the script file itself, rather than in another file within the filesystem.

The syntax for using in-line documents is:

```
command << string
...
string
```

where:

<code><< string</code>	Tells the shell to read up to the line that contains <i>string</i> (with nothing else on that line) and cause the <i>stdin</i> to refer to these lines
<code><<- string</code>	Causes any leading tab characters to be stripped from the in-line document

If any character of *string* is quoted, no interpretation is placed upon the characters of the in-line document.

For example:

```
write steve << EOF
```

```
    This command is getting its input from right here!  
    The document ends when EOF appears on a line by  
    itself, so it is okay to mention EOF in this  
    message.
```

```
    EOF
```

```
    Leading or trailing spaces around the string  
    are not sufficient to terminate the message.  
    But now, here's the end of this message:
```

```
EOF
```

Do a live demo of the **write** command using an in-line document (as above). Use **write** again, but use standard input redirection for an existing file. Note that in both cases, the whole message gets transmitted at once. This can also be demonstrated with **mail**.

Point out that when the *stdin* refers to the usual terminal file, the message gets transmitted to the other user line-by-line.

Exercise: Using in-line documents

In the command line:

```
cat > datafile
```

The **cat** command opens the *stdin* (presumably the keyboard) and writes to the *stdout* (which the shell redirected to *datafile*). In this example, the *stdin* terminates when the user enters CTRL **d** from the keyboard.

Modify the command line so that the *stdin* terminates when the user enters a period (.) on a line by itself.

2.2.6 Pipelines

The caret (^) is an older syntax that has the same functionality as pipe (|), but should be avoided because it is incompatible with some other shells.

A pipeline is a sequence of commands that operate concurrently on a stream of data. Each command is separated by a vertical bar (|).

In this example of a simple pipeline, the shell executes **command1** and causes its *stdout* to act as the *stdin* of **command2**:

```
command1 | command2
```

The program **command1** must be able to write to the *stdout*; **command2** must be able to read from the *stdin*.

A command that can both write to the *stdout* and read from the *stdin* is called a *filter*. An example pipeline using filters is:

```
who | sort | tee rawoutput | pr | lp
```

GuruNotes The Bourne shell can send the *stderr* through a pipeline using this syntax:

```
cmd1 2>&1 | cmd2
```

2.2.7 Environment control

The shell provides a customized environment in which you can execute your commands. The environment includes:

- A current working directory for pathname evaluation
- A home directory for use with the **cd** command
- A **PATH** variable for locating commands
- A **TERM** variable for screen-oriented applications, such as **vi**
- An **IFS** variable for customizing how the shell parses arguments on the command line
- A file creation mask for setting default permissions on new files (see **umask**)
- Incoming mail notification services
- Signal handling services
- A library of shell built-in commands, including **cd**, **echo**, **exit**, and **pwd**

2.2.8 Programming language

The shell provides an interpreted programming language with which you can write shell scripts or use interactively at a command-line prompt.

In an interpreted language, there is no need to process any program code with a compiler:

- The interpreter (e.g., the shell) reads and executes the next program statement without regard to the next statement in the program.
- Interpreted programs tend to be slower than a similar compiled application (compiled applications usually have faster code because the compiler reads the *entire* program before it generates code, and can thus optimize some operations).
- Development time is typically much shorter than the time required for a similar compiled application.

The Bourne shell language has many of the same programming constructs as other languages, even compiled ones:

- Variables
- Comments
- Decision constructs
- Loops
- Functions

Do not go into syntax details of the shell language now. There is extensive coverage of the language later in the course. Just provide a general overview of programming languages.

2.3 Lesson Summary

The UNIX kernel controls the basic operations of the system, but does not directly interact with users—that functionality is reserved for a command line interpreter such as a shell.

The Bourne shell is widely available on many UNIX systems and provides these services:

- A command-line interpreter
- Filename expansion
- I/O redirection and pipes
- Environment control
- Interpreted programming language

It is important to note that shell special command-line services (such as variable/filename expansion and I/O redirection) occur before the command begins to execute.

Review Questions

1. What are the services provided by the kernel?
2. What are some of the actions performed by the shell on a command line before a command is executed?
3. What are the shell wildcard metacharacters? What do they mean?
4. What does the shell do when setting up I/O redirection for a command?

5. What is a *here document*?
6. What is a filter? Do all commands in a pipeline need to be a filter? Explain.
7. Name at least three items in the working environment controlled by the shell.
8. What is the difference between an interpreted language and a compiled language? What are characteristics of each?

2.4 Relevant Commands, Files, and Syntax

The following commands and files were presented in this lesson.

Relevant commands

sh	The Bourne shell
cs h	The C shell
k sh	The Korn Shell
rsh , rksh	Subsets of the Bourne and Korn shells designed for accounts with limited access to the system

Relevant files and devices

<i>unix</i>	Usual kernel file which is loaded at boot-time
<i>stdin</i>	Name of the file descriptor which a program uses for input
<i>stdout</i>	Name of the file descriptor which a program uses for normal output
<i>stderr</i>	Name of the file descriptor which a program uses for error or diagnostic messages

Relevant syntax

*	Match zero or more of any character
?	Match exactly one of any character
[<i>string</i>]	Match exactly one of any character in <i>string</i> (which can be specified as a range)
[! <i>string</i>]	Match exactly one of any character <i>not</i> in <i>string</i>
> <i>file</i> or 1> <i>file</i>	Create and write to <i>file</i> for the <i>stdout</i>
2> <i>file</i>	Create and write to <i>file</i> for the <i>stderr</i>

<code>>> file</code> or <code>1>> file</code>	Open and append write to <i>file</i> for the <i>stdout</i>
<code>2>> file</code>	Open and append write to <i>file</i> for the <i>stderr</i>
<code>< file</code> or <code>0< file</code>	Open and read from <i>file</i> for the <i>stdin</i>
<code><< string</code>	Read up to a line matching <i>string</i> for the <i>stdin</i>
<code><<- string</code>	Read up to a line matching <i>string</i> for the <i>stdin</i> , stripping leading tab characters
<code>command1 command2</code>	Execute <i>command1</i> and cause its <i>stdout</i> to act as the <i>stdin</i> of <i>command2</i>

2.5 Answers to Review Questions

1. Some of the kernel services include: filesystem management, I/O management, process control, and memory management.
2. Some of the actions performed by the shell include: reading the command line, variable substitution, filename expansion with wildcards, and I/O redirection.
3. The shell wildcard metacharacters include *, ?, and []. The asterisk character (*) matches zero or more of any character. The question mark character (?) matches exactly one of any character. [string] matches exactly one of any character in string, which can be a range of characters. [!string] matches exactly one of any character not in string, which can be a range of characters.
4. When setting up I/O redirection for a command, the shell opens a file and changes the file descriptor table of the process so that it refers to this file. This normally occurs for one of the three standard file descriptors: stdin (0), stdout (1), or stderr (2).
5. A here document is a form of I/O redirection for the standard input and is also known as an in-line document. With it, the shell reads up to a line that contains string (with nothing else on that line) and causes the stdin to refer to these lines.
6. A filter is a command which can read from the standard input and write to the standard output. All commands in a pipeline need not be filters—the first command need not be able to read from the standard input, the last command need not be able to write to the standard output. However, all other commands in the pipeline need to be filters.
7. The shell controls many items in the working environment, including: a working and home directory; a PATH variable for locating commands; a TERM variable for screen-oriented applications, such as vi; an IFS variable for customizing how the shell parses arguments when reading input; a file creation mask for setting default permissions on new files (see umask); incoming mail notification services; signal handling services; a library of shell built-in commands, including cd, echo, exit, and pwd.
8. An interpreted language uses a text file which, at run time, is read and executed line-by-line by an interpreter program (such as a shell). A compiled program originates as a text file, but is translated to another language (such as a machine or binary language) by a translator program (such as a compiler or assembler). Compiled code usually runs much more quickly than interpreted code, but is often more difficult to write than interpreted code.

3

Command Execution

Instructor Guide

Instructor Guide

Lesson 3

Command Execution

Suggested lesson duration

This lesson should take approximately 1.25 hours to complete, including the lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 3-1

The slide is titled "Command Execution" in a blue-bordered box at the top. It contains a bulleted list of four items. At the bottom, there is a blue footer bar with the text "sp0301" on the left and "© The Santa Cruz Operation, Inc." on the right.

Command Execution

- Identify three UNIX command types
- Describe a UNIX process
- Describe how the shell locates commands
- Describe three command execution techniques

sp0301 © The Santa Cruz Operation, Inc.

Instructor Guide

Introduction

The UNIX operating system has the ability to execute many programs concurrently. To effectively write and use these applications, it is useful to recognize what a program is and how it is run by the system.

This lesson describes the basic UNIX command types and how they are executed in a process. This lesson also describes how the shell executes commands on behalf of a user.

Objectives

This lesson describes the basic mechanisms of executing commands. By the end of this lesson, you will be able to:

- Identify three UNIX command types
- Describe a UNIX process
- Describe how the shell locates commands
- Describe three command execution techniques

AV 3-1: Command Execution

This first slide describes the objectives covered in this lesson.

3.1 Command Types

UNIX supports three types of commands or programs:

- Binary programs
- Shell built-in commands
- Shell scripts

The **type** and **file** commands can be used to determine the type of command.

Do this exercise interactively with the class. Note how the two commands can identify the type of command. Use **type** first—if it identifies the command as built-in, there's no need to go further. If it is not built-in, **type** provides the pathname to the file (only if it is in a \$PATH directory). If **file** cannot read the file (“cannot open for reading” error message), but you can execute the program, it must be a binary program because scripts must have read access to be executable.

Exercise: Command types

Use the **type** and **file** commands to determine the type of these commands:

- **batch**
- **clear**
- **ls**
- **set**
- **umask**

3.1.1 Binary programs

Binary programs, created by a compiler or assembler, are the most common type of command. They execute relatively quickly. Examples include:

- **cat**
- **vi**
- **ps**
- **sh**

3.1.2 Shell built-in commands

Shell *built-in* commands are part of the shell binary itself meaning:

- No separate files exist that contain the code for these commands
- Some commands can execute differently in other shells
- Some commands might also exist as binary files for backward compatibility

Built-in commands execute very quickly; they do not execute in a subshell. Examples include:

- **cd**
- **echo**
- **pwd**
- **exit**

At one time, the **echo** and **pwd** commands were not built-in commands—rather, they were binaries, which of course, executed in a sub-process. The **cd** and **exit** commands have always been built-in to the shell; they could not operate properly if they executed in a subshell.

3.1.3 Shell scripts

Shell scripts, also known as *shell programs*, *command files*, and *batch files*, are text files that contains a list of commands which are executed by a shell.

Different shells have different command syntax. For example, a script written for the Bourne shell may not execute properly in the C shell. However, Bourne shell scripts will execute properly when run by the Korn shell because it is backward compatible with the Bourne shell.

Any command line in a script can also be entered interactively at a command-line prompt (and vice versa).

Shell scripts execute in a subshell and are relatively slow. Examples include:

- **shutdown**
- **calendar**
- **spell**

3.2 Processes

Recall that a *program* is a file within a filesystem that contains code (i.e., instructions) and data to accomplish some task. A *process* is a program which has been loaded into main memory and is in the midst of execution.

Processes are identified by a unique process identification number (PID) and managed by the kernel via entries in kernel tables (such as the Process Table).

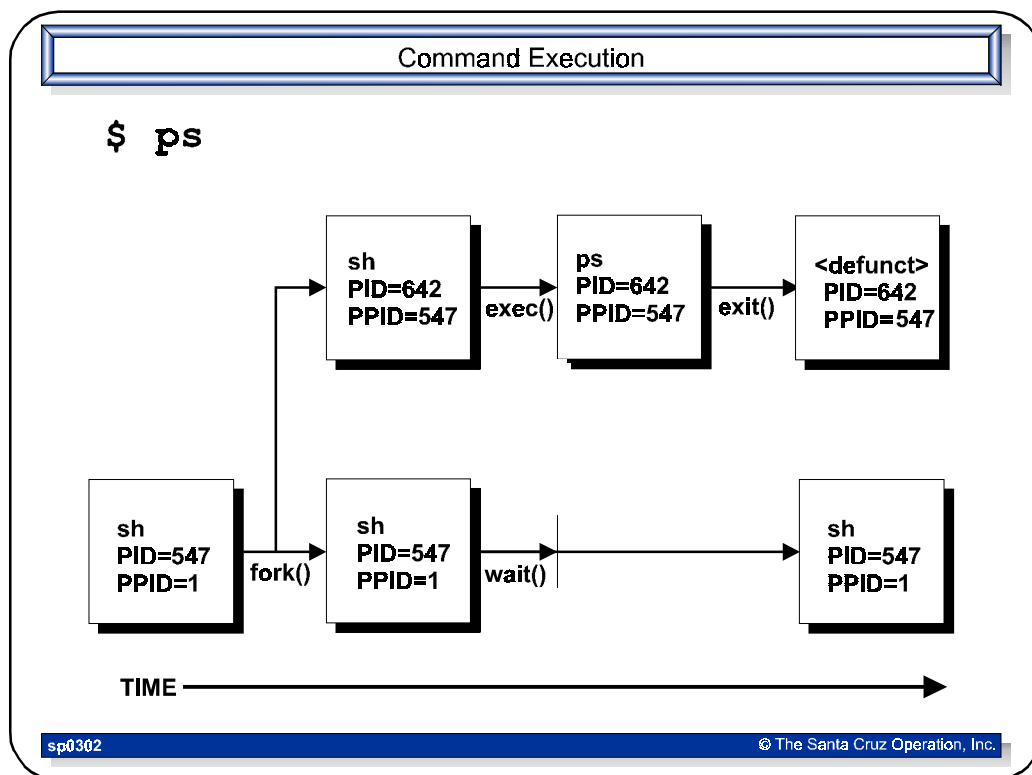
A new process is created by a parent process and is referred to as a child process. A child process knows its parent by its parent PID (or PPID).

Exercise: Parent process

Use the **ps** command to determine the parent of your login shell.

You might also request that students check out the PPID of the **ps** process itself, which is the login shell.

AV 3-2



3.2.1 Creating processes

AV 3-2: Command Execution

This slide shows the sequence of steps taken to execute the **ps** command. You might want to draw similar diagrams later in the course when explaining how certain programs execute.

Note that the boxes on the bottom row represent the *same* process seen at different points in time—each individual process has its own “row.”

Show that the top process (PID=642) goes through a change of identity after the **exec** call, but it is still the *same process*, regardless of the fact that it is now executing a different *program*.

Ask students what is different for background execution. (Answer: no **wait** call.)

Except for shell built-in commands, the execution of a command typically creates a new process. This is accomplished as follows:

1. The shell reads the command line and prepares to execute the command.
2. The shell issues a **fork** system call which creates a new process that is nearly identical to the calling (parent) shell process. (One obvious difference is the PID for the child process.)
3. The new child process issues an **exec** system call which overlays the calling process with a new program to execute, namely the program specified on the command line.
4. The child process runs the new program and eventually finishes by issuing an **exit** system call. At this point, the child process is said to have *died* or *terminated* and enters a *zombie* (defunct) state until its parent process handles its remains.
5. In the meantime, the parent issues a **wait** system call which causes it to wait until its child process has finished and terminated.

GuruNotes Regarding temporal synchronization of the parent and child processes:

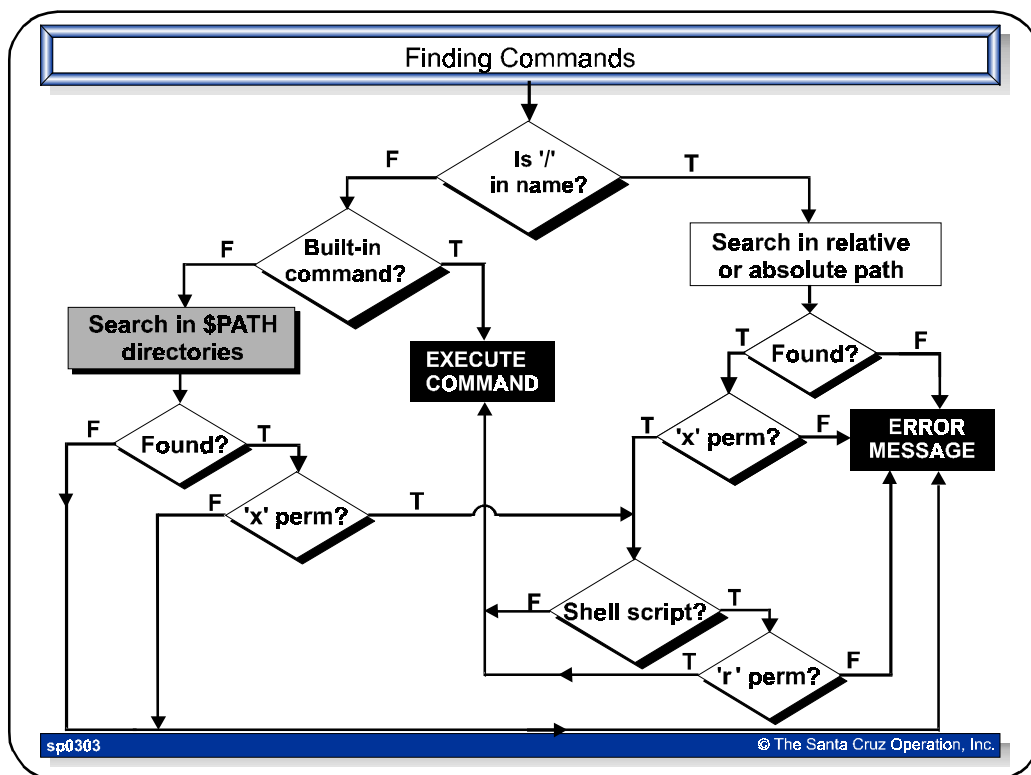
Inside the **fork** call, the kernel arranges for the child to execute before the parent continues.

The parent's **wait** commonly occurs before the child's **exit**, but this need not always occur in this order. The parent's **wait** immediately returns if the child has already died.

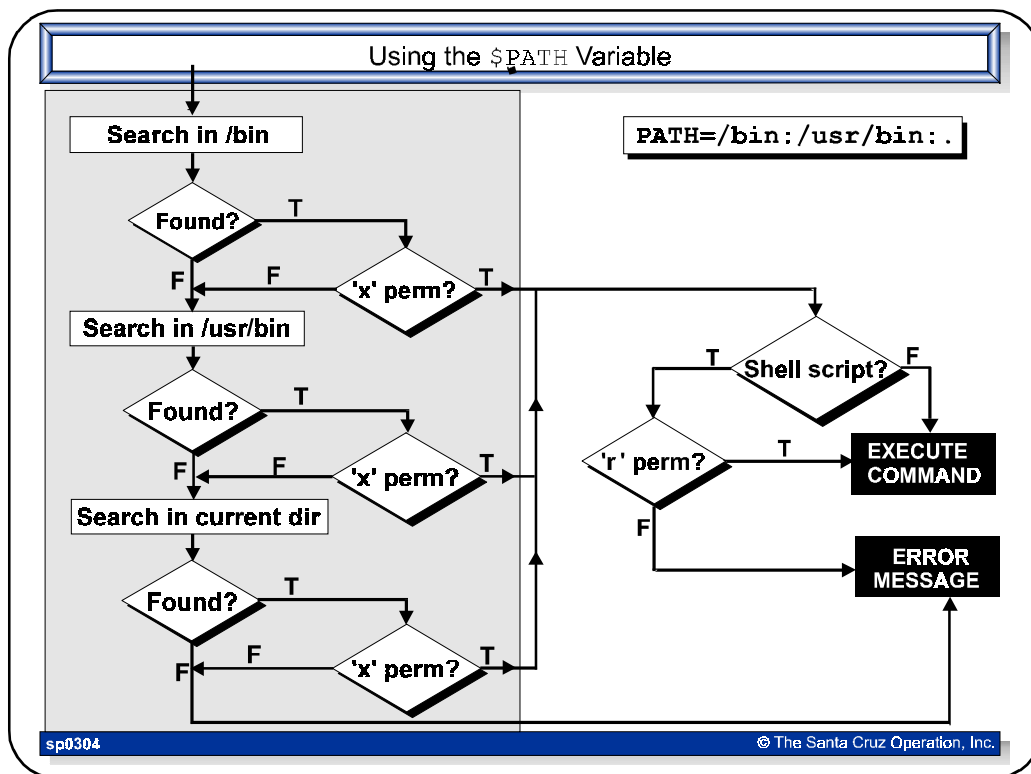
GuruNotes Some differences between the parent process and the new child process include (see **fork** for more information):

- Different PID and PPID
- No signals are pending in the child process
- Process locks, text locks, and data locks are not inherited by the child process
- Return value from **fork** system call (0 for child, non-zero for parent)

AV 3-3



AV 3-4



3.2.2 Finding commands

AV 3-3: Finding Commands

This flow chart shows how commands are found and executed. The shaded box is expanded in AV 3-4.

AV 3-4: Using the \$PATH Variable

In this slide, the shaded box shows how the PATH variable is used to find commands. Note that if the command is found, but execute permission is not granted, the search continues. If execute permission is granted and the command is a script, then read permission is required, too, otherwise an error occurs. The search does not continue if read access is not granted.

A command is defined as a program or shell built-in that begins a shell command line.

The shell locates these commands in different ways, depending on the type of the command and how the command was specified on the command line. This is shown in the diagrams on the opposite page.

- A command which has a slash (/) character in the command name (for example, **/bin/date**, **bin/date**, **./date**) specifies a particular path to the command
- Built-in commands are already known to the shell and are directly executed.
- Any other commands must be found in the filesystem. The shell uses the PATH variable to locate them. (Shell variable syntax is covered in detail in a later lesson.)

To execute binaries and scripts as commands, you must have execute permission on the files. (Later, you will see that you do not require execute permission to run a script unless you want the script to be treated as a command.)

The intent of this exercise is to dispel the misconception that the current directory is always searched when looking for commands. It isn't.

Note also that the shell expects to see a directory path on both sides of a colon (:) character. If nothing is there, the shell assumes the current directory (i.e., dot (.)), hence the last two examples are the same.

The **root** account commonly does not search the current directory for security reasons (it helps avoid "Trojan horses"). For any account, if the current directory is searched, it is best to search it after the other directories for these same security reasons.

Exercise: Using the PATH variable

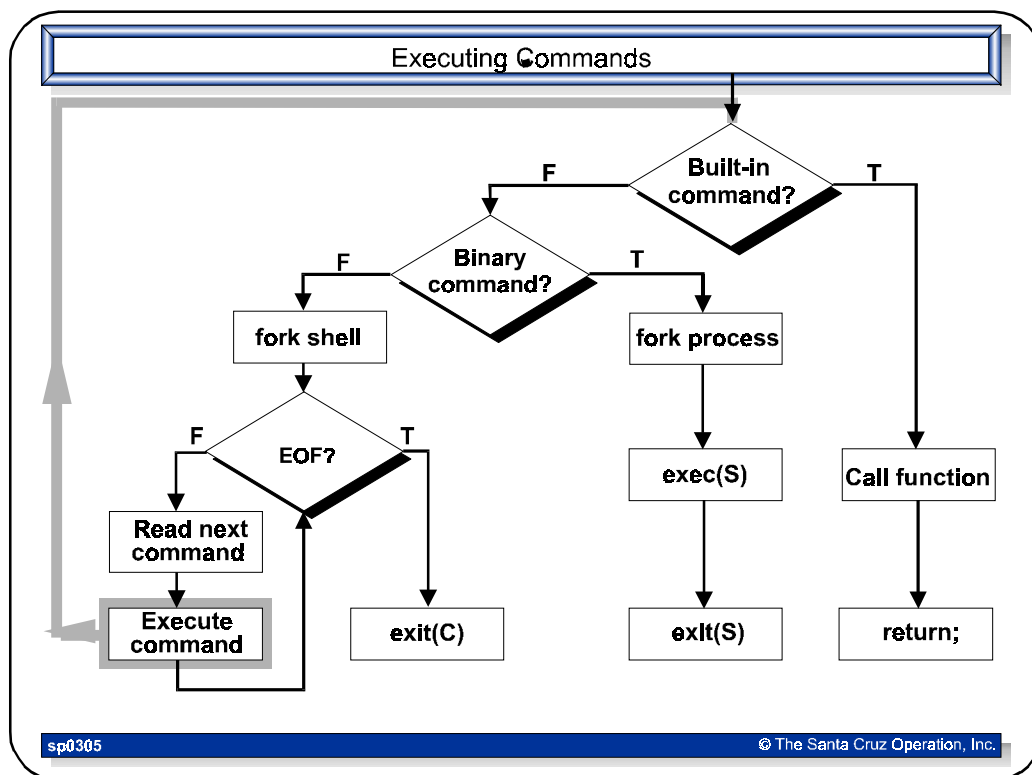
For which of these PATH variables will the current directory be searched for a command? Verify your answer.

PATH=/bin:/usr/bin

PATH=/bin:/usr/bin:

PATH=/bin:/usr/bin:.

AV 3-5



Instructor Guide

3.2.3 Executing commands

AV 3-5: Executing Commands

This flow chart shows how commands are executed. Show the recursive nature of the shaded box “Execute command” at the bottom of the diagram. The shaded box repeats the entire process of determining whether the next command in a shell script is a binary, built-in, or shell script.

The three different types of commands—binaries, built-ins, and scripts—execute in somewhat different ways, as shown in AV 3-5.

- Binary programs execute as shown in AV 3-2 on page 3-10. A process is **forked**, the program is **execed**, and ultimately **exits**.
- Shell built-ins are executed directly by the shell. No subshell is created because the command is executed as a function call within the shell binary. Ultimately, the function terminates and **returns**.
- Shell scripts are executed in a subshell. The new shell reads, interprets, and executes each command line in the script until it encounters an **exit** command or the end-of-file (EOF).

The execution of these commands is recursive in nature. If the command is a binary command, the shell forks a process which is overlaid by the binary program. If it is a built-in command, then it is executed directly by the shell. Script commands are executed in a subshell that in turn is interpreted by another subshell, and so on. Eventually, a binary or built-in command is encountered which stops this recursion.

3.3 Execution Techniques

The shell provides techniques which offer flexibility when executing commands. This section describes some of these techniques.

3.3.1 Background execution

Normally, the shell **waits** for its children to **exit** before continuing. This is called *foreground* execution.

For commands that take a while to complete and require no user intervention, the waiting can be suppressed by terminating the command with the ampersand (&) character. This is called *background* execution.

To accomplish background execution, the shell omits the **wait** call. Both parent and child get to execute concurrently.

The shell displays the PID of the new background process when it starts. For example:

```
$ sort datafile > output &
24355
```

The ampersand character (&) acts as a command terminator, so another command might directly follow. For example:

```
$ sort datafile > output & ps
24370
  PID  TTY      TIME COMMAND
 24363  01        0:00 sh
 24370  01        0:00 sort
 24371  01        0:00 ps
$
```

The shell changes the default file for the *stdin* to be */dev/null*.

It is usually a good idea to redirect the output of the background command if it writes to the *stdout*, *stderr*, or both; otherwise, the output might appear on the display at inopportune times.

3.3.2 Command grouping

Two or more commands can be specified on one command line, separated by a semi-colon (;) character. Each command runs in the foreground from the current shell. For example:

```
$ ps ; who
  PID  TTY      TIME COMMAND
 24363 p00      0:00 sh
 24382 p00      0:00 ps
sue      tty2c      Sep 08 07:54
steve    tty2f      Sep 08 08:55
david    tty00      Sep 08 15:33
```

A command list placed within parentheses executes in a subshell. For example:

```
$ (ps ; who)
  PID  TTY      TIME COMMAND
 24363 p00      0:00 sh
 24384 p00      0:00 sh
 24385 p00      0:00 ps
sue      tty2c      Sep 08 07:54
steve    tty2f      Sep 08 08:55
david    tty00      Sep 08 15:33
```

The subshell **forks** and **execs** for each binary and script command in the command list, except for the last command in the list for which the subshell only **execs**. This is readily observed with this example:

```
$ (ps; ps; ps)
  PID  TTY      TIME COMMAND
 24363 p00      0:00 sh
 24393 p00      0:00 sh
 24394 p00      0:00 ps
  PID  TTY      TIME COMMAND
 24363 p00      0:00 sh
 24393 p00      0:00 sh
 24397 p00      0:00 ps
  PID  TTY      TIME COMMAND
 24363 p00      0:00 sh
 24393 p00      0:00 ps
```

Command grouping is useful for I/O redirection of several commands. For example:

```
$ (ps ; who) > statusfile
```

Without the parentheses, only **who**'s *stdout* is redirected.

Command grouping is also useful for executing commands so that they do not affect the current shell environment (such as the current working directory). For example:

```
$ (cd /path/to/some/other/dir; vi datafile)
```


GuruNotes Another type of command grouping uses curly braces:

```
{ command1; command2; command3; }
```

This notation, which requires a space (or return) after the left brace and a semi-colon (or return) before the right brace, groups the commands to be executed by the *current* shell rather than in a *subshell*, as is the case with the parentheses grouping notation.

```
cmd1; cmd2           # Both execute in current shell
(cmd1; cmd2)          # Both execute in a subshell
{ cmd1; cmd2; }       # Both execute in current shell
```

If I/O redirection or background execution occur, then the brace notation operates exactly like parentheses—a subshell *will* be created to execute the commands!

```
cmd1; cmd2 > f         # Both execute in current shell;
                        # but only cmd2 redirected
(cmd1; cmd2) > f        # Both execute in a subshell
                        # but both are redirected
{ cmd1; cmd2; } > f    # Both execute in a subshell
                        # but both are redirected
```

So far, it appears that the braces have no distinctive operation. Their primary use is for decision constructs (which the students are not ready to learn yet!). For example:

```
cmd1 && cmd2 ; cmd3
    # execute cmd1 and cmd3 always; cmd2 conditionally
cmd1 && (cmd2 ; cmd3)
    # execute cmd1 always; cmd2 & cmd3 (in subshell)
    # conditionally
cmd1 && {cmd2 ; cmd3; }
    # execute cmd1 always; cmd2 & 3 (in current shell)
    # conditionally
```

Exercise: Command grouping

How do these command lines differ?

```
ps ; who &
```

```
(ps ; who) &
```

How do these command lines differ?

```
ps > datafile ; who > datafile
```

```
(ps ; who) > datafile
```

What is displayed by this command sequence?

```
cd /etc  
(cd /usr/bin)  
pwd
```

3.3.3 The dot (.) command

Normally the shell creates a subshell to execute a script. The creation of a subshell can be suppressed by executing the script with the built-in dot (.) command.

Dot causes the current shell to read and execute the commands from a named script file. For example:

```
$ . myscript  
$ . .profile
```

The PATH variable is used to find the directory containing the script.

The dot command works only for shell scripts; it cannot be used to execute a binary program.

GuruNotes When invoked with the dot command, a script cannot access any command-line arguments via its positional parameters. For example, when a script executes via the dot command, \$1 will refer to the *current shell's* first positional parameter rather than the first argument on the invocation line.

GuruNotes The dot command is commonly used to read shell functions into a script.

3.4 Lesson Summary

UNIX supports three command types:

- Binary programs
- Shell built-in commands
- Shell scripts

The shell executes binaries and scripts with child processes by using a sequence of **fork**, **exec**, **exit**, and **wait** system calls. Because they run in a subshell, they cannot affect the parent's working environment. On the other hand, shell built-in commands are executed by the current shell and therefore *can* affect its environment.

The shell allows flexibility in command execution. Commands can be run in the background and grouped together. Further, the current shell can be made to directly execute a shell script with the dot (.) command.

Review questions

1. How can you determine whether a command is a binary, built-in, or script?
2. Which type of command generally runs the fastest: a binary, built-in, or script? Which runs the slowest? Why?
3. What is the difference between a program and a process?
4. When the `PATH` variable is set up to search the current directory for a command, the current directory is usually set up to be the last component of the search path. Why might this be a good security precaution?

5. How do you tell the shell to execute a command in the background? How does the shell do it?

6. What happens when you enter a command line within a set of parentheses?

7. What does the dot (.) command do? Which type of command is it: a binary, built-in, or script? Why?

3.5 Relevant Commands and Files

The following commands and files were presented in this lesson.

Relevant commands

dot (.)	Causes the current shell to read and execute the commands from a named shell script
file	Classifies files into file types
type	Shell built-in command which indicates how it would be interpreted by the shell

Relevant files

<i>/dev/null</i>	A special file which discards all data written; reads from null always return 0 bytes
------------------	---

3.6 Answers to Review Questions

1. Use the `type` and `file` commands to determine whether a command is a binary, built-in, or script. The `type` command will identify whether a command is built-in or not; the `file` command can distinguish binary commands from shell scripts.
2. Shell built-in commands generally run the fastest because no new processes need to be created for them to run. Binary commands generally run faster than scripts because the compiler is usually able to produce efficient, optimized code and the program usually runs as a single process. Shell scripts generally run the slowest of all three command types because they usually require many processes to be generated as the program executes.
3. A program is a file that contains code and data to perform a given task. A process is a program that is currently executing.
4. The current directory is usually set up to be the last component of the search path to help prevent potential security problems. The user will execute the “real” command instead of being spoofed by running a different program with the same name in the current directory.
5. Terminate a command with an ampersand (&) to tell the shell to execute a command in the background. This causes the shell to omit the `wait` system call that it would ordinarily make.
6. When you enter commands within a set of parentheses, they are executed in a subshell.
7. The dot (.) command causes the current shell to read and execute a shell script—without creating a subshell first. It is a built-in command—and must be—so that a subshell is not created first.

4

Basic Scripts

Instructor Guide

Instructor Guide

Lesson 4

Basic Scripts

Suggested lesson duration

This lesson should take approximately 1.5 hours to complete, including the lab exercises.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 4-1

Instructor Guide

Basic Scripts

- Define the term *shell script*
- Create and execute a simple shell script
- Place comments in a shell script
- Specify which shell is to execute a script
- User the **echo** command
- Use shell options

sp0401© The Santa Cruz Operation, Inc.

Introduction

As you have learned already, the shell provides an interpreted programming language with which you can write shell scripts. These scripts automate routine tasks that might otherwise require you to enter several lines at a command-line prompt.

Shell scripts can be rather sophisticated and use many different programming constructs, such as variables and control flow statements. Likewise, they can be fairly simple in design and operate much like a *batch* file that is found in other operating systems (e.g., MS-DOS *.bat* files). This lesson describes how to write these basic kinds of scripts.

Objectives

This lesson describes how to write and execute simple shell scripts. By the end of this lesson, you will be able to:

- Define the term *shell script*
- Create and execute a simple shell script
- Place comments in a shell script
- Specify which shell is to execute a shell script
- Use the **echo** command to display messages
- Use shell options to trace execution of a shell script

AV 4-1: Basic Scripts

This slide describes the objectives covered in this lesson.

4.1 Overview

A *shell script* is a text file that contains a list of commands that are executed by a shell. They are also known as *shell programs*, *command files*, and *batch files*.

Different shells have different command syntax, so a script written for the Bourne shell might not work properly in other shells.

Any command line in a script can also be entered interactively at a command-line prompt (and vice versa). This means that the commands in a script can use:

- I/O redirection and pipelines
- Filename metacharacters (wildcards)
- The shell environment
- Any other shell feature

Shell scripts generally execute slowly, but require much less development time than a corresponding binary program.

Stress that any command line in a script can also be entered interactively at a command-line prompt (and vice versa). Knowing this simplifies writing scripts. It also comes in handy later when the flow control constructs are learned—they can be easily tested at the command line without having to create a script file first.

AV 4-2

Sample Shell Script

```

$ cat sample1
echo Today is
date
echo Calendar for this month:
cal

$ sh sample1
Today is
Sun Jun 28 19:52:46 EDT 1998
Calendar for this month:
  June 1998
Su Mo Tu We Th Fr Sa
 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

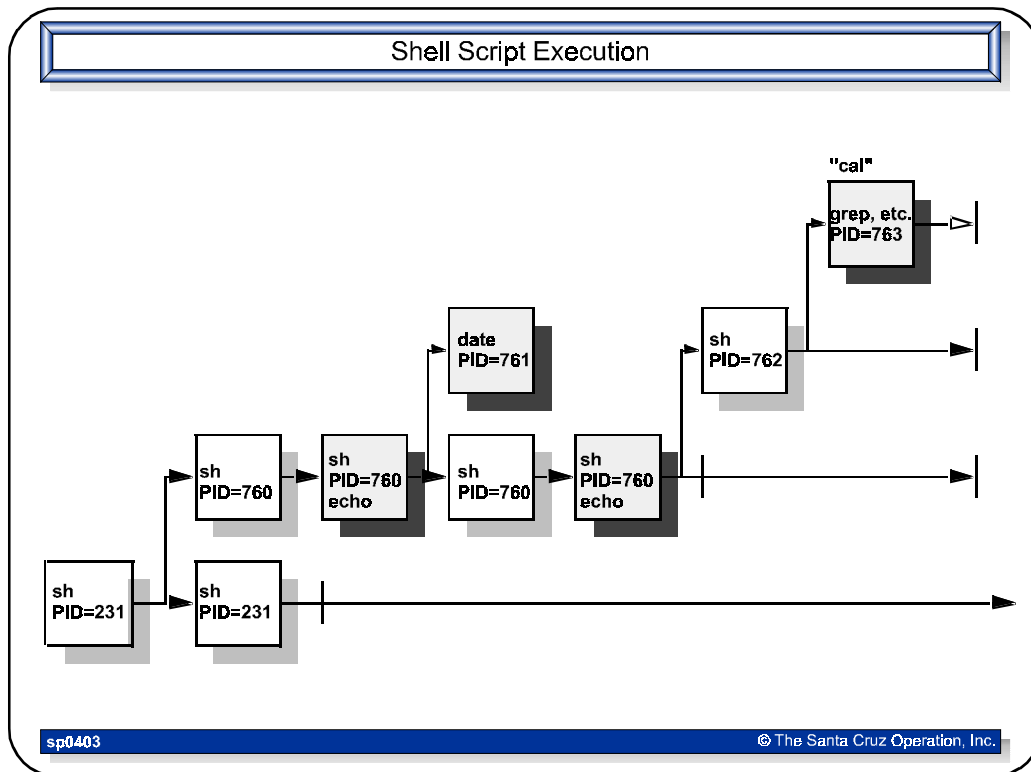
$ chmod +x sample1
ls -l sample1
-rwxr-wr-w 1 steve admin 53 Jun 28 19:56 sample1

$ sample1
Today is
Sun Jun 28 19:52:46 EDT 1998
Calendar for this month:
  June 1998
Su Mo Tu We Th Fr Sa
 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

```

sp0402 © The Santa Cruz Operation, Inc.

AV 4-3



4.2 Shell Script Execution

A shell script is a text file that contains a list of commands. This is a simple example:

```
echo Today is
date
echo Calendar :
cal
```

Once written, a script is executed in a subshell either explicitly or implicitly.

AV 4-2: Sample Shell Script

This slide shows a script file which demonstrates each type of command (binary, script, built-in) being executed from within a script.

AV 4-3: Shell Script Execution

This slide shows how the shell script in AV 4-2 executes. Note:

- To conserve space in this diagram, the extra boxes depicting the **exec** calls are omitted.
- The shaded boxes are the commands in the script.
- The “cal” box represents all additional processes necessary to complete the **cal** command (a binary command).
- Because **echo** is built-in, no process is created for it.

4.2.1 Explicit execution

To explicitly invoke a subshell to execute a script, enter:

```
$ sh script
```

This creates a subshell which reads its input from the named *script*. You must have read permission on *script* to execute it in this explicit fashion.

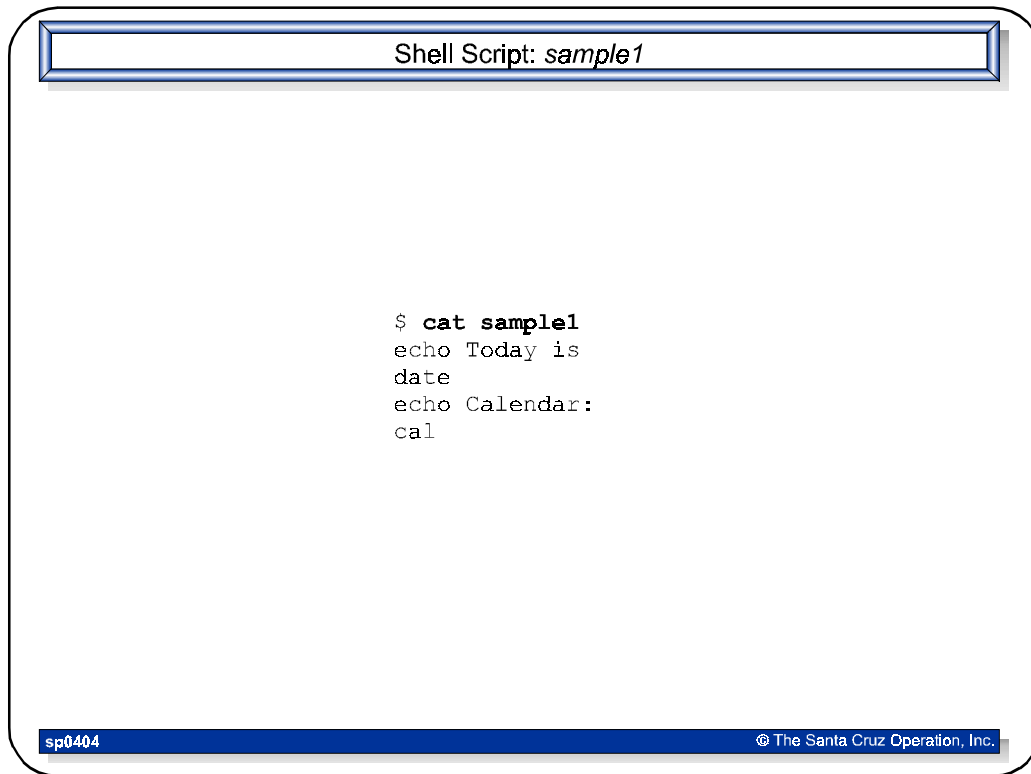
4.2.2 Implicit execution

To implicitly invoke a subshell to execute a script, enter:

```
$ script
```

This also creates a subshell which reads its input from the file named *script*. To implicitly execute a script, you must have both read and execute permissions on *script*. See **chmod** for more information about setting permissions.

AV 4-4



4.2.3 Explicit vs. implicit execution

Explicit and implicit execution provide nearly identical results:

- The same number of processes is generated for either technique.
- The `PATH` variable locates the script file only for implicit execution; it is needed to locate the `sh` command for explicit execution.
- The shell can accept options more flexibly with explicit execution (shell options are discussed later in this lesson)

Execution via the two methods is virtually identical. The difference is that with explicit execution, shell functions in the current shell are not “exported” to the new child process. Students learn about functions later in the course.

AV 4-4: Shell Script: *sample1*

This slide repeats the script discussed in Section 4.2 on page 4-9 to aid students in completing the exercise.

Exercise: Explicit and implicit execution

- | | |
|---------|--|
| Step 1. | Create a shell script named <i>sample1</i> that contains the four lines of code shown in AV 4-4. |
| Step 2. | Execute the script explicitly with the <code>sh</code> command. |
| Step 3. | Now, use <code>chmod</code> to give the file execute permission. Execute the script implicitly. |
| Step 4. | Add a fifth line to the script to invoke the <code>ps</code> command. Execute the script again using both techniques to verify that their execution is the same. |

AV 4-5

Comments

```
$ echo one # two
one
$ echo one #two
one
$ echo one#two
one#two
```

sp0405© The Santa Cruz Operation, Inc.

Instructor Guide

4.3 Comments

AV 4-5: Comments

This slide shows several examples using the comment character, pound (#).

A comment is a line (or part of a line) that does not execute, but instead provides internal documentation for a program.

The Bourne shell uses the *pound* (#) character (also called “hash” or “sharp”) to create a comment and uses the following rules:

- A word or argument that begins with # causes that word and the following characters up to the newline character to be ignored.
- The pound character (#) has no special meaning if it is embedded in the middle of a word.
- A single comment cannot extend beyond one line, however many adjacent lines can be “commented out.”

The following are suggested practices for using comments:

- Use comments at the beginning of the script to document basic functionality, syntax, known limitations, and revision history.
- Every line does not need a comment, but lines that are unclear or “tricky” should be well documented.
- Blocks of code are typically preceded by comments to describe what the upcoming code block does.
- Blank lines can also be used to improve the readability of a shell script by making blocks of code more obvious.

You might not think it necessary to include comments for yourself, but be aware that someone else might need to read your code someday!

4.4 Which Shell Executes a Script?

Several shells are available under SCO UNIX, including **sh**, **csh**, and **ksh**. The different shells have different programming syntax and can be incompatible.

Shell scripts written for the Bourne shell (**sh**) can be executed by Bourne and Korn shell users without problems—the Korn shell (**ksh**) is a superset of the Bourne shell.

C shell users can sometimes have difficulty executing a script written for the Bourne shell. The C shell executes scripts under the C shell when *both* of the following are true (otherwise, it uses the Bourne shell):

- The `shell` environment variable is set to `/bin/csh` (this is usually true for most C shell users).
- The first character in the script is the pound (`#`) character.

The different shells have different programming syntax and can be incompatible. For example, here are comparable Bourne/Korn shell (`$` prompt) and C shell (`%` prompt) command lines:

```
$ PATH=/bin:/usr/bin:.  
% set path=(/bin /usr/bin .)  
  
$ MYVAR=value  
% set MYVAR=value  
  
$ NEWVAR=value; export NEWVAR  
% setenv NEWVAR value  
  
$ cmd > file 2>&1  
% cmd >& file  
  
$ for WORD in str1 str2 str3  
> do  
>   cmd-list  
> done  
% foreach WORD (str1 str2 str3)  
>   cmd-list  
> end
```



```
$ while [ $VAR -lt 10 ]
> do
>     cmd-list
> done
% while ($VAR < 10)
>     cmd-list
> done
```

4.4.1 Traditional shell determination

To ensure that a Bourne shell script can be executed by C shell users, do not begin a script with the pound (#) character. The pound character (#) designates a comment and is often used to start a script. For portability, begin scripts with the built-in null command (:). For example:

```
:
# This script begins with the null command and can
# reliably be executed by C Shell users

echo Today is
date
echo Calendar:
cal 2> /tmp/errors
```

The main point about starting a script with the null command is that it acts as a benign placeholder that prevents the first character in the script from being the pound (#) character. This is sufficient to ensure that a C shell does not execute the script *for C shell users* (it was never an issue for Bourne/Korn shell users).

4.4.2 Hashpling

A more recent method to specify the shell that executes a script is called the *hashpling* mechanism. The hashpling mechanism directly indicates which shell is to execute the script.

To indicate the Bourne shell, begin a script with:

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

To indicate the C shell, begin a script with:

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

To indicate some other shell or interpreter, begin a script with:

```
#!/path/to/interpreter
```

For example:

```
#!/bin/sh
# This script begins with a hashpling and can
# reliably be executed by C Shell users

echo Today is
date
echo Calendar:
cal 2> /tmp/errors
```

The hashpling mechanism disables the “exporting” of shell functions to child shells. This is mentioned again in Section 9.1.4, “Exporting functions,” on page 9-11.

4.5 The echo Command

The **echo** command writes to the standard output. Its arguments are separated by a space and terminated by a new-line character. It is typically used for:

- Generic print statements in a shell script
- Prompting users before input
- Displaying contents of shell variables
- Displaying error messages

The syntax is:

```
echo [-n] [arg] ...
```

The **-n** option suppresses the terminating newline character and is sometimes useful when prompting users before input.

A built-in version of this command is provided by **sh**, **ksh**, and **csk** and might behave differently. You can assure execution of the binary command by entering **/bin/echo**.

AV 4-6

echo Escape Sequences

\b	Backspace
\c	Print line without newline (equivalent to -n option)
\f	Formfeed
\n	Newline
\r	Carriage return (no linefeed)
\t	Tab
\v	Vertical tab
\\	Backslash
\0n	The character whose ASCII code is a 1, 2 or 3-digit octal number

sp0406

© The Santa Cruz Operation, Inc.

4.5.1 echo escape sequences

AV 4-6: echo Escape Sequences

This slide lists the escape sequences that can be used with **echo**.

Point out that the `\0n` notation requires the leading zero and that up to three octal digits may follow. Refer students to the *ascii* manual page for an ASCII table. Note that `\07` is equivalent to `\007`.

The **echo** command also understands C-like escape conventions for representing characters that are otherwise difficult to type.

These sequences need to be protected from shell interpretation with a quoting mechanism (covered in a later lesson). The use of single quotes ('...') is frequently adequate for this need.

Examples using escape characters with **echo** include:

```
echo 'Enter your name ==> \c'  
echo 'Please try again.\07'  
echo 'Two\nlines'
```

See the *ascii* manual page for a complete list of *ascii* equivalents.

4.5.2 Error messages

Programs which display error messages typically write to the standard error (*stderr*). The **echo** command always writes to the standard output (*stdout*).

The shell can cause a command's *stdout* to refer to the *stderr* with the following syntax:

```
command >&2
```

For example:

```
echo 'Please try again \007' >&2
```

Exercise: The echo command

What does the following display?

```
echo What is this?... '\0101'
```

What happens when the single quotes are removed?

What happens when the leading zero is removed from the escape sequence?

What is an equivalent **echo** command for the following?

```
echo -n No newline after this
```

Replace the following three lines with one **echo** statement that displays identical output:

```
echo SCO, Inc.  
echo Santa Cruz  
echo California
```

4.6 Debugging Techniques

Like other commands, **sh** can take command-line options to affect its operation. Two of these options, **-v** and **-x**, are useful for debugging shell scripts. Refer to the **sh** manual pages for a description of other command-line options.

■ Demonstrate the debugging flags with a live demo on the LCD panel.

4.6.1 -v option

The **-v** option causes the shell to display shell input lines as they are read.

The effect of this option occurs very early in the command line parsing sequence, so the output shows exactly what was input to the shell. The syntax is:

```
sh -v script
```

4.6.2 -x option

The **-x** option causes the shell to display shell input lines just before they are executed.

The effect of this option occurs near the end of the command-line parsing sequence:

- After variables and wildcards are expanded
- After I/O redirection and command substitution are performed

The output shows what is executed by the shell, with each output line preceded by the plus character (+). The syntax is:

```
sh -x script
```

Both the **-v** and **-x** options can be specified. The syntax is:

```
sh -vx script
```

4.6.3 The set command

■ The **set** command has additional functionality which will be covered in later lessons.

The **set** command is a shell built-in command that affects the current shell's options and flags. For example, to enable the **-v** option to the current shell so that all shell input lines are displayed as they are read, enter:

```
set -v
```

To disable the **-v** option to the current shell, enter:

```
set +v
```

The **-x** option can be enabled and disabled in the same way.

The **set** command can be useful for debugging a portion of a script:

- Enable **-v** and/or **-x** just before questionable code
- Disable the flags past this section

Demonstrate the **set** command and its effect on your login shell. Set the **-x** option and then execute commands such as:

```
$ echo $PATH
$ file /dev/fd0*
```


4.7 Lesson Summary

The shell accepts commands equally well from interactive users as from text files which contain a listing of commands to execute. This latter form is referred to as a shell script.

Shell scripts are interpreted line-by-line by a shell and provide similar programming features as other programming languages. This lesson discussed how to create, execute, and debug a simple “batch” script; and also how to issue basic output statements using the **echo** command.

Review questions

1. What is a *shell script*?
2. Which file permissions are *required* to execute a shell script? Which are useful, but not necessary?
3. Where are recommended places to place a comment in a shell script?
4. Why do many scripts begin with a colon (:)? What does it represent? With what do more recently developed scripts begin instead?
5. How can you cause **echo** to write to the *stderr* for diagnostic messages?

6. What is the difference between the shell's **-v** and **-x** options? How do following commands differ?

```
sh -v script
set -v
```

4.8 Lab: Working with Batch Files

This lab will give you experience writing basic shell scripts that use only sequential flow constructs—much like MS-DOS batch files. Also practice debugging techniques by executing your scripts with the shell flags **-v** and **-x**.

4.8.1 Your login environment

In this lab, you will modify your login environment to execute a set of commands each time you log in.

4.8.1.1 Overview

The login work environment for Bourne shell users is initially established by commands located in the system file */etc/profile* and is then further established by a user's personal *.profile*.

Modify your *.profile* start-up file so that these activities occur during login:

- Set the screen colors (hardware permitting).
- Update a personal log with the current date and time.
- Automatically invoke a mailer to read any new mail that may have arrived.
- Enable messages to be sent to your screen using the **write** command.

On OpenServer systems, **prwarn** is automatically added to *\$HOME/.profile*. This command is not supported by UnixWare 7. Although it does not appear in the solution to this lab in Appendix B: "Answers to Lab Exercises," students working on SCO OpenServer™ Release 5 systems might see it in their *.profile*.

4.8.1.2 Description

These commands might be useful for this exercise:

- **setcolor**
- **date**
- **mail**
- **write**

- **mesg**
- **echo**

4.8.2 System status

In this lab you will write a script which displays status information for your system.

4.8.2.1 Overview

Generate a report containing current status information including:

- Who is logged onto the system
- Which filesystems are mounted
- Which processes are running on the system
- Printer queue status

To improve readability of the report, a text header should be displayed before each section of the report.

4.8.2.2 Description

The program should be written so that information is not written to the screen faster than a user can read it. Think of ways to handle this situation. These commands might be useful for this exercise:

who

df

ps

lpstat

sleep

echo

more

4.9 Relevant Commands

The following commands and files were presented in this lesson.

Relevant commands

chmod	Changes the mode (permissions) of a file
echo	Displays arguments to the standard output
set	Bourne shell built-in command which can set shell options for the current shell
ascii	Displays a map of the ascii character set

4.10 Answers to Review Questions

1. A shell script is a text file that contains a list of commands to be executed by the shell. A script is also known as a shell program, a command file, and a batch file.
2. Read permission is always required to execute a script. Execute permission allows you to execute the script as a command. Without it, you must invoke the `sh` command explicitly and provide the name of the script as an argument.
3. Comments should be placed: At the beginning of the script, to describe the overall functionality, syntax, limitations, and revision history Before each major block of code, to describe what the block does Before any code whose purpose is not readily apparent
4. Many scripts begin with the null command (`:`) so that C shell users can execute the script. When a script begins with a non-comment character (`#`), the C shell uses the Bourne shell to execute the script. To avoid this guesswork, more recent scripts use the hashpling mechanism to explicitly indicate the shell which will execute the script. For example: **`#!/bin/sh`**
5. You can cause `echo` to write to the `stderr` with shell I/O redirection. For example: **`echo message >&2`** Note that in some versions of UNIX, `echo` has a `-e` option that directly causes `echo` to write to the `stderr`.
6. The **`-v`** option causes the shell to display the command line exactly as it was entered. The **`-x`** option causes the shell to display the command line later in the command-line parsing sequence, therefore the display shows the results of variable expansion, filename expansion via wildcards, and other command line processing. The first command causes a new shell to be created to execute the file named `script`. The **`-v`** option will be in effect as it executes. The second command causes the **`-v`** option to be in effect for the current shell for subsequent operations.

5

Shell Variables

Instructor Guide

Instructor Guide

Lesson 5

Shell Variables

Suggested lesson duration

This lesson should take approximately 3.5 hours to complete, including the lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 5-1

Instructor Guide

Shell Variables

- Create and access user-defined variables
- Describe the shell's standard environment variables
- Use the **read** command to obtain input
- Use positional parameters in a shell script
- Define special variables

nf0501© The Santa Cruz Operation, Inc.

Introduction

Like other programming languages, the shell provides a mechanism to store and change values that may be needed in a program. This *variable* mechanism stores string values for later use. A program design that uses these variables can make a program more flexible than one that uses *constant* (or unchangeable) values only.

This lesson describes the use of the Bourne shell's four basic variable types:

- User-defined
- Environment
- Positional parameters
- Special

Objectives

This lesson describes the use of shell variables in scripts and in the general working environment. By the end of this lesson, you will be able to:

- Create and access user-defined variables
- Describe the shell's standard environment variables
- Use the **read** command to interactively obtain input from a user
- Use positional parameters in a shell script
- Define the shell's special variables

AV 5-1: Shell Variables

This first slide describes the objectives covered in this lesson.

5.1 User Variables

Shell variables are objects that the shell uses to store values. Unlike some other programming languages:

- Shell variables do not need to be “declared” prior to their first use
- All shell variables store character strings (the C language, for example, has integers, floating points, pointers, arrays, etc.)

5.1.1 Variable identifiers

An identifier is the name given to a shell variable (or, as you will see later, shell function). Valid identifiers obey these rules:

- Begin with a letter (A-Z, a-z) or the underscore (_) character
- Use only alphanumeric characters (A-Z, a-z, 0-9) or the underscore (_) character

The shell is case sensitive; these identifiers do not refer to the same variable:

Name1
name1
NAME1

Identifiers may contain virtually any number of characters; the shell treats each character significantly. By convention, user-defined variables do not use lower case letters.

The term *identifier* is often interchanged with *name*, however “name” is a jargon used by programmers.

Although identifiers may contain virtually any number of characters, in practice, they rarely exceed 20 characters in length.

Exercise: Identifiers

Identify which of these are valid identifiers for user variables and give reasons why the others are invalid.

MYVAR
 myvar12
 12MYVAR
 MYVAR 12
 myvar -12
 MYVAR_12
 myvar#12
 MY_VARIABLE_WHICH_IS_NUMBERED_TWELVE

5.1.2 Setting variables

To create or set a variable, use the syntax:

name=value

where:

name Must be a valid user identifier

value May be any character string

If *value* contains characters which are interpreted by the shell (such as * or space), then a quoting mechanism is needed. This is discussed in the next lesson.

■ Issues relating to quoting are covered in Lesson 6: "Quoting."

There can be no spaces adjacent to the equal sign (=).

A variable can also be set to contain a null string value by omitting *value*:

name=

Examples of setting variables include:

```
LPDIR=/usr/spool/lp
FLOPPY=/dev/rfd0135ds18
PATH=/bin:/usr/bin:.
LPDEST=laser5
EXINIT='set number autowrite showmode'
```

5.1.3 Unsetting variables

To unset or remove a variable, use the built-in **unset** command. The syntax is:

```
unset name
unset name1 name2 name3 ...
```

where *name* is a valid identifier.

Several variables can be unset from one command line.

Point out that there is no dollar sign (\$) before the variable name.

5.1.4 Accessing variables

To access the value of a shell variable on a command line, place a dollar sign (\$) before the identifier. For example:

```
cd $LPDIR
tar tvf $FLOPPY
echo $PATH
```

The shell expands the value to its string value *before* the command is executed.

A variable that is unset expands to a null string.

Emphasize that the shell expands a variable *before* the command is executed.

When doing demos, be careful not to get into a rut by using only the **echo** command in every example. Note that shell variables can be used with any shell command. Point out that they can even be assigned values that are shell commands themselves, as is common in many shell scripts. For example:

```
$ today=date
$ $today
```

```
Fri Jun 19 01:40:13 1998
```

5.1.5 `${name}` notation

Use the special notation `${name}` when a variable is used in a context where the identifier could be misinterpreted because it is immediately followed by a letter, digit, or underscore character:

For example:

```
$ MYVAR=word
$ echo MYVAR is $MYVAR
MYVAR is word
$ echo MYVAR is $MYVAR+
MYVAR is word+
$ echo MYVAR is $MYVARs
MYVAR is
$ echo MYVAR is ${MYVAR}s
MYVAR is words
```

Interactively demonstrate the commands shown in this section. Many students are fooled and expect `$MYVARs` to expand properly. Carefully explain why there is a problem.

Another use of the `${name}` syntax is to set variables either temporarily or permanently:

<code>\${var:-string}</code>	Temporarily assigns the new value <i>string</i> if <i>var</i> is not set or set to null
<code>\${var:+string}</code>	Temporarily assigns the new value <i>string</i> if <i>var</i> is <i>not</i> null
<code>\${var:=string}</code>	Permanently assigns the new value <i>string</i> if <i>var</i> is not set or set to null

These examples demonstrate how to check and change variables using this syntax.

To set `printer` temporarily to `postscript`:

```
$ echo $printer
[ null string ]
$ echo ${printer:-postscript}
postscript
$ echo $printer
[ null string ]
```

To set `printer` permanently to `postscript`:

```
$ echo ${printer:=postscript}
postscript
$ echo $printer
postscript
```

If the variable is already set, temporarily change printer to hpcl:

```
$ echo $printer
postscript
$ echo ${printer:+hpcl}
hpcl
$ echo $printer
postscript
```


AV 5-2

The `export` Command

```
$ FLOPPY=/dev/fd0135ds18
$ PRINTER=laser5
$ export FLOPPY

$ cat devicescript
#!/bin/sh
# Sample script to verify the effect of exported variables

echo The value of FLOPPY is $FLOPPY
echo The value of PRINTER is $PRINTER

# Rest of program which can use these variables can follow...

$ devicescript
The value of FLOPPY is /dev/fd0135ds18
The value of PRINTER is
```

nf0502

© The Santa Cruz Operation, Inc.

5.1.6 Exporting variables

A variable can be accessed only in the shell in which it is set unless the shell *exports* it to its children.

Use the built-in **export** command to mark a variable as being available to subsequently created child processes and their descendants. The syntax is:

```
export name
export name1 name2 name3 ...
```

Do not place a dollar sign (\$) before each *name*.

If no arguments are given, a list of all names that are exported *by this shell* is displayed.

The *current* value of all exported variables is copied to the environment of the new process when the shell creates it:

- It is unnecessary for a shell to export a variable again if it happens to alter its value; subsequently created processes will access the new value.
- If a shell alters a variable that it inherited from its parent, then this shell must export the variable if its child is to access the new value, otherwise, the child process will access its “grandparent” process value.

Variable information can pass from parent process to child process, but never from child process to parent process.

AV 5-2: The export Command

This slide shows a simple program that demonstrates what happens when a variable (namely, PRINTER) is not exported.

GuruNotes A variable can be set and exported on the same command line as another command. For example:

```
$ MYVAR=value command arg1 arg2 arg3
```

Here, the MYVAR variable is set and automatically exported for use while *command* executes. However, it is only available to the shell which executes *command* (and its children, if any); it will not be available to the current shell.

Note that a space separates the variable assignment from the rest of the command line. Using a semi-colon would nullify the exporting effects.

GuruNotes Show students this code:

```
unset MYVAR          # Delete MYVAR
MYVAR=somevalue      # Create a brand new variable
echo $MYVAR
```

Now ask them how it was possible for **echo** to display the value of MYVAR *considering that the value was not exported*. (The variable was unset just to be certain that we were dealing with a new, unexported variable.)

Students are likely to suggest the reason is because **echo** is built-in to the shell. While this is true, that has not always been the case (witness `/bin/echo` which *does* execute in a subshell, yet still works). Being built-in is not the reason **echo** works. Press for answers until you get the correct solution: **echo** never saw \$MYVAR on the command line. It was expanded to `somevalue` by the shell *before* **echo** even began to execute!

Indicate that exporting is needed when a command needs to get variable information from the environment rather than from the command line. As an example, **vi** needs to know the value of \$TERM, which it finds in the environment; it is not necessary to indicate the value on each **vi** command invocation line.

5.1.7 Readonly variables

To prevent the current shell from reassigning a value to a variable, or from unsetting one, use the built-in **readonly** command. The syntax is:

```
readonly name
readonly name1 name2 name3 ...
```

Do not place a dollar sign (\$) before each *name*.

If no arguments are given, a list of all names that are marked as read-only *by this shell* is displayed.

This command affects the current shell only. It does not affect child shells.

Although **readonly** is not often explicitly used in practice, the restricted shells, such as **rsh**, do not allow a user to change the value of the PATH variable.

5.1.8 Displaying variables

There are several ways to view the values of shell variables, including the **echo**, **set**, **env**, and **export** commands:

echo	The echo command displays what it sees on the command line to the standard output. Recall that variable substitution occurs before the command begins to execute.
set	The built-in set command can be used to set the options to the current shell. When given no command-line arguments, the set command displays the values of all variables that are available to the current shell (these values might not be available to its children).
env	The env command displays all variables that are in the current shell's environment. This includes only those variables which the current shell has exported itself or has inherited from its parent. The output describes all the values that the current shell's children will receive.
export	The built-in export command marks a variable as being available to subsequently created child processes and their descendents. When given no command-line arguments, export displays a list of all names (without their values) that <i>this shell</i> has exported.

AV 5-3

Shell Environment Variables

```
HOME=/usr/steve
PATH=/bin:/usr/bin:/usr/local/bin:$HOME/bin:.
CDPATH=./usr:/etc:$HOME:/
MAIL=/usr/spool/mail/steve
MAILPATH=/usr/spool/mail/uucp:/usr/spool/mail/cron
MAILCHECK=60
PS1='<root> '
PS2='> '
IFS=$IFS,
```

nf0503

© The Santa Cruz Operation, Inc.

5.2 Shell Environment Variables

AV 5-3: Shell Environment Variables

This slide shows some of the environment variables that are commonly set in the shell. These are usually set up in `/etc/profile` or `$HOME/.profile`, but the shell may have default values for some of them. These may be exported to allow child shells to access them.

Environment variables are used by many commands for various purposes. The shell uses these variables:

HOME	Your “home” directory; acts as a default argument for the cd command
PATH	The search path for commands; used for non-built-in commands that do not have a slash (/) in their name
CDPATH	The search path for the cd command: • Defines where cd should look for directories which will become the current directory • It is customary for CDPATH to begin the search in the current working directory (this does not pose a security risk as it does with the PATH variable).
MAIL	Identifies the path to your system mailbox: • The shell informs the user of the arrival of mail in the specified file. • Has no impact on where the mailing system deposits incoming mail
MAILPATH	A colon (:) separated list of mailbox pathnames: • Informs the user of the arrival of mail in any of the specified files (as well as the mailbox specified by MAIL, if any) • Each filename can be followed by % and a message that will be printed when the modification time changes. The default message is “you have mail.”

MAILCHECK Specifies (in seconds) how often the shell will check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** variables:

- Default value is 600 seconds (10 minutes)
- If set to 0, the shell will check before each prompt

PS1 Primary prompt string; default value is '\$ '

GuruNotes Although difficult to do in the Bourne shell, it is easy for the Korn shell to have **PS1** include the current directory path (use shell functions to emulate this in Bourne shell):

```
PS1='$PWD '
PS1='$PWD ! ' # Also includes command line number
```

GuruNotes In the C shell, the `prompt` variable is equivalent to **PS1**. It can also include the working directory and command line number, but an alias must be set up to do it:

```
alias cd 'cd \!:*; set prompt="\`pwd`\! "'
```

PS2 Secondary prompt string used for multi-line command lines; default value is '> '

IFS Internal field separators delimit the arguments when reading input (using the **read** built-in, for example). Default values are normally space, tab, and newline.

5.3 The read Command

In a shell script, you will often need to obtain information from the user interactively. This task is easily performed with the built-in **read** command.

5.3.1 Syntax

The general syntax is:

```
read name
read name1 name2 name3 ...
```

Do not place a dollar sign (\$) before each *name*.

The **read** command reads one line from the *stdin* and assigns values to the named user variables:

- Each word on the input line is delimited from the next by one or more IFS characters.
- The first word is assigned to *name1*, the second to *name2*, and so on.
- If there are more words than *names*, the leftover words are assigned to the last *name*.
- If there are fewer words than *names*, the leftover *names* are assigned a null string.

AV 5-4

The read Command

```
$ cat printit
#!/bin/sh
# Friendlier command to print a file to a given printer

echo 'Which file do you want to print? ==> \c'
read FILE DUMMY

echo Default printer information:
lpstat -d

echo To which printer do you want to send the file ${FILE}?
echo '==> \c'
read PRINTER DUMMY

echo Sending $FILE to $PRINTER
lp -d$PRINTER $FILE

$ printit
Which file do you want to print? ==> /etc/motd
Default printer information:
system default destination: curdev
To which printer do you want to send the file /etc/motd?
==> laser5
Sending /etc/motd to laser5
request id is laser5-20087 1 file
```

nf0504

© The Santa Cruz Operation, Inc.

5.3.2 Usage

AV 5-4: The read Command

This slide shows the general usage of the **read** command. Note the use of the DUMMY variable to force a one-word value to be placed into PRINTER.

Each **read** command in a script is commonly preceded by an **echo** statement which prompts the user for input.

Use **read** with a dummy variable to suspend script execution until the user presses the  key. For example:

```
read DUMMY
```

Use **read** with a dummy variable to ensure that a variable is assigned only one word value; extra input words, if any, are assigned to the dummy variable:

```
read ANSWER DUMMY
```

5.4 Positional Parameters

The shell reads and parses a command line before it executes it. Part of the parsing sequence includes separating the words on the command line. These words are known as *command-line arguments*.

The shell provides these command-line arguments to the new child process as part of its environment:

- Child shell scripts access the command-line arguments with special variables known as *positional parameters*.
- Child binary applications have similar mechanisms to access the command-line arguments.

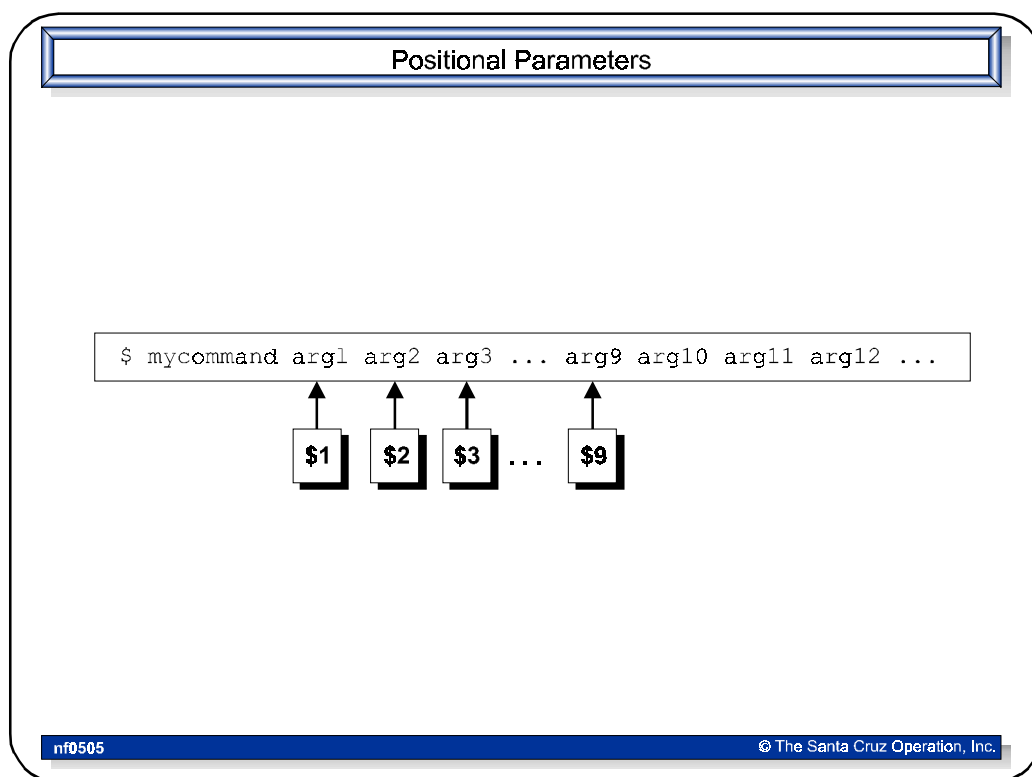
Command-line arguments provide useful information to a command. In this example, the command-line arguments tell **vi** to edit the files *report* and *memo*:

```
vi report memo
```

Here, the command-line arguments act as options and tell **shutdown** to run silently (**-y**) and allow no grace period before shutting down (**-g0**):

```
shutdown -y -g0
```


AV 5-5



Instructor Guide

5.4.1 Syntax

AV 5-5: Positional Parameters

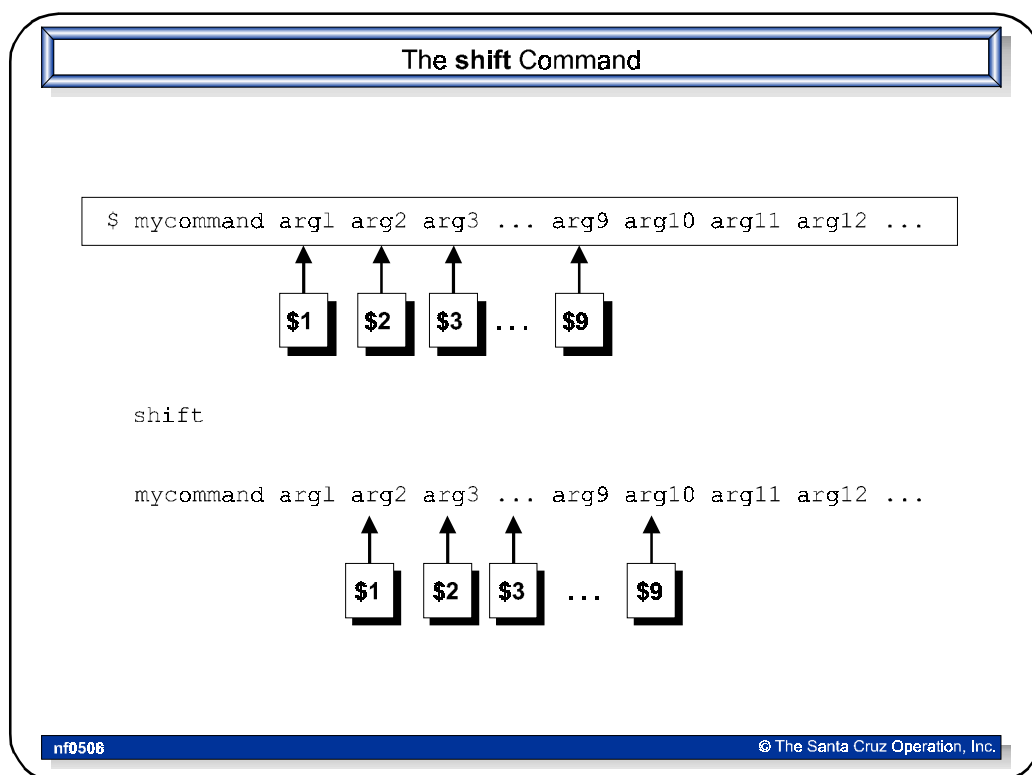
Use this slide to show that although a command line can have many arguments, there are only nine positional parameters.

The shell has nine positional parameters which it automatically sets when a script starts. These are \$1, \$2, \$3, ..., \$9. The first command-line argument is assigned to \$1, the second to \$2, and so on.

- The shell assigns only one command-line argument to a positional parameter.
- If there are fewer than nine command-line arguments, the remaining positional parameters are assigned null values.
- If there are more than nine command-line arguments, the shell can directly access only the first nine of them. This is because there are only nine positional parameters.

Other methods will allow access to the remaining parameters. These will be discussed shortly.

AV 5-6



5.4.2 The shift command

AV 5-6: The shift Command

Use this slide to emphasize:

- The built-in **shift** command reassigns the values of *all* positional parameters.
- Parameters cannot be re-shifted back to the left, but you can save the current parameter values in user-defined variables before shifting.

The built-in **shift** command reassigns the values of *all* positional parameters. The syntax is:

```
shift [n]
```

Without any arguments, or when $n = 1$, **shift** reassigns the value of \$2 to \$1, \$3 to \$2, and so on:

- Parameters are “shifted down” by one position.
- Additional values beyond \$9 are shifted as well.
- If there is no value to shift into a positional parameter, it is assigned a null value.

When $n > 1$, parameters are shifted down by n positions:

- For example, **shift 2** reassigns the value of \$3 to \$1, \$4 to \$2, and so on.
- Shifting further than there are values will result in null values for all parameters. The shell will also display an error message (**cannot shift**) when this happens.

Parameters cannot be re-shifted back to the left, but you can save the current parameter values in user-defined variables before shifting. For example:

```
OLDPARAM1=$1
```

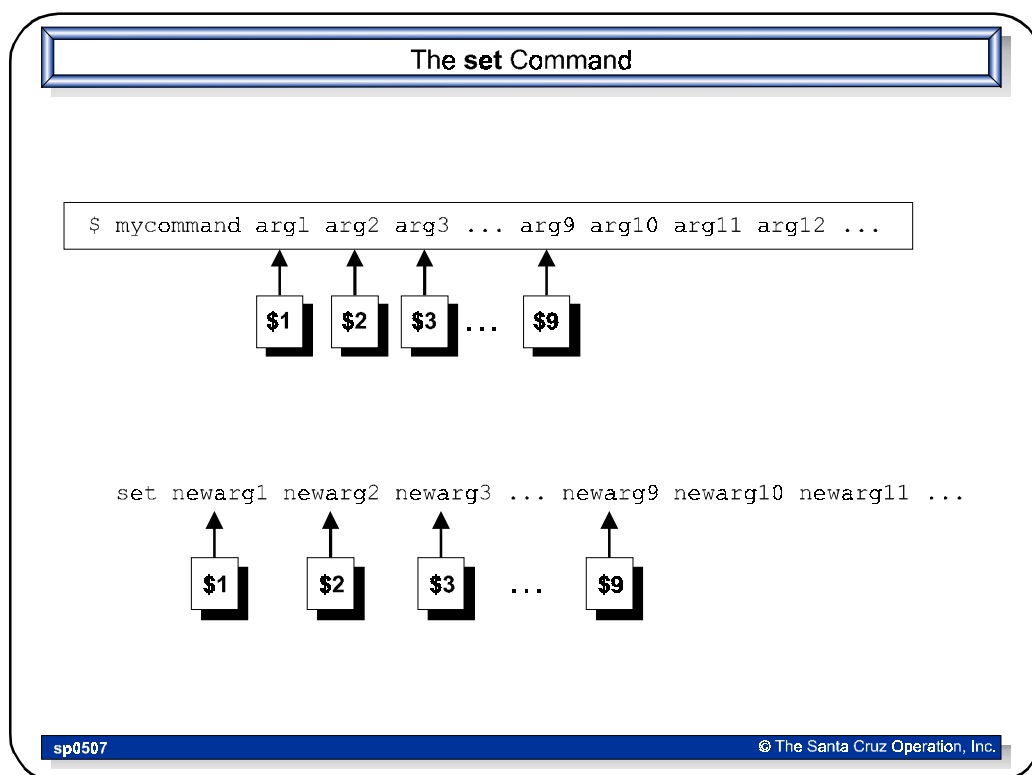
The **shift** command is useful for accessing more than nine command-line arguments, but:

- Once shifted, the original value of \$1 is lost (\$2 and more, if $n > 1$) unless saved in a user variable.
- The shell cannot **shift** upwards.

The **shift** command is also useful in looping constructs (discussed in a later lesson) because it allows you to access each command-line argument as **\$1**. For example, the following code, when repeated many times (as in a loop), accesses each command-line argument as **\$1**:

```
echo the next parameter is $1
shift
```


AV 5-7



5.4.3 The set command

AV 5-7: The set Command

Use this slide to show that although a script may start out with command-line argument values in the positional parameters, once **set** executes, *all* of these values change.

Two purposes of the **set** command have already been discussed in this course:

- It sets flags for the current shell (for example, **-v** and **-x**).
- When used without arguments, it displays the variables in the current shell.

The **set** command has another use related to positional parameters.

Recall that positional parameters are automatically set by the shell; you cannot directly assign their values with statements such as:

```
$1=newvalue      # This is wrong!  
1=newvalue       # This is wrong, too!
```

However, you can use the **set** command to assign values to *all* positional parameters in the current shell. The syntax is:

```
set arg1 arg2 arg3 ...
```

where:

- *arg1* is assigned to **\$1**, *arg2* to **\$2**, and so on.
- You can specify more than nine arguments, but will need to **shift** to access the extra arguments.

Use **set** to establish default values for positional parameters if a user invokes a script without command-line arguments.

The **set** command changes the value of *all* positional parameters. To change only **\$3**, leaving **\$1** and **\$2** intact, enter:

```
set $1 $2 newvalue
```

This assumes **\$4** and beyond are not needed because they are reset as well.

Use the “- -” option if the new value for **\$1** contains a leading dash character. Otherwise the shell misinterprets your new **\$1** value as being an option itself.

For example, to set \$1 to contain the string value “-v”:

```
set -v      # This is wrong! It sets the -v option
set -- -v   # This is correct! It sets $1 to "-v"
```


AV 5-8

Special Shell Variables

<code>\$0</code>	Command name
<code>\$#</code>	Number of positional parameters
<code>\$*</code>	All positional parameters
<code>@</code>	All positional parameters
<code>?</code>	Exit status of last foreground command
<code>\$\$</code>	PID of current shell
<code>!</code>	PID of last background process
<code>-</code>	Flags set in the current shell

sp0508

© The Santa Cruz Operation, Inc.

5.5 Special Shell Variables

AV 5-8: Special Shell Variables

This slide summarizes all of the special shell variables.

The shell has other special variables which it automatically sets. Like the positional parameters, you cannot directly assign a value to these variables.

\$0 This variable contains the name of the shell script command. It is useful for:

- Identifying the name of the script in error messages. For example:

`echo $0: cannot open file >&2`
- Providing a script with two or more program names (via hard links) where each program performs slightly differently. The script can see how it was invoked to identify what it will do differently.

Note that \$0 is not a positional parameter. The positional parameters are \$1 through \$9.

\$# This variable contains the number of positional parameters (can be greater than nine):

- The value is affected by the **shift** and **set** commands.
- It is useful for determining whether a user invokes the script with sufficient command-line arguments.

\$* This variable contains all positional parameters (including those beyond \$9).

\$@ This variable contains all positional parameters (including those beyond \$9).

The difference between \$* and \$@ is discussed in Lesson 7: "Decision Constructs."

\$? This variable contains the exit status of the last command executed in the foreground. By convention, commands exit with zero upon successful completion and with a non-zero value for unsuccessful completion. Exit codes are discussed in a later lesson.

\$\$ This variable contains the PID of the current shell. It is useful for creating temporary files with unique names:

```
sort datafile > /tmp/data$$$.tmp
```

\$! This variable contains the PID of the last command that executed in the background. It is useful when you want to **wait** for the last command invoked in the background to finish:

```
wait $!
```

It is also useful if you need to **kill** the last command invoked in the background:

```
kill $!
```

\$- This variable contains flags supplied to the shell when it was invoked or by the **set** command.

5.6 Lesson Summary

Shell variables provide scripts—as well as interactive command lines—with extra flexibility by storing string values for later use. Users can create and manipulate their own variables or access the special parameters which the shell automatically sets up.

Review Questions

1. What is an identifier? What constitutes a valid one?
2. What is the purpose of the **IFS** shell variable?
3. When is it necessary to export a variable?
4. How does the **read** command work when more input words are provided than variables? With fewer input words?
5. What are positional parameters? Which shell commands can affect their values?
6. What are three uses for the **set** command?
7. What are some practical uses of the **\$0**, **\$#**, and **\$\$** special variables?

5.7 Lab: Using Shell Variables

This lab gives you experience using shell variables in your login environment, on interactive command lines, and in your shell programs.

5.7.1 Login environment

Set or update these variables in your *.profile*:

- CDPATH
- MAILCHECK
- PS1

5.7.2 Phone database

Write two programs that maintain a small telephone database. These programs will either add or display an entry in the database.

5.7.2.1 Add phone book entries

Write a program named **add** that adds a new entry to the telephone database by prompting the user for:

- Name
- Department
- Phone extension
- Login name

This information should be added to the text file *\$HOME/phone.data* as a single line with tab-separated fields. For example:

```
Andrew Tonkin    Tech Support    8649    andyt
```

After each entry, use the **add** program to sort the telephone database.

Execute and test your script.

5.7.2.2 Display phone book entries

Write a program named **ext** that searches the telephone database and displays the line(s) that match a pattern provided by the user.

Prompt the user for a pattern (which can be the name, department, extension, or login name) and then perform the search. Execute and test your script.

5.7.2.3 Advanced options

Modify both scripts so that needed information is not prompted, but rather comes in from the command line as positional parameters. Again, execute and test your script.

5.7.3 A friendly find

Write a program named **search**, a user-friendly **find** command, that allows users to enter:

```
search lostfile at start
```

instead of the more cumbersome command line:

```
find start -name lostfile -print
```

Execute and test your script.

5.7.3.1 Advanced options

Modify the script to prompt for needed information, rather than coming in from the command line. Again, execute and test your script.

5.8 Relevant Commands

env	Displays all variables that are in the current shell's environment
export	Marks a variable as being available to subsequently created child processes and their descendents; displays those variables which the current shell has marked for export
read	Reads one line from the standard input and assigns values to user variables
readonly	Prevents a shell variable from being reassigned or unset
set	Sets flags and positional parameters in the current shell; displays the values of all variables that are available to the current shell
shift	Reassigns the value of \$2 to \$1, \$3 to \$2, and so on; or shifts by <i>n</i> positions
unset	Removes a shell variable

5.9 Answers to Review Questions

1. An identifier is a name that specifies a given shell variable or function. A valid identifier is composed of only alpha-numeric and underscore (`_`) characters, and does not begin with a digit. It may contain any number of significant characters.
2. The IFS shell variable is the internal field separator. The shell uses the characters specified by this variable to separate the individual arguments when reading input (such as with a `read` command).
3. A variable must be exported when its value needs to be accessed by a child process.
4. When more input words are provided than variables, the `read` command assigns the extra input words to the last variable. When fewer input words are provided than variables, `read` assigns a null string to the extra variables.
5. Positional parameters are nine special shell variables that the shell automatically sets to the values of the command line arguments. The `set` and `shift` commands can affect their values.
6. The `set` command sets options (flags) for the current shell, such as `-v` or `-x`; displays the variables (and shell functions) set in the current shell (when given no command line options); and resets the positional parameters for the current shell.
7. The `$0` variable refers to the command name of the current shell. It is frequently used in error messages to identify the command. It can also be used to determine program functionality for a given invocation. For example, one name might indicate that certain command-line options should be in effect; another name doesn't. The `$#` variable refers to the number of positional parameters. It can be used to ensure that a command is invoked with the expected number of command-line arguments. Default values can be assumed if the number does not agree, or error actions can be generated. The `$$` variable refers to the PID of the current shell process. It is commonly used to generate a unique filename that can be used for transient purposes.

6

Quoting

Instructor Guide

Instructor Guide

Lesson 6

Quoting

Suggested lesson duration

This lesson should take approximately 2.25 hours to complete.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 6-1

Instructor Guide

Quoting

- Identify shell metacharacters
- Use quoting mechanisms
- Use the shell command substitution mechanism
- Describe the shell command line parsing sequence
- Use the **eval** command to reparse a line

sp0601© The Santa Cruz Operation, Inc.

Introduction

The shell uses many characters to denote some special meaning. For example:

- `*` means substitute zero or more characters in a filename
- `>` means that command output is to be redirected to a file
- `$` means to substitute the contents of a variable

You might have occasions when entering a command line either interactively or in a shell script where you do *not* want that special meaning. Fortunately, the shell provides a way—in fact, three ways—to neutralize the special meaning: single quotes (`'`), double quotes (`"`), and the backslash (`\`).

This lesson discusses the differences between these three mechanisms and another quote-look-alike character: the backquote (```), which is used for command substitution.

Objectives

This lesson describes the shell's quoting and command-substitution mechanisms. By the end of this lesson, you will be able to:

- Identify shell metacharacters
- Use quoting mechanisms to neutralize the special meaning of shell metacharacters
- Use the shell command-substitution mechanism
- Describe the shell command-line parsing sequence
- Use the **eval** command to reparse a line

AV 6-1: Quoting

The first slide describes the overall topics covered in this lesson.

AV 6-2

Bourne Shell Metacharacters

#	Comment
\$	Shell variable
and ^	Pipe
< and >	I/O redirection
&	Background execution
*, ?, and []	Filename wildcards
() and { }	Command grouping
' , " , and \	Quoting characters
`	Command substitution
Tab and Space	Internal field separator
Enter	Internal field separator and command terminator
;	Command terminator

sp0602

© The Santa Cruz Operation, Inc.

6.1 Shell Metacharacters

AV 6-2: Shell Metacharacters

Review the characters on this slide and their special meanings. Students should be familiar with metacharacters, except for the quote characters.

A shell metacharacter is a character the shell interprets in some special way. This list summarizes each of the shell metacharacters, some of which are discussed later in this course:

#	Comment
\$	Shell variable
and ^	I/O redirection
< and >	&
*, ?, and []	Filename wildcards
() and { }	Command grouping
', ", and \	Quoting characters
'	Command substitution
tab and space	Field separator
newline	Field separator and command terminator
;	Command terminator

Note that dot (.) is not a shell metacharacter. You need not quote the dot on a command line to prevent the shell from specially interpreting it.

The slash character (/) is also not a shell metacharacter. It is, however, a *kernel* metacharacter; the kernel uses it when evaluating pathnames. Again, you need not quote the slash on a command line to prevent the shell from specially interpreting it.

AV 6-3

Single Quotes

```
$ ls
mailfolder
memo
report
$ find . -name m* -print
find: missing conjunction
$ find . -name 'm*' -print
./memo
./mailfolder

$ echo Home is $HOME
Home is /usr/steve
$ echo 'Home is $HOME'
Home is $HOME

$ PS1=<steve>
syntax error: `newline or ;' unexpected
$ PS1='<steve> '
<steve>
<steve> echo one      two
one two
<steve> echo 'one      two'
one      two
```

sp0603

© The Santa Cruz Operation, Inc.

6.2 Quoting Mechanisms

On some command lines, you might want to use a shell metacharacter as a literal character that possesses no special meaning to the shell. The shell has three mechanisms to neutralize that special meaning:

- Single quotes ('... ')
- Double quotes ("... ")
- Backslash (\)

6.2.1 Single quotes

Single quotes are used in pairs to remove the special meaning of all enclosed shell metacharacters with the exception of other single quote characters.

Of all the quoting mechanisms, students should find single quotes to be the easiest to understand because they neutralize everything (except other single quotes).

AV 6-3: Single Quotes

This slide shows several sets of command lines where the first of each set attempts a task and fails because quotes were not provided. The second of each set does it properly. Ask students what went wrong with the first command line. Interactively execute each command line with the shell's `-x` option in effect to see exactly what is on the command line when the command starts.

AV 6-4

Double Quotes

```
$ find . -name "m*" -print
./memo
./mailfolder

$ echo Home is $HOME
Home is /usr/steve
$ echo 'Home is $HOME'
Home is $HOME
$ echo "Home is $HOME"
Home is /usr/steve

$ VAR='one      two'
$ echo VAR is $VAR
VAR is one two
$ echo "$VAR"
VAR is one      two

$ VAR='two
> lines'
$ echo VAR is $VAR
VAR is two lines
$ echo "VAR is $VAR"
VAR is two
lines
```

sp0004

© The Santa Cruz Operation, Inc.

6.2.2 Double quotes

Double quotes are used in pairs to remove the special meaning of all enclosed shell metacharacters with the exception of:

- Other double quote characters
- Dollar (\$)
- Backslash (\)
- Backquote (`)

There are special circumstances for the backslash character which is discussed shortly.

These four characters retain their special meaning when enclosed in double quotes. This means that inside double quotes, you can still:

- Expand variables
- Protect characters from shell interpretation
- Perform command substitution

It is frequently a good idea to enclose a variable within double quotes on a command line so that its value does not get interpreted by the shell.

AV 6-4: Double Quotes

This slide shows similar command lines that use no quotes, single quotes, and double quotes:

- The first example uses double quotes, but single quotes could have been used instead.
- The next example shows that \$ is expanded in double quotes, but not in single quotes.
- The next two examples show that it is usually a good idea to use variables within double quotes to protect any special characters that are expanded. In both of these examples, the special characters were space and newline, both of which are field separators.

AV 6-5

Backslash

```

$ find . -name m'*' -print
./memo
./mailfolder
$ find . -name m\* -print
./memo
./mailfolder

$ echo two'
> 'words
two
words
$ echo two\
> words
twowords

$ TAPE=/dev/rct0
$ echo "TAPE is \$TAPE"
TAPE is $TAPE

$ echo "Backslash: \\"
Backslash: \
$ echo "Double Quote: \""
Double Quote: "
$ echo "Asterisk: \*"
Asterisk: \*
$ echo "Percent: \% "
Percent: \%

```

sp0605

© The Santa Cruz Operation, Inc.

AV 6-6

Quoting Exceptions

Quote Mechanism	Metacharacters not affected
Single quotes ('..')	Other single quotes
Double quotes ("...")	Other double quotes
	Dollar (\$)
	Backslash (\)
	Backquote (`)
Backslash (\)	Enter (ENTER)

sp0606

© The Santa Cruz Operation, Inc.

6.2.3 Backslash

The backslash (\) removes the special meaning of the one shell metacharacter which follows it. It is nearly identical to placing single quotes around a single character.

The exception is for the newline (or “Return”) character:

- Single quotes around a newline metacharacter tells the shell to treat it as a literal character.
- A backslash before newline tells the shell to ignore the character so that the line can be continued onto the next line.

The backslash retains its meaning inside double quote characters *if the character that follows (still) has special meaning and can be neutralized*; otherwise, the double quotes also neutralize the backslash character.

AV 6-5: Backslash

This slide shows the use of the backslash character with other quotes.

- The first example is very straight forward—the * is neutralized with either quotes (double quotes would work, too) or the backslash.
- The second example shows the one distinction between placing single quotes around a single character and using a backslash instead:
 - Single quotes around `[ENTER]` tells the shell to treat `[ENTER]` as a literal character.
 - A backslash before `[ENTER]` tells the shell to ignore the character so that the line may be continued onto the next line.

Using a backslash inside double quotes is trickier. The next examples on the slide address this usage:

- The third example shows that the backslash could neutralize the \$ because it otherwise has meaning inside double quotes.
- The fourth example shows that the first backslash could neutralize the second one because it otherwise has meaning inside double quotes.
- The next example shows that the backslash could neutralize the double quote (") because it otherwise has meaning inside double quotes.
- The last two examples show that the backslash had nothing to neutralize, so the shell left it alone. The asterisk (*) was already neutralized by being inside the double quotes; the percent (%) never had any special meaning even outside the double quotes.

AV 6-6: Quoting Exceptions

This slide, summarizing the metacharacters not affected by quoting mechanisms, might help students with this exercise.

Exercise: Using quotes

Use one **echo** command to display this message to the *stdout*:

John asked, "What's A:\setup.exe on this disk?"

Command Substitution

```
$ echo "The date is:    <\date\>"
The date is:    <Wed Oct 05 10:26:43
PDT 1994>

$ CWD=\pwd\
$ cd /some/directory

$ cd $CWD

$ cat editnames
memo
report
letter
$ vi \cat editnames\

$ GREETING=\banner hello\
$ echo $GREETING
# # ##### # # ##### # # # # # # ##### ##### # # # # # # # # # # # # # #
# # # ##### # # # # # # # # # # # # # #
$ echo "$GREETING"

# # ##### # # # #####
# # # # # # # # # # # # # #
##### ##### # # # # # #
# # # # # # # # # # # # # #
# # # # # # # # # # # # # #
# # ##### # # # # # # # # # #
```

sp0007

© The Santa Cruz Operation, Inc.

6.3 Command Substitution

Another shell feature exists which uses a different quote character: the backquote (```)—also called “grave” [pronounced *gräv*] or “backtick.” This mechanism is unrelated to the other shell quoting features which neutralize the special meaning of a shell metacharacter. Backquotes provide for *command substitution*, which causes output from a command to be sent to a command line.

6.3.1 Syntax

Backquotes are used in pairs (`` ... ``) and cause the enclosed command list to be executed in a subshell:

- The command list may be a single command, several commands separated by semicolons (`;`), or a pipeline.
- The *stdout* of that command list is substituted onto the current command line in place of the backquoted section.
- The *stdin* and *stderr* of that command list are unaffected by command substitution.

Because they do not neutralize backquotes, command substitution can occur within double quotes.

Emphasize these points:

- The *stdout* of the command list is substituted.
- The *stdin* and *stderr* of that command list are unaffected by command substitution.

AV 6-7: Command Substitution

This slide shows several examples of using command substitution on a command line. Note that this substitution can occur within double quotes and it might be a good idea to do so (as shown in the first example).

In the last example, note again the benefits of expanding variables within double quotes to protect their contents (here, lots of extra spaces).

6.3.2 The line command

The **line** command is a filter which reads one line from the *stdin* and writes it to the *stdout*.

It is commonly used with command substitution to assign values to user variables and positional parameters. For example:

```
MYVAR=`line`  
set `line`
```

■ Review that a *filter* is a command which reads from the *stdin* and writes to the *stdout*.

6.4 Command-Line Parsing Sequence

The shell goes through a sequence of steps when it reads, parses, and executes a command line. In order, these steps include:

1. Read the command line.

During this step, the shell also performs a preliminary syntax check and recognizes (but may defer processing of) symbols for quoting, I/O redirection, conditional execution, background execution, and delimiters.

If `-v` is set, the command line is echoed to the *stdout*.

2. Perform command substitution.

This refers to a command list enclosed within backquotes. Note that this command list is subject to shell command-line evaluation of its own, so it can use any of the shell's special features.

3. Perform variable and parameter substitution.

This step expands user-defined variables as well as positional parameters and special shell variables.

4. Perform filename metacharacter expansion.

This expands the filename wildcards into appropriate filenames. This can affect the values that will be assigned to the positional parameters if a given wildcard matches multiple filenames.

The command line prints with substitutions if the `-x` flag is set.

5. Set up I/O redirection.

In this step, the shell sets up the file descriptor table for the process.

6. Break up command line into arguments.

This step breaks the command line into individual arguments which will be assigned to the positional parameters.

7. Perform environment processing

In this step, environment variables are assigned values and the command is located, using the `$PATH` variable, if necessary (recall Section 3.2.2, “Finding commands,” on page 3-15).

8. Execute the command.

Lastly, the command begins to run according to the process described in Section 3.2.3, “Executing commands,” on page 3-19.

Again, stress that the shell does a lot of processing to a command line before the command even begins to execute. What you enter as a command line and what executes can be very different.

Parsing a Command Line (1)

```
$ set -x
$ echo This is `pwd` : * > $HOME/$$.tmp
+pwd
+echo This is /home/gen : README README.general
fixup maillist messagel setup title
$
$ if $HOME/*.tmp
/home/chris/1071.tmp
$
$ cat /home/chris/1071.tmp
This is /home/gen : README README.general
fixup maillist messagel setup title
```

sp0608

© The Santa Cruz Operation, Inc.

6.4.1 Examples

AV 6-8: Parsing a Command Line (1)

This slide shows an example of how a command line is parsed. Knowing this sequence gives students a better understanding of why the **eval** command is sometimes needed. It might be helpful to demonstrate these examples using the Korn shell with the **-x** option set. The expansion of environment and special variables is shown, as is redirection.

The example in AV 6-8 creates a file in the users home directory using the process ID as part of the file name. Setting the **-x** option first shows how the command is parsed:

- | | |
|--|---|
| (1) Read the command line | The shell sees the echo command and recognizes the symbols for redirection, substitution, etc. The -v flag causes the command to be echoed to the <i>stdout</i> . |
| (2) Perform command substitution | <code>`pwd`</code> is executed and expanded to <i>/etc/default</i> . Note that the parsing sequence is recursive—it is again started to parse the pwd command. |
| (3) Perform variable/parameter substitution | <code>\$\$</code> is expanded to 838 (the PID of the echo command). <code>\$HOME</code> is assigned to <i>/home/chris</i> . |
| (4) Perform filename metacharacter expansion | <code>*</code> is expanded to the list of files in the current directory

The command line prints with substitutions because the -x flag is set. |
| (5) Set up I/O redirection | <i>stdout</i> is redirected to the file descriptor for the file <i>838.tmp</i> . |
| (6) Parse the command line | The command line is broken into arguments. |
| (7) Perform environment processing | echo is located (determined to be a shell built-in). |
| (8) Execute the command | echo is executed. |

AV 6-9

Parsing a Command Line (2)

```
$ set -x
$ echo `id | sed 's/^[a-z0-9=].*//'` > userid
+sed 's/^[a-z0-9=].*//'
+id
+echo uid=0

$ cat userid
uid=0
```

sp0600

© The Santa Cruz Operation, Inc.

AV 6-9: Parsing a Command Line (2)

This slide shows another example of how a command line is parsed. This is built upon in the next section to help demonstrate why the **eval** command is sometimes needed.

The example in AV 6-9 echoes the user ID to the file *userids*. It uses the **id** command which produces the following type of output:

```
uid=0(root)gid=3(sys)groups=0(root),1(other),2(bin),3(sys),
4(adm),5(uucp),6(mail),7(tty),8(audit),10(nuucp),12(daemon),
23(cron),25(dtadmin),47(priv),9(lp)
```

To extract only the user ID information, **sed** is used to remove everything but the string `uid=0`.

The command executes as follows:

It is not important for the students to understand how **sed** works at this point. For now, suffice to say that this **sed** command does what is intended. The **sed** command is covered in Section C.4, "Editing Files with sed," on page C-43.

- | | |
|--|---|
| (1) Read the command line | The shell sees the echo command and recognizes the symbols for redirection, substitution, etc. The -v flag causes the command to be echoed to the <i>stdout</i> . |
| (2) Perform command substitution | <code>`sed   id...`</code> is parsed, executed, and substituted. The result is the string <code>uid=0</code> (when run by root). |
| (3) Perform variable/parameter substitution | There are no variables used in the echo command line. |
| (4) Perform filename metacharacter expansion | There are no metacharacters to expand (in the echo command line). The command line prints because the -x flag is set. |
| (5) Set up I/O redirection | <i>stdout</i> is redirected to the file descriptor for the file <i>userid</i> . |
| (6) Parse the command line | The command line is broken into arguments. |
| (7) Perform environment processing | echo is located (determined to be a shell built-in). |

(8) Execute the command **echo** is executed.

Similar code might be used in a shell script as a security feature. Some scripts should only be run by the **root** user with UID 0. Setting up a variable containing the user ID value allows a program to verify the user has the proper ID before executing certain commands or portions of the program. An example is provided in Section 6.5.2.1, “Setting a variable using eval,” on page 6-35.

6.5 The eval Command

On occasion, a command line might need to be parsed in a different sequence. Although you cannot cause the shell to use a different parsing order, you can use the built-in **eval** command to cause the command line to be parsed a second time, thereby hitting a sequence step that might have been missed from the first parsing. This is generally necessary when:

- Shell variable expansions include a shell metacharacter which needs to be interpreted
- You intend to execute the output of a command using command substitution

6.5.1 eval syntax

Place the command name **eval** in front of any command line:

`eval command-line`

The command line is parsed twice:

- The first parsing is performed by the shell as it does for any command line.
- The second parsing is performed by **eval**. Because it is a shell built-in command, it performs the same parsing sequence as the shell.

Note: Make sure special characters are properly protected when using **eval**; the first parsing will remove quotes.

AV 6-10

The eval Command

```

$ PIPE='|'
$ ls $PIPE wc
| not found
wc not found
$ eval ls $PIPE wc
      4      4      29

$ tset -s
export TERM;
TERM=ansi;
$ `tset -s`
TERM;: is not an identifier
$ eval `tset -s`
$

```

sp0610

© The Santa Cruz Operation, Inc.

AV 6-11

Coding a Shell Escape

```

$ cat shellescape1
#!/bin/sh
# Sample code to demonstrate how to do a "shell escape"
...
echo "Command? ==> \c"
`line`
...

$ shellescape1
Command? ==> ls -l > logfile
> not found
logfile not found

$ cat shellescape2
#!/bin/sh
# Even better sample code to demonstrate how to do a "shell escape"
...
echo "Command? ==> \c"
eval `line`
...

$ shellescape2
Command? ==> ls -l > logfile
$

```

sp0611

© The Santa Cruz Operation, Inc.

6.5.2 eval Examples

AV 6-10: The eval Command

This slide shows examples of command lines that need to use **eval**

The first example in AV 6-10 initially fails because the pipe metacharacter appeared on the command line too late (Step 2) for the shell to recognize that I/O redirection was supposed to take place. The shell recognizes I/O redirection symbols in Step 1, although it defers setting up the redirection until Step 4.

The second example in AV 6-10 initially fails because the semi-colon appeared on the command line too late (Step 3) to recognize it as a command delimiter (Step 1).//re entered. The second example, **shellescape2**, uses **eval** to parse command lines properly with metacharacters.

6.5.2.1 Coding a shell escape using eval

AV 6-11: Coding a Shell Escape

This slide shows how to use **line** with **eval** and command substitution.

The first example program in AV 6-11, **shellescape1**, works when the user enters simple command lines that do not contain shell metacharacters. It fails when metacharacters are entered. The second example, **shellescape2**, uses **eval** to parse command lines properly with metacharacters.

AV 6-12

Setting the uid Variable (1)

```
$ cat auth-check
uid=""
`id | sed 's/[^a-z0-9=].*//'`
if [ "$uid" = 0 ]
then
    echo You are root; ok to run $0
else
    echo You are not root; cannot run $0
fi

$ sh -x auth-check
uid=
+ sed 's/[^a-z0-9=].*//'
+ id
+ uid=0
UX: sh (u) ERROR: uid=0: Not found
+ [ = 0 ]
+ echo You are not root; cannot run auth-check
You are not root; cannot run auth-check
```

sp0612

© The Santa Cruz Operation, Inc.

AV 6-13

Setting the uid Variable (2)

```
$ cat auth-check1
uid=""
eval `id | sed 's/[^a-z0-9=].*//'`
if [ "$uid" = 0 ]
then
    echo You are root; ok to run $0
else
    echo You are not root; cannot run $0
fi

$ sh -x auth-check1
uid=
+ sed 's/[^a-z0-9=].*//'
+ id
+ eval uid=0
uid=0
+ [ 0 = 0 ]
+ echo You are root; ok to run auth-check1
You are root; ok to run auth-check1
```

sp0613

© The Santa Cruz Operation, Inc.

6.5.2.2 Setting a variable using eval

AV 6-12: Setting the uid Variable (1)

This slide builds on the example in AV 6-9 on page 6-26. It demonstrates the need for **eval** to set a variable using the result of another command (**id**).

AV 6-13: Setting the uid Variable (2)

This slide builds on the example in AV 6-9 on page 6-26. It uses a similar command line with **eval** to set a variable, `uid`, which can be used to test authorization.

The example program fragment in AV 6-12 does not set the `uid` variable to 0 as might be expected when run the program is run by **root**. Because the second line of the program fragment (`id | sed ...``) is only parsed once, the shell tries to execute the string `uid=0` as a command.

The sample program fragment in AV 6-13 corrects the problem using **eval**.

6.6 Lesson Summary

Shell metacharacters tell the shell to provide a variety of services, but sometimes the service is not wanted. The shell permits you to use a metacharacter as a literal character by neutralizing it with a quoting mechanism. The three quoting mechanisms are:

Single quotes (')	Neutralize all metacharacters except other single quote characters
Double quotes (")	Neutralize all metacharacters except other double quote characters, dollar (\$), backslash (\), and backquote (`)
Backslash (\)	Neutralizes the one metacharacter that follows

The shell also has a command substitution mechanism that uses backquotes (`) to cause the *stdout* of a command to be substituted onto a command line.

Review Questions

1. What is a shell metacharacter? Is the slash (/) character one? What is its meaning? Who interprets it?
2. How does the shell interpret single quotes differently than double quotes?
3. How does a backslash work when it is surrounded by double quotes?
4. Placing single quotes around a single character is nearly the same as placing one backslash in front of it instead. When is this not the case?
5. How does command substitution work?

6. Which command is more efficient at assigning a value to a user variable? Why?

```
read MYVAR  
MYVAR=`line`
```

7. When should the **eval** command be used?

6.7 Lab: Quoting

This lab gives you experience with the various shell quoting mechanisms as well as using command substitution.

6.7.1 Command-line prompt

Modify your command-line prompt so that upon login, it displays your machine name and login name in this format:

```
machine|loginname
```

Use command substitution to determine the names rather than hard-coding particular names. This allows you to use the same command line for different users on different machines. There should be a space character following the prompt.

6.7.2 Phone database: revisited

Modify the Phone Database scripts from the last lesson to prompt for the needed information. This will differ from the original design. This time, do not store the information in user-defined variables; instead, use positional parameters. Execute and test your script.

6.7.3 Shell escape

Write a shell program (which can be incorporated into another lab exercise later in the course) that prompts the user for a command line and then executes it. Your script should handle user input such as:

```
ps -e | more
who > logfile
echo $TERM
```

6.7.4 Recursive grep

Write a script called **grepall** that searches for a pattern in all files in a given directory and also in all subdirectories from that point on down the directory tree structure. The output should include the matching line(s) in the file as well as the filename itself. The **find** and **grep** commands might be useful to accomplish this exercise.

6.8 Relevant Commands

The following commands and files were presented in this lesson.

line	Reads one line from the <i>stdin</i> and writes it to the <i>stdout</i>
eval	Causes a command line to be parsed twice by the shell

6.9 Answers to Review Questions

1. A shell metacharacter is a character that has special meaning to the shell (i.e., the shell specially interprets the character). Some shell metacharacters include `*`, `?`, `<`, `>`, and space. The slash (`/`) character is not a shell metacharacter. The kernel interprets it as meaning that another directory is to be searched in a pathname.
2. Single quotes remove the special meaning of all shell metacharacters except for other single quote characters. Double quotes remove the special meaning of all shell metacharacters except for other double quote characters, dollar (`$`), backquote (```), and backslash (`\`).
3. When surrounded by double quotes, a backslash (`\`) removes the special meaning of the character which follows if it is a: dollar (`$`), backquote (```), double quote (`"`), or backslash (`\`). If a different character follows, that character would have already been neutralized by the double quotes; therefore, the backslash is also neutralized by the double quotes.
4. Placing single quotes around the newline character (or another single quote) is not the same as placing one backslash in front of it instead. Single quotes around it removes the special meaning (command termination); a backslash causes the shell to ignore the newline for line continuation purposes.
5. Command substitution occurs when a command is enclosed within backquote characters (```). The command executes in a subshell and its standard output is substituted onto the current command line in place of the quoted command.
6. The `read` command is more efficient than `line` at assigning a value to a user variable because it is a shell built-in command and does not require a process to be created before it can run.
7. The **`eval`** command should be used when a command needs to be parsed again because a command-line parsing step has already occurred by the time certain command-line substitutions take place. Its use is recommended when you want to execute the output of a command using command substitution.

7

Decision Constructs

Instructor Guide

Instructor Guide

Lesson 7

Decision Constructs

Suggested lesson duration

This lesson should take approximately 3 hours to complete, including the lab exercises.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 7-1

Instructor Guide

Decision Constructs

- Define the conventional successful and unsuccessful error codes
- Use the **exit** command to terminate or abort a shell script
- Use the **test** command
- Use the **if** statement to conditionally execute command lines
- Use the **case** statement to conditionally execute command lines
- Use special shell constructs for conditional command execution

sp0701© The Santa Cruz Operation, Inc.

Introduction

A characteristic of programming languages is the ability to branch or skip over lines of code when certain conditions are present; the shell programming language possesses this feature as well. The Bourne shell uses these statements to implement decision control flow:

- **if / then / fi**
- **if / then / else / fi**
- **if / then / elif / fi**
- **case / esac**

This lesson describes the nature of the conditions upon which decisions are based as well as the use of the shell's decision-making constructs.

Objectives

This lesson describes the decision-making constructs of the Bourne shell. By the end of this lesson, you will be able to:

- Define the conventional successful and unsuccessful error codes
- Use the **exit** command to terminate or abort a shell script
- Use the **test** command
- Use the **if** statement to conditionally execute command lines
- Use the **case** statement to conditionally execute command lines
- Use special shell constructs for conditional command execution

AV 7-1: Decision Constructs

This first slide describes the objectives covered in this lesson.

7.1 Exit Codes

When a process is created, it inherits properties from its parent process, including the environment variables that had been exported. Normally, the child process has little opportunity to communicate with its parent process—in fact, the parent process usually just waits for its child process to terminate.

When the child process does terminate, it can inform its waiting parent process of the event and also why it terminated. This information is presented to the parent process in the form of an integer value called an *exit code* or *exit status*.

7.1.1 Communication mechanism

A parent process can determine why a child process terminated by looking at the child process's exit status. The shell automatically sets the `$?` variable to indicate the exit status of the last command which executed in the foreground (where the parent process waits for the child process to finish).

Certain shell programming constructs implicitly examine the value of `$?` to determine why a process terminated.

GuruNotes The Bourne and Korn shells set the `$?` variable to indicate the exit status of the last command which executed in the foreground; the C shell uses the `$status` variable instead.

7.1.2 Conventional exit codes

By convention, a command which terminates successfully (it performs its duties and has nothing more to do) exits with a zero (0) value.

By convention, a command which terminates unsuccessfully (it aborted because something went wrong) exits with a non-zero value in the range 1-255 (inclusive).

The non-zero value is program-defined and may have a different meaning for different commands.

For example, the **grep** command searches for a pattern within a file and displays those lines which contain the pattern. It uses these exit codes:

- | | |
|---|--|
| 0 | Successfully opens the file, finds at least one line with the pattern, and prints it |
| 1 | Successfully opens the file, but cannot find the pattern in the file |

2 Cannot open the file or some other error occurs

The shell regards a zero exit status as “true” and a non-zero status as “false.” This is in contrast to other programming languages—such as C—where a zero integer value is false and any other value is true.

Some commands use exist codes even when exiting successfully to pass information to the parent process.

The next exercise gives students an opportunity to observe different exit codes. To supplement this exercise, show these examples:

```
$ grep root /etc/passwd          # This will fail
grep: cannot open /etc/passwd
$ echo $?                        # As expected, $? = 2
2
$ echo $?                        # But now, $? = 0
0
```

The point is that `$?` refers to the exit status of the *last* foreground command. The second **echo** command displays the result of the first **echo** command—not the result of **grep**.

Exercise: Viewing an exit status

Use the **echo** command to determine the exit status of these commands:

```
grep root /etc/passwd
grep route /etc/passwd
grep root /etc/password
```

7.1.3 The exit command

Indicate why this command *must* be a shell built-in. Ask how it would execute if it were a binary program instead.

A shell script can use the built-in **exit** command to terminate at any time—for success or failure. The syntax is:

```
exit [n]
```

If *n* is omitted, \$? (the exit status of the last foreground command) is assumed.

Use *n=0* for successful termination and a non-zero value in the range 1-255 to indicate failure.

It is a good idea to terminate all scripts with a known exit value rather than letting it “fall off the edge” and terminate implicitly. For this reason, the last line of most shell scripts is:

```
exit 0
```

Programmers may choose any exit status they want when they write code. Aside from following the convention of zero for success and non-zero for failure, there is no convention for *which* non-zero code to use for a particular failure. That is completely program defined and should be documented by the programmer.

7.1.4 Specialized commands with constant exit codes

The system has a few commands that perform no real work but nonetheless exit with a known exit status. These commands are commonly used in looping constructs (discussed in a later lesson).

7.1.4.1 The null command (:)

The built-in null command (:) does nothing; however, it returns with a zero (0) exit status.

This command is sometimes used in conditional contexts to satisfy syntax requirements that mandate that some “true clause” be provided.

7.1.4.2 The true command

The **true** command does nothing but exit with a zero (0) exit status.

7.1.4.3 The false command

The **false** command does nothing but exit with a non-zero exit status.

7.2 The test Command

GuruNotes Although **test** is now a shell built-in command, it used to be a binary that had two file links: `/bin/test` and `/bin/[` (bracket `[` is a valid filename). These files still exist for backwards compatibility, but they are now scripts which invoke the built-in version.

The **test** command is a shell built-in command that tests various conditions and then silently exits with an exit status that indicates the outcome of the test: a zero (0) status indicates the condition was true; a non-zero status indicates the condition was false. This command is used in the majority of shell decision and looping constructs. The **test** command has two names with which it can be invoked.

- **test**
- **[** (left bracket)

The left bracket (`[`) form of the command has a syntax that requires that the last command-line argument be a matching right bracket (`]`). This is in addition to the general requirement that all command-line arguments be separated by a space character. Be sure there is a field separator character after the command name (e.g., `[`) and before the last argument (e.g., `]`).

The bracket form of the command is often preferred over the **test** form because it tends to make certain shell programming constructs easier to read (like other programming languages). The two commands are otherwise identical in function. The general syntax is:

```
test arg ...  
[ arg ... ]
```

Stress to students that bracket (`[`) is the name of a command and should be treated like any other command name—namely, it must be followed by a space. Also stress that the bracket form of the command must have a closing right bracket (`]`) as the last command line argument. This argument should be treated like any other command-line argument—namely, it must be surrounded by field separator characters (here, space to the left, newline to the right).

The **test** (or `[`) command tests three different types of conditions:

- File status
- Character strings
- Integer expressions

The arguments may be constants or, more commonly, variables. Variables are usually quoted to prevent syntax errors when they contain null strings or shell metacharacters.

AV 7-2

Testing Files

```
test option file
```

```
or
```

```
[ option file ]
```

Option**Description**

-r	True if <i>file</i> exists and is readable
-w	True if <i>file</i> exists and is writable
-x	True if <i>file</i> exists and is executable
-f	True if <i>file</i> exists and is a regular file
-d	True if <i>file</i> exists and is a directory
-h or -L	True if <i>file</i> exists and is a symbolic link
-c	True if <i>file</i> exists and is a character special file
-b	True if <i>file</i> exists and is a block special file
-u	True if <i>file</i> exists and its set-user-ID bit is set
-g	True if <i>file</i> exists and its set-group-ID bit is set
-k	True if <i>file</i> exists and its sticky bit is set
-s	True if <i>file</i> exists and has a size greater than zero
-t <i>n</i>	True if the open file whose file descriptor number is <i>n</i> (1 by default) is associated with a terminal device

sp0702

© The Santa Cruz Operation, Inc.

7.2.1 File status

AV 7-2: Testing Files

This slide summarizes the **test** options.

The **test** command can check on the status of a file by testing:

- Whether certain permissions are granted
- Whether a file is empty or not
- Whether a file descriptor is associated with a terminal device

If the test is true, the command exits with a zero (0) exit status; if false, with a non-zero status.

The general syntax is:

```
test option filename  
[ option filename ]
```

These examples use the **test** commands to check the status of a file:

```
test -r "$FILE"  
[ -b /dev/rfd0135ds18 ]  
[ -t 1 ]
```

The **-t** option can be used to help determine whether I/O redirection is currently in effect. For example, if file descriptor 1 (the *stdout*) is not associated with a terminal (as it usually is), then redirection must have taken place.

AV 7-3

Testing Character Strings

```
test string1 option string2    [ string1 option string2 ]  
test option string1            or [ option string1 ]  
test string1                   [ string1 ]
```

<u>Option</u>	<u>Description</u>
=	True if <i>string1</i> and <i>string2</i> are identical
!=	True if <i>string1</i> and <i>string2</i> are not identical
-z	True if the length of <i>string1</i> is zero
-n	True if the length of <i>string1</i> is non-zero
(no option)	True if <i>string1</i> is not the null string (-n alternate)

sp0703

© The Santa Cruz Operation, Inc.

7.2.2 Character strings

AV 7-3: Testing Character Strings

This slide summarizes the **test** options for testing character strings.

The **test** command can be used to:

- Compare two character string values
- Determine whether a string has a null value

If the test is true, the command exits with a zero (0) exit status; if false, with a non-zero status.

The general syntax is:

```
test string1 option string2
test option string1
test string1
```

```
[ string1 option string2 ]
[ option string1 ]
[ string1 ]
```

These examples use the **test** commands with strings:

```
test "$REPLY" = YES
[ -z "$ANSWER" ]
[ "$MYVAR" ]
```

Remind students that it is usually a good idea to enclose variables within double quotes. Show these examples to drive home the point:

```
x= '    no    ' # Note the spaces in $x
[ $x = no ]      # Spaces in $x not protected; exit with 0
[ "$x" = no ]    # Spaces protected; exit with 1
```

Also note that the following example produces a syntax error when \$x is not quoted:

```
x= ' '          # $x is set to null
[ $x = no ]     # Shell sees no operand to the left of the equal to (=) operator
```

AV 7-4

Testing Integer Expressions

```
test string1 option string2  
or  
[ string1 option string2 ]
```

<u>Option</u>	<u>Description</u>
-eq	True if <i>string1</i> is algebraically equal to <i>string2</i>
-ne	True if <i>string1</i> is algebraically not equal to <i>string2</i>
-lt	True if <i>string1</i> is algebraically less than <i>string2</i>
-le	True if <i>string1</i> is algebraically less than or equal to <i>string2</i>
-gt	True if <i>string1</i> is algebraically greater than <i>string2</i>
-ge	True if <i>string1</i> is algebraically greater than or equal to <i>string2</i>

sp0704

© The Santa Cruz Operation, Inc.

7.2.3 Integer expressions

AV 7-4: Testing Integer Expressions

This slide summarizes the **test** options for testing integer expressions.

The **test** command can interpret two strings as integer values and compare them for arithmetic relationships.

If the relationship is true, the command exits with a zero (0) exit status; if false, with a non-zero status.

The general syntax is:

```
test string1 option string2
[ string1 option string2 ]
```

These examples use the **test** commands with strings:

```
test "$COUNT" -lt 25
[ $# -eq 3 ]
```

Contrast the difference between testing for string equality and testing for numeric equality. Show this example:

```
x=' 5 '          # Note the spaces in $x
y=5             # Note no spaces in $y
[ "$x" = "$y" ]  # Spaces protected, strings not equal
[ "$x" -eq "$y" ] # Spaces protected, values are equal
```

7.2.4 Logical operators

The preceding tests involving files, strings, and integer values all formed expressions that used the primary operators of the **test** command. These expressions can be combined with **test**'s logical operators:

- **!** (logical NOT)
- **-a** (logical AND)
- **-o** (logical OR)

These three logical operators are listed in descending order of precedence (**-a** has higher precedence than **-o**).

Use parentheses `[ ( ... ) ]` if a different evaluation order is needed for your expression.

- Remember that parentheses have meaning to the shell and will need to be quoted.
- Each parenthesis (left and right) must appear as a separate command line argument; spaces should come before and after.

7.2.4.1 The **!** operator

The **!** operator negates the outcome of the following **test** expression: true becomes false (non-zero exit status), false becomes true (zero exit status). For example:

```
[ ! -r "$FILE" ]
```

A common misunderstanding is that this operator can be used for any command. Of course, this is not true—it is interpreted by the **test** command. For example:

Do I *not* have read access on a file? This is valid:

```
[ ! -r filename ]
```

Did **grep** *not* find a pattern in a file? This is invalid:

```
! grep pattern filename
```

7.2.4.2 The **-a** operator

The **-a** operator logically ANDs two expressions.

Both expressions must be true for **test** to exit with a true (zero) status. If either expression (or both) is false, **test** exits with a false (non-zero) status. For example:

```
test -r "$PROGNAME" -a -x "$PROGNAME"  
[ "$NUMBER" -gt 0 -a "$NUMBER" -lt 10 ]
```

Note the following is not a valid expression:

```
[ "$NUMBER" -gt 0 -a -lt 10 ]
```

7.2.4.3 The -o operator

The **-o** operator logically ORs two expressions.

Either expression (or both) must be true for **test** to exit with a true (zero) status. If both expressions are false, **test** exits with a false (non-zero) status. For example:

```
test -r "$FILE1" -o -r "$FILE2"  
[ "$REPLY" = YES -o "$REPLY" = yes ]
```

Exercise: Using the test command

Indicate the **test** command for these conditions:

- Is **\$FILE** a directory with both read and search permissions?
- Is **\$DEVICE** a character special file with either read or write permissions?
- Has the standard input been redirected?
- Did a user answer a prompt with a “no” response? (Allow for any permutation in the capitalization of the word “No”.)
- Is the current number of positional parameters not greater than 12?

AV 7-5

The If Statement

```
if test "$ANSWER" = quit
then
    echo "Quitting program at your request."
    exit 0
fi

if [ $# -lt 2 ]
then
    echo "$0: Insufficient arguments ($#)"
    echo "Usage: $0 f1 f2"
    echo "$0 f1 ... fn d1"
    exit 1
fi >&2

if mkdir "$NEWDIR"
then
    cd "$NEWDIR"
fi

if who | grep "$1" > /dev/null 2>&1
then
    write "$1" <<- EOF
        Please call me ASAP!
EOF
```

sp0705

© The Santa Cruz Operation, Inc.

7.3 The if Statement

One decision-making construct of the shell is the **if** statement. With this statement, you can cause certain command lines in your shell script to be executed or skipped over, depending on a given condition.

7.3.1 Syntax

The basic **if** syntax is:

```
if command-list1
then
    command-list2
    ...
fi
```

AV 7-5: The if Statement

This slide shows several examples of simple **if** statements. Note the use of **test** with both names (**test** and **[]**); also note that **test** does not need to be the command in the *command-list*. Any command can be used.

WARNING! The second example on the slide shows a problem with performing I/O redirection on an **if** statement. The **exit** statement will *not* terminate the script—even if **\$#** is less than 2! Why? Because the redirection caused the body of the **if** statement to execute in a *subshell* and it is this subshell that **exit** will terminate.

7.3.1.1 Overview

Command lists have these properties:

- Each *command-list* is either a single command, a list of commands, or a pipeline.
- *command-list1* frequently—but not always—consists of the **test** command.
- Each *command-list* is executed by the current shell unless I/O redirection or a pipe has been set up (see below).
- Although not required, it is conventional to indent *command-list2* to improve the readability of the construct.

Strongly recommend that students get in the habit of indenting. The tab character is most commonly used, but even a few spaces is better than no indenting at all!

7.3.1.2 I/O redirection and pipes

Input/output redirection to or from a file can be set up for the entire **if** construct (including *command-list1* and *command-list2*) by placing the appropriate I/O redirection syntax (e.g., `<`, `>`, `>>`) immediately after the **fi** statement.

Piping input to an **if** construct is done immediately before the **if** statement. Piping output from an **if** construct is done immediately after the **fi** statement.

Setting up I/O redirection or a pipe causes both *command-lists* to be executed in a subshell.

AV 7-6

The else Statement

```
if mkdir "$NEWDIR"
then
    cd "$NEWDIR"
else
    echo "Cannot create directory $NEWDIR" >&2
    exit 1
fi

if who | grep "$1" > /dev/null 2>&1
then
    write "$1" <<- EOF
    Please call me ASAP!
EOF
else
    mail -s "Please call" "$1" <<- EOF
    Please call me ASAP!
EOF
fi

if [ -d "$FILE" ]
then
    rmdir "$FILE"
else
    rm "$FILE"
fi
```

sp0706

© The Santa Cruz Operation, Inc.

7.3.2 Execution

Execution of an **if** statement proceeds as follows:

- *command-list1* executes. The last command which executes in the list is the “relevant command.”
- The exit status of the relevant command is examined:
 - If it is zero (0), then *command-list2* is executed.
 - If it is non-zero, then *command-list2* is skipped and execution continues with the first statement which follows the **fi** statement.

7.3.3 The else statement

AV 7-6: The else Statement

This slide shows several examples of simple **if-else** statements.

In a regular **if** statement, if *command-list1* exits with a non-zero exit status, then *command-list2* is skipped and execution continues with the first statement which follows the **fi** statement.

An **if** statement may also include an optional **else** statement so that if *command-list1* exits with a non-zero exit status, *command-list2* is still skipped, but *command-list3* is executed instead.

The syntax is:

```
if command-list1
then
    command-list2
    ...
else
    command-list3
    ...
fi
```

As usual, *command-list1* always executes. Either *command-list2* or *command-list3* always executes, but never both of them for a given execution instance.

AV 7-7

The elif Statement

```
if mkdir "$NEWDIR"
then
    cd "$NEWDIR"
elif mkdir "/tmp/$NEWDIR"
then
    cd "/tmp/$NEWDIR"
else
    echo "Cannot create directory" >&2
    exit 1
fi

if [ "$MENUITEM" = c -o "$MENUITEM" = C ]
then
    cat "$FILE"
elif [ "$MENUITEM" = m -o "$MENUITEM" = M ]
then
    more "$FILE"
elif [ "$MENUITEM" = l -o "$MENUITEM" = L ]
then
    lp "$FILE"
elif [ "$MENUITEM" = r -o "$MENUITEM" = R ]
then
    rm -i "$FILE"
else
    echo "Unknown menu choice" >&2
```

sp0707

© The Santa Cruz Operation, Inc.

7.3.4 The elif statement

AV 7-7: The elif Statement

This slide shows several examples of **elif** statements. Show the equivalent code that uses **else if** statements instead (but be sure to indicate the need for extra **fi** statements).

An **else** statement causes *command-list3* to execute when *command-list1* exits with a non-zero exit status. This *command-list3* may consist of any number of statements or command lines.

If *command-list3* consists of only one statement and it is an **if** statement (which may contain any number of its own statements), then the **else/if** statements may be substituted with an **elif** statement.

If *command-list3* consists of several statements which include an **if** statement, then **elif** cannot be logically substituted.

An **else/if** construct requires a closing **fi** for that additional **if**; an **elif** construct does not require the closing **fi**.

An **elif** statement may optionally have an **else** or **elif** statement of its own.

The syntax is:

```
if command-list1
then
    command-list2
    ...
elif command-list
then
    command-list
    ...
else
    command-list
    ...
fi
```

AV 7-8

Nested If Statements

```
echo "Do you want to format the disk now? [y/n] \c"
read REPLY1 DUMMY
if [ "$REPLY1" = y ]
then
    echo "Are you sure? [y/n] \c"
    read REPLY2 DUMMY
    if [ "$REPLY2" = y ]
    then
        echo "Last chance. Are you really, really sure? [y/n] \c"
        read REPLY3 DUMMY
        if [ "$REPLY3" = y ]
        then
            echo Formatting...
            format "$DISK"
        fi
    fi
fi
else
    echo "Installation cannot proceed at this time. Please
    backup the data on the disk and try again later."
    exit 1
fi
```

sp0708

© The Santa Cruz Operation, Inc.

7.3.5 Nested if statements

AV 7-8: Nested if Statements

This slide shows an example of a nested **if** statement.

The **then** statement causes *command-list2* to execute when *command-list1* exits with a zero (0) exit status. This *command-list2* may likewise consist of any number of statements or command lines.

One of the statements in *command-list2* can be another **if** statement (which in turn may have its own **else** statement). This is referred to as a *nested if* statement.

There is no construct similar to the **elif** statement. Therefore, the nested **if** will require a closing **fi** statement of its own.

The syntax is:

```
if command-list1
then
    if command-list
    then
        command-list
        ...
    else
        command-list
        ...
    fi
else
    command-list
fi
```

All **else** statements are optional. The shell associates an **else** statement with the closest **if** statement above it that is not already closed off with the **fi** statement.

AV 7-9

Nested If Statements

```
echo "Do you want to format the disk now? [y/n] \c"
read REPLY1 DUMMY
if [ "$REPLY1" = y ]
then
    echo "Are you sure? [y/n] \c"
    read REPLY2 DUMMY
    if [ "$REPLY2" = y ]
    then
        echo "Last chance. Are you really, really sure? [y/n] \c"
        read REPLY3 DUMMY
        if [ "$REPLY3" = y ]
        then
            echo Formatting...
            format "$DISK"
        fi
    fi
fi
else
    echo "Installation cannot proceed at this time. Please
    backup the data on the disk and try again later."
    exit 1
fi
```

sp0709

© The Santa Cruz Operation, Inc.

AV 7-9: Nested if Statements

AV 7-8 is repeated to help students with this exercise.

Exercise: Nested if Statements

Modify the program fragment in AV 7-8 (on paper only) to add this functionality:

- Allow the user to enter upper- or lower-case one-character responses.
- Currently, the program deals with “y” and non-”y” responses from a user. Add the logic so that it can deal with “y”, “n”, and non-”y/n” responses (for \$REPLY1). When a response other than “y” or “n” is entered, display the message “Unknown response” and then exit.
- Currently, what happens when a user changes his or her mind on the second or last chance prompt? Modify the code so that the “Installation cannot proceed ...” message is displayed if a user enters “n” at any of the three prompts.

AV 7-10

The case Statement

```
#!/bin/sh
# Sample program which demonstrates menus with a case statement

echo "Enter filename ==> \c"
read FILE DUMMY

echo "FILE MENU
-----
c)at filename
m)ore filename
l)ine printer filename
r)emove filename
Enter choice ==> \c"

read MENUITEM DUMMY
case "$MENUITEM" in
  c | C ) cat "$FILE"
          ;;
  m | M ) more "$FILE"
          ;;
  l | L ) lp "$FILE"
          ;;
  r | R ) rm -i "$FILE"
          ;;
  * )    echo "\07Unknown menu choice" >&2
          exit 1
          ;;
esac
```

sp0710

© The Santa Cruz Operation, Inc.

7.4 The case Statement

The shell has another decision-making construct called the **case** statement.

It is often easier to use a **case** statement than a corresponding **if** statement when you are trying to compare one value against many string values for equality.

Numeric relationships and string tests for *inequality* cannot be tested with a **case** statement; use **if** for this purpose.

7.4.1 Syntax

AV 7-10: The case Statement

This slide shows a simple menu program that uses a **case** statement.

The general syntax is:

```
case word in
    pattern_1 ) command-list
                ;;
    pattern_2 ) command-list
                ;;
    ...
    pattern_n ) command-list
                ;;
esac
```

where:

- The expression *word* is nearly always the value in a shell variable. It is often a good idea to quote it.
- A pair of semicolons (*;;*) terminates each *command-list*. It is not required that they be on a line by themselves, but doing so will improve the readability of your program.
- Each *pattern* is a character string that is compared for equality against *word*. The string may consist of wildcards that are the same as those used for filename generation:
 - *** matches zero or more of any character
 - *?* matches exactly one of any character
 - [*string*] matches exactly one of any character in *string*

- `[!string]` matches exactly one of any character not in *string*

A given *pattern* may be one or more separate strings, each separated by a vertical bar (`|`).

7.4.2 Execution

The **case** statement executes the *command-list* associated with the first *pattern* that matches *word*.

The *command-list* is terminated by a pair of semicolons (`;;`). Execution resumes with the first statement that follows the **esac** statement.

If no *pattern* matches *word*, then none of the *command-lists* are executed; execution resumes with the first statement that follows the **esac** statement.

Use the following as an optional default for when no other *pattern* matches *word*:

```
* ) command-list
   ;;
```

This should be specified as the last *pattern* in the **case** statement, otherwise you might not get selective execution.

Exercise: The case Statement

- Simplify each of these **case** statement *patterns* with ones that use wildcards:

NO | no | No | nO)

s | S)

21 | 22 | 23 | 24 | 25 | 26 | 29)

- What *pattern* matches any three-character string that begins with either “y” or “Y”?
- What *pattern* matches any string that does *not* begin with an “n” or “N”?

AV 7-11

The `&&` and `||` Constructs

```
[ -r report -a -w report ] && vi report  
mkdir /tmp/dir$$ && cd /tmp/dir$$ && PS1="`pwd` "  
mkdir /tmp/dir$$ || echo "Cannot create /tmp/dir$$" >&2  
write jeff < message || mail -s 'Read this' jeff < message
```

sp0711

© The Santa Cruz Operation, Inc.

7.5 Special Constructs

AV 7-11: The && and || Constructs

This slide shows an example of the && and || decision constructs.

The shell also has the && and || decision constructs which are shorthand notations for the **if** construct discussed earlier.

7.5.1 The && construct

GuruNotes The C shell also has && and || constructs, but due to a historical bug, they have the opposite meaning from the Bourne/Korn shells, the C programming language, and other UNIX utilities!

The syntax for the && construct is:

```
command1 && command2
command1 && command2 && command3 ...
```

command1 executes and its exit status is examined:

- If it exits with a zero (0) exit code, then *command2* executes; otherwise *command2* (or the rest of the line) is skipped.
- If *command2* exits with zero, then *command3* executes, and so on.

An equivalent **if** statement is:

```
if command1
then
    command2
fi
```

7.5.2 The || construct

The syntax for the || construct is:

```
command1 || command2
command1 || command2 || command3 ...
```

command1 executes and its exit status is examined:

- If it exits with a non-zero exit code, then *command2* executes; otherwise *command2* (or the rest of the line) is skipped.
- If *command2* exits with a non-zero status, then *command3* executes, and so on.

An equivalent **if** statement is:

```
if command1
then
    : # do nothing
else
    command2
fi
```


AV 7-12

The `&&` and `||` Constructs

```
[ -r report -a -w report ] && vi report
mkdir /tmp/dir$$ && cd /tmp/dir$$ && PS1="`pwd` "
mkdir /tmp/dir$$ || echo "Cannot create /tmp/dir$$" >&2
write jeff < message || mail -s 'Read this' jeff < message
```

sp0712

© The Santa Cruz Operation, Inc.

7.5.3 Combining the && and || constructs

Both the && and || constructs can be used on one command line; for example:

```
command1 && command2 || command3 && command4
```

command1 executes and its exit status is examined:

- If it exits with a zero (0) exit status, then *command2* executes; otherwise the rest of the line is skipped.
- If *command2* exits with non-zero, then *command3* executes; otherwise the rest of the line is skipped.
- If *command3* exits with zero (0), then *command4* executes; otherwise the rest of the line is skipped.

AV 7-12: The && and || Constructs

AV 7-11 is repeated to aid students in completing the exercise.

Exercise: The && and || Constructs

Indicate the logically equivalent **if** statement for each command line in AV 7-12.

Instructor Guide

GuruNotes The shell uses curly braces ({}) to group commands together for execution by the current shell (compared to parentheses which groups and executes commands in a subshell). These are equivalent:

```
if command1
then
    command2
    command3
fi
```

```
command1 && { command2; command3; }
```

Here, the braces are required, otherwise *command3* would always execute regardless of the outcome of *command1*.

7.6 Lesson Summary

The shell has several mechanisms that allow you to conditionally execute lines of code during the execution of your shell script: the **if** and **case** statements, and the **&&** and **| |** constructs.

Central to making decisions is the exit status of a command. Every command terminates with an exit status that can be explicitly seen with the **\$?** shell variable. It is implicitly used by several of the shell programming statements (such as **if**). By convention, a command that terminates successfully exits with a zero (0) exit status; unsuccessful commands terminate with a non-zero exit status in the range 1-255 (inclusive).

Review questions

1. What is the conventional exit status for a command which terminates normally? Abnormally?
2. What value does **exit** use by default?
3. Which special shell variable allows you to view the exit status of a command?
4. Which three types of conditions can the **test** command examine?
5. What are the primary operators of the **test** command? How do they work?

6. Does this statement evaluate to true or false when `$FILE` is a regular (non-empty) file with full read permission?

```
[ ! -f "$FILE" -a -r "$FILE" -o ! -s "$FILE" ]
```

7. What is the purpose of *command-list1* in an **if** statement? How is it interpreted?
8. What is a *nested if* statement? How does the shell match up the **else** statement to its corresponding **if** statement?
9. Which wildcards can be used in a *pattern* of a **case** statement? What pattern can be used as a default case?
10. What is the equivalent **if** statement for the **&&** and **||** constructs?

7.7 Lab: Decision Constructs

This lab gives you experience with the shell's decision-making constructs by modifying programs written earlier in the course to:

- Accept multiple command-line arguments
- Present an option menu and prompt for input

7.7.1 A friendly find: revisited

Modify the **search** program from Section 5.7.3, “A friendly find,” on page 5-39, so that it can accept 0, 1, or 2 command-line arguments.

Specifications

These specifications should be implemented in the program:

0 arguments	Begin the search at the current directory. Search for a filename as specified in the environment variable SEARCHFILE . If the variable is NULL (or not set), prompt the user for a filename.
1 argument	Begin the search at the current directory. Search for a filename as specified in the argument.
2 arguments	These specify both the start directory and the filename

7.7.2 Phone database: revisited

In this exercise, modify the Phone Database programs from your previous labs to use a menu interface.

Overview

Write a front-end script called **phone** that displays an initial menu that resembles this:

```
TELEPHONE DATABASE MENU
-----
1. Add name to database
2. Search database
3. Display entire database
4. Purge the database
5. Quit

Enter choice ==>
```

Specifications

These specifications should be implemented in the program:

- A user can select a given menu item by choosing either the corresponding number or by entering the first letter of the menu choice text. For example, a user can enter “2”, “S”, or “S” to search the database.
- Menu choices 1 and 2 incorporate the functionality you have already written in the previous labs.
- Menu choice 3 displays the entire database with a file pager, such as **more** or **pg**.
- Menu choice 4 verifies that the user really wants to purge the file (i.e., truncated, but not removed).

7.8 Relevant Commands and Files

The following commands and files were presented in this lesson.

Relevant commands

:	Built-in null command that does nothing other than exit with a zero (0) exit status
case / esac	Built-in command which compares a word against patterns to select a command list to execute
exit	Built-in command which terminates a shell
false	Does nothing other than exit with a non-zero (255) exit status
grep	Searches a file for regular expressions and prints lines that match
if / then / elif / else / fi	Built-in command which allows conditional execution of command lines
test, [	Built-in command which tests file status, character strings, and integer expressions
true	Does nothing other than exit with a zero (0) exit status

7.9 Answers to Review Questions

1. The conventional exit status for a command which terminates normally is zero (0); for abnormal termination, it is any non-zero integer value in the range 1-255 (inclusive).
2. By default, exit uses the exit status of the previously executed command (i.e., \$?).
3. \$? is the special shell variable that allows you to view the exit status of a command.
4. The three types of conditions that the test command examines are: file status, string values, and integer expressions.
5. The primary operators of the test command are: logical NOT (!), logical AND (-a), and logical OR (-o). NOT negates the truth value of an expression—true (0) becomes false (non-zero) and vice versa. AND combines two expressions and evaluates to true (0) only if both expressions were true; otherwise it returns false (non-zero). OR combines two expressions and evaluates to false (non-zero) only if both expressions were false; otherwise it returns true (0).
6. The logical operators, !, -a and -o, are evaluated in descending order of precedence and tests true when \$FILE is a regular (non-empty) file with read permission. Using parenthesis to show the order of evaluation: [(((! -f "\$FILE") -a -r "\$FILE") -o -s "\$FILE")] [(((! T) -a T) -o T)] [(((F -a T) -o T))] [(F -o T)] [T]
7. In an if statement, command-list1 always executes. Its success or failure (as measured by its exit status) indicates whether or not to execute command-list2.
8. A “nested” if statement is comprised of an if statement within another if statement—specifically, it’s an if that makes up the “true” clause of another if statement. The shell associates an else statement with the closest if statement above it that has not already been closed off with the fi statement.
9. The wildcards that can be used in a pattern of a case statement are the same as those used for filename matching: *, ?, and [string]. Use the * pattern as a default case.
10. The equivalent if statement for command1 && command2 is: if command1 then
command2 fi
The equivalent if statement for command1 || command2 is: if
command1 then : # do nothing else command2 fi

8

Looping Constructs

Instructor Guide

Instructor Guide

Lesson 8

Looping Constructs

Suggested lesson duration

This lesson should take approximately 2 hours to complete, including the lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 8-1

The slide is titled "Looping Constructs" in a blue-bordered box at the top. Below the title, there is a bulleted list of four items. At the bottom of the slide, there is a blue footer bar containing the text "sp0801" on the left and "© The Santa Cruz Operation, Inc." on the right.

- Use the **expr** command
- Use **while** and **until** statements
- Use a **for** statement
- Control loop execution

sp0801 © The Santa Cruz Operation, Inc.

Instructor Guide

Introduction

Another characteristic of many programming languages is the ability to repeatedly execute a block of code, as long as certain conditions are present. These constructs can simplify your programming job so that it is unnecessary to indicate the same statement(s) over and over again in your code. The Bourne shell uses these statements to accomplish this repetitive functionality:

- **while**
- **until**
- **for**

This lesson describes the implementation of the Bourne shell looping constructs and the related commands that can affect their execution.

Objectives

This lesson describes the shell's looping constructs. By the end of this lesson, you will be able to:

- Use the **expr** command to perform integer arithmetic on variables
- Use **while** and **until** statements to execute a section of code repeatedly
- Use a **for** statement to execute a section of code repeatedly
- Control the execution of a loop

AV 8-1: Looping Constructs

This first slide describes the objectives covered in this lesson.

AV 8-2

The `expr` Command

```

$ expr 2 + 13
15

$ NUMBER=42
$ expr "$NUMBER" / 5
8

$ expr "$NUMBER" % 5
2

$ expr "$NUMBER" % 6
0

$ expr "$NUMBER" % 60
42

$ expr 7 + 3 \* 8
31

$ expr \( 7 + 3 \) \* 8
80

```

0802
© The Santa Cruz Operation, Inc.

8.1 The `expr` Command

AV 8-2: The `expr` Command

This slide shows an example of the `expr` command.

The `expr` command evaluates expressions on the command line and writes the results to the standard output. It is useful for performing variable arithmetic within a script. You can also use `expr` to evaluate regular expressions. This functionality, while not discussed in this course, is documented in the `expr` manual pages.

This course does not cover the use of `expr` to evaluate regular expressions. Refer advanced students to the `expr` manual pages if they want information on this topic.

8.1.1 General syntax

The general syntax of `expr` is:

```
expr arg ...
```

Each *arg* is a term in the expression. A term is either an operand or operator. For example, the expression “4 + 2” consists of three terms: “4”, “+”, and “2”.

Some operators can be shell metacharacters. It is necessary to quote them to prevent interpretation by the shell.

Numeric terms must be integer values (positive or negative). Floating point terms are illegal; floating point results are truncated towards zero (0) to produce an integer value.

8.1.2 Arithmetic operators

Some students might be unfamiliar with the modulus operation. Review with them as necessary. A good example of modulus division is converting time from a 24-hour clock to a 12-hour one (18:00 = 6:00pm) (modulo 12).

The `expr` command supports two types of arithmetic operators:

- Multiplicative operators:

*	Multiplication
/	Integer division
%	Modulus (remainder division)

- Additive operators:

- + Addition

- Subtraction

AV 8-3

Variable Arithmetic

```

$ COUNT=10
$ COUNT=$COUNT+1
$ echo $COUNT
10+1

$ COUNT=10
$ COUNT=`expr $COUNT + 1`
$ echo $COUNT
11

```

sp0803
© The Santa Cruz Operation, Inc.

Instructor Guide

8.1.3 Operator precedence

Remind students that parentheses `[ ( ... ) ]` have meaning to the shell and therefore have to be quoted to avoid interpretation. One cannot quote all the command arguments to **expr** with just one set of parentheses—that would also remove the meaning of the spaces which are still required. Each individual argument must be quoted (if needed).

The multiplicative operators are equal with each other in associative precedence; likewise, the additive operators are equal with each other.

The multiplicative operators are higher in precedence than the additive operators.

Use parentheses `[ ( ... ) ]` to change the associativity.

8.1.4 Variable arithmetic

AV 8-3: Variable Arithmetic

This slide shows how not to do variable arithmetic, and then how to properly do it with the **expr** command used with command substitution.

In the Bourne shell, you cannot directly do arithmetic on shell variables. Use **expr** with command substitution to perform arithmetic operations on variables.

For example:

```
$ NUMBER=35
$ NUMBER=`expr "$NUMBER" \* 5`
$ echo $NUMBER
175
```

Exercise: Using expr

- A shell script uses two variables for timekeeping purposes: \$HOUR and \$MIN. Modify \$HOUR to reflect a 12-hour clock instead of a 24-hour clock. (Do not be concerned with AM/PM notations).
- Check the value of the shell variable \$COUNT. If it is not larger than 100, increment it by one (1); otherwise, reset it to zero (0).

AV 8-4

The **while** Statement

```
#!/bin/sh
# Send a file multiple times to a printer with no formfeed
# between copies. (Useful for label printers)
# Syntax: lplabel ncopies filename

if [ $# -ne 2 ]
then
    echo "Syntax: $0 ncopies filename" >&2
    exit 1
fi

> tmp$$ # Create temporary file
COUNT=0

while [ "$COUNT" -lt $1 ]
do
    cat "$2" >> tmp$$
    COUNT=`expr "$COUNT" + 1`
done

lp tmp$$
rm tmp$$
```

sp0804

© The Santa Cruz Operation, Inc.

8.2 The while Statement

One of the shell looping constructs is the **while** statement. This causes a command list to repeatedly execute as long as certain conditions remain true.

8.2.1 Syntax

AV 8-4: The while Statement

This slide shows a sample shell script which uses a **while** loop. The script concatenates a file several times in a temporary file and then prints it. Review the `##` and `$$` shell special variables when discussing the program operation.

The basic **while** syntax is:

```
while command-list1
do
    command-list2
...
done
```

Note that general syntax and operation are somewhat similar to the **if** statement.

8.2.1.1 Overview

Each *command-list* is either a single command, a list of commands, or a pipeline; *command-list1* frequently—but not always—consists of the **test** command.

Each *command-list* is executed by the current shell unless I/O redirection or a pipe has been set up (see Section 8.2.1.2, “I/O redirection and pipes,” on page 8-15).

Although not required, it is conventional to indent *command-list2* to improve the readability of the construct.

8.2.1.2 I/O redirection and pipes

I/O redirection to or from a file can be set up for the entire **while** construct (including *command-list1* and *command-list2*) by placing the appropriate I/O redirection syntax (e.g., `<`, `>`, `>>`) immediately after the **done** statement.

Piping input to a **while** construct is done immediately before the **while** statement. Piping output from a **while** construct is done immediately after the **done** statement.

Setting up I/O redirection or a pipe causes both *command-lists* to be executed in a subshell.

These notes also apply to the **until** and **for** loop constructs (as well as to the decision constructs already discussed).

AV 8-5

Nested while Statements

```
#!/bin/sh
# Display several files with line numbers
# Syntax: number file ...
while [ $# -gt 0 ]
do
    LINE_NUM=1
    cat $1 | while read LINE
    do
        echo "$LINE_NUM\t$LINE"
        LINE_NUM=`expr $LINE_NUM + 1`
    done
    shift
    if [ $# -gt 0 ]
    then
        echo "Press <Enter> to continue..."
        read DUMMY
    fi
done
```

sp0805

© The Santa Cruz Operation, Inc.

8.2.2 Execution

AV 8-5: Nested while Statements

This program shows nested **while** statements. It displays a file along with line numbers. Things to note include:

- The output of **cat** is piped into the inner loop. The loop will continue until **read** fails at EOF.
- Each inner loop iteration continues in the file from where the previous iteration left off, processing the next line in the file.
- The **shift** command is used to allow many files to be specified as command-line arguments. It will decrement the value of \$# so that the outer loop will ultimately terminate.
- The **read DUMMY** statement is a convenient way to pause the program until the user is ready to continue.

Like any command, the **while** statement can be used on interactive command lines (such as from a login shell). Unlike the **if** statement, it is not uncommon for it to be used in this way.

Execution of a **while** statement proceeds as follows:

- *command-list1* executes. The last command which executes in the list is the *relevant command* (just like the **if** statement).
- The exit status of the relevant command is examined:
 - If it is zero (0), then *command-list2* is executed and then control returns back to the **while** statement to execute *command-list1* again.
 - If it is non-zero, then *command-list2* is skipped and execution continues with the first statement which follows the **done** statement.

The **while** loop executes as long as *command-list1* exits with a “true” (zero) exit status; once it exits with a “false” (non-zero) status, the loop terminates and execution continues following the **done** statement.

Because *command-list1* can exit with a non-zero status the first time it executes, there is no guarantee that *command-list2*—the body of the loop—will execute even once.

It is important that *command-list1* exits with both “true” and “false” exit codes.

- If it can only exit with zero (true), then an infinite loop is created. (On an interactive command line, this can be useful because the loop can usually be stopped by pressing the interrupt key—usually `DEL` or `CTRL c`.)
- If it can only exit with non-zero (false), then the loop body never executes.

AV 8-6

The **until** Statement

```
#!/bin/sh
# Call a user as soon as he or she logs in
# Syntax: caller username

if [ $# -ne 1 ]
then
    echo "Syntax: $0 username" >&2
    exit 1
fi

until who | grep "^$1 " > /dev/null 2>&1
do
    echo '.\c'
    sleep 2
done

# The user has logged in. Allow time to complete login sequence.
sleep 30

write $1 < $MSG_FILE
```

sp0806

© The Santa Cruz Operation, Inc.

8.3 The until Statement

Like the **while** statement, the **until** construct causes a command list to repeatedly execute. Unlike **while**, which executes as long as conditions remain true, **until** executes as long as conditions remain *false*.

8.3.1 Syntax

AV 8-6: The until Statement

This slide shows a sample shell script which uses an **until** loop. The loop causes the program to wait (displaying a string of periods) until a user logs in, and then it sends a message to their screen.

NOTE: By default, write access to a terminal by others is not permitted. This program assumes that the intended recipient permitted this action by placing this statement in their start-up script (such as *.profile*):

```
mesg y
```

The basic **until** syntax is:

```
until command-list1
do
    command-list2
...
done
```

8.3.2 Execution

Execution of an **until** statement proceeds as follows:

- *command-list1* executes. The last command which executes in the list is the *relevant command* (just like the **if** and **while** statements).
- The exit status of the relevant command is examined:
 - If it is non-zero, then *command-list2* is executed and then control returns back to the **until** statement to execute *command-list1* again.
 - If it is zero (0), then *command-list2* is skipped and execution continues with the first statement which follows the **done** statement.
- The **until** loop executes as long as *command-list1* exits with a “false” (non-zero) exit status; once it exits with a “true” (zero) status, the loop terminates and execution continues following the **done** statement.

- Because *command-list1* could exit with a zero status the first time it executes, there is no guarantee that *command-list2*—the body of the loop—will execute even once.
- Like the **while** statement, it is important that *command-list1* be able to exit with both “true” and “false” exit codes.

AV 8-7

The for Statement

```
$ for USER in bonnie charlie david jeff steve sue
> do
>   write $USER << EOF
> Don't forget the meeting starts in 5 minutes. Don't be late!
> EOF
> done
```

```
$ lc
letter memo      report status
$ mv * *.txt
mv: *.txt not found
$ for FILE in *
> do
>   mv $FILE ${FILE}.txt
> done
$ lc
letter.txt memo.txt      report.txt status.txt
```

sp0807

© The Santa Cruz Operation, Inc.

8.4 The for Statement

The last of the shell looping constructs is the **for** statement. Like **while** and **until**, it causes a command list to repeatedly execute. Unlike those statements which use exit codes to determine whether a loop executes again, **for** executes as long as string values are available.

8.4.1 Syntax

AV 8-7: The for Statement

This slide shows several examples of a **for** loop as they are used on an interactive command line. The Unexpected errors exercise on page 8-28 requires the student to explain what went wrong with the **mv** command.

The *strings* can be constants or variables, such as `$*`.

The basic **for** syntax is:

```
for name in string1 string2 string3 ...
do
    command-list
...
done
```

8.4.2 Overview

The argument *name* is the identifier for a shell variable. Do not precede it with a dollar sign (\$) which refers to the *contents* of the variable.

Each *string* is a character string which will be assigned to *name*. Strings have these properties:

- They are separated by a field separator character (usually a space or tab).
- A *string* representing a filename can use shell wildcards.
- At least one *string* must follow the keyword “in”; any number may follow.
- Use a backslash (\) for line continuation if all *strings* do not fit on one line.

This exercise requires that the students remember how a command line executes. Point out that the **mv** command executes *after* the shell expands the `*` wildcards, where possible. Show what the command line looks like after this expansion by executing it in a shell that has the expand trace (`-x` option) enabled.

Exercise: Unexpected errors

Explain why these commands from AV 8-7 on page 8-26 produced an error message:

```
$ lc
letter memo report status
$ mv * *.txt
mv: *.txt not found
```

8.4.3 Execution

Execution of a **for** statement proceeds as follows:

- *name* is assigned the value of *string1* as if this statement is executed:
name=string1
- The *command-list* then executes, up to the **done** statement.
- *name* is then assigned the value of *string2* and the *command-list* executes again.
- The *command-list* executes once for each *string* provided. During each iteration, *name* takes on each value in turn (and may be used within the loop body as well).
- Following the last iteration, *name* retains the value of the last *string* it was assigned.

AV 8-8

Positional Parameters and for Loops

```
$ cat sample_for_1
#!/bin/sh
# Sample for-loop using positional parameters

echo The number of positional parameters is $#

for NAME in $*
do
    echo The next name is $NAME
done

$ sample_for_1 mary ellen pete
The number of positional parameters is 3
The next name is mary
The next name is ellen
The next name is pete

$ sample_for_1 "mary ellen" pete
The number of positional parameters is 2
The next name is mary
The next name is ellen
The next name is pete
```

sp0808

© The Santa Cruz Operation, Inc.

AV 8-9

Using \$@ in a for Loop

```
$ cat sample_for_2
#!/bin/sh
# A better sample for-loop using positional parameters

echo The number of positional parameters is $#

for NAME in "$@"
do
    echo The next name is $NAME
done

$ sample_for_2 mary ellen pete
The number of positional parameters is 3
The next name is mary
The next name is ellen
The next name is pete

$ sample_for_2 "mary ellen" pete
The number of positional parameters is 2
The next name is mary ellen
The next name is pete
```

sp0809

© The Santa Cruz Operation, Inc.

8.4.4 The \$@ variable: revisited

AV 8-8: Positional Parameters and for Loops

This slide shows a first attempt to access all the positional parameters in a **for** loop. It misbehaves on the second invocation attempt because the space in \$1 (mary ellen) could not be protected.

AV 8-9: Using \$@ in a for Loop

This slide corrects the problem from the previous slide by using the \$@ variable in double quotes. Show also that the **for** loop could omit the `in strings` list and it would mean the same thing.

In a previous lesson you learned about the two shell special variables, \$* and \$@, which hold all of the shell's positional parameters. These variables are commonly used as *string* values in **for** loops.

- \$* refers to \$1 \$2 \$3 \$4 ...
- \$@ refers to \$1 \$2 \$3 \$4 ...

Although nearly identical, there is a distinction between the two variables *when quoted*:

- "\$*" refers to "\$1 \$2 \$3 \$4 ... "
- "\$@" refers to "\$1" "\$2" "\$3" "\$4" ...

Because "\$@" is frequently used with **for** loops—and is usually safer to use than \$*—a shorthand notation is available. These two code fragments are equivalent:

```
for name in "$@"
do
    ...
done

for name
do
    ...
done
```

This exercise gives students more practice with using **for** loops and requires them to remember how to use quotes to solve problems.

Exercise: Using \$@ in a for loop

Using the program in AV 8-9, explain the output you get from entering this command line:

```
$ sample_for_2 "mary      ellen" pete
```

How can you modify the program to avoid the problem?

AV 8-10

The **break** and **continue** Statements

```
#!/bin/sh
# Rename files with standard extensions
# Syntax: rename file(s)

for FILE
do
    echo "With which extension do you want to rename $FILE?"
    \tt) \".txt\" \td) \".doc\" \tr) \".rpt\"
    \tn) no extension? (q to quit) ==> \c"
    read REPLY DUMMY

    case "$REPLY" in
        [tT] )    EXT=txt ;;
        [dD] )    EXT=doc ;;
        [rR] )    EXT=rpt ;;
        [nN]* )   continue ;;
        [qQ]* )   break ;;
        * )       echo "Invalid response. Bye!" >&2
                  exit 1 ;;
    esac

    mv "$FILE" "$FILE.$EXT" && echo "$FILE renamed to $FILE.$EXT"

    LIST="$LIST $FILE.$EXT"
done

echo "Summary: These files now have new names:"
ls -l LIST
```

sp0810

© The Santa Cruz Operation, Inc.

8.5 Loop Commands

AV 8-10: The break and continue Statements

This slide shows a sample program that uses **break** and **continue** statements within a loop. The program renames a list of files with confirmation from the user. The **continue** statement skips only the current file; **break** skips the rest of the filenames in the list.

A set of built-in commands is available which affects the execution of any of the shell looping constructs (**while**, **until**, or **for**). These commands are **break** and **continue**.

Because you will not want these commands to *always* affect a given loop, they will nearly always be used in a conditional context (such as an **if** statement). Note that they are *used* conditionally, but they *control* a loop.

8.5.1 The break statement

The **break** statement terminates the current loop or, if nested, the innermost *n* loops. This causes the command which follows the next **done** statement to execute (or *n* **done** statements away).

The syntax is:

```
break  
break n
```

If *n* is omitted, 1 (one) is assumed.

This command can be used to terminate an otherwise infinite loop.

It might be useful to write some pseudo code on the board that shows some nested loops to demonstrate how the argument to **break** works. For example:

```
while condition
do
    ...
    for VAR in s1 s2 s3 ...
    do
        ...
        break n
    ...
done
statement # jump to here if n==1
done
statement # jump to here if n==2
```


AV 8-11

The **break** and **continue** Statements

```
#!/bin/sh
# Rename files with standard extensions
# Syntax: rename file(s)

for FILE
do
    echo "With which extension do you want to rename $FILE?"
    \tt) \".txt\" \td) \".doc\" \tr) \".rpt\"
    \tn) no extension? (q to quit) ==> \c"
    read REPLY DUMMY

    case "$REPLY" in
        [tT] )    EXT=txt ;;
        [dD] )    EXT=doc ;;
        [rR] )    EXT=rpt ;;
        [nN]* )   continue ;;
        [qQ]* )   break ;;
        * )       echo "Invalid response. Bye!" >&2
                  exit 1 ;;
    esac

    mv "$FILE" "$FILE.$EXT" && echo "$FILE renamed to $FILE.$EXT"

    LIST="$LIST $FILE.$EXT"
done

echo "Summary: These files now have new names:"
ls -l LIST
```

sp0811

© The Santa Cruz Operation, Inc.

8.5.2 The continue statement

The **continue** statement causes the remaining statements in the current loop (or, if nested, the innermost n loops) to be skipped.

The syntax is:

```
continue
continue  $n$ 
```

If n is omitted, 1 (one) is assumed.

For a **while** or **until** loop, this causes *command-list1* to execute. If it exits with a zero exit status (non-zero for **until**), the loop runs again; otherwise the loop terminates and execution continues with the statement that follows the closing **done** statement.

For a **for**-loop, this causes *name* to be assigned the next *string* value and the loop body runs again. If there are no more *strings*, the loop likewise will terminate.

It might be useful to write some pseudo code on the board that shows some nested loops to demonstrate how the argument to continue works. For example:

```
while condition
do
    ...
    for VAR in s1 s2 s3 ...
    do
        ...
        continue n
        ...
    done # skip to here if n==1 and begin next for-loop
        # iteration
    ...
done # skip to here if n==2 and begin next while-loop
    # iteration

statement
```

AV 8-11: The break and continue Statements

AV 8-10 is repeated to aid students in completing the following exercise.

The exercise helps to reinforce the idea of performing error checks throughout programs to make them more user friendly. A simple **if** statement at the beginning of the program can accomplish this.

Exercise: Using break and continue

In the **rename** program in AV 8-11, what happens if a user enters no command-line arguments? What might you do to improve the program in this matter?

8.6 Lesson Summary

The shell has three programming constructs which allow a command list to execute repeatedly. Two of them—**while** and **until**—execute the loop based on the exit status of a command list: “while” the status is zero or “until” the status is zero. The third construct—**for**—executes once for each string value provided.

These looping constructs are also controlled by the **break** and **continue** statements which terminate the current loop (**break**) or skip the remaining statements in the current iteration of the loop (**continue**).

Review questions

1. How can you program a loop to execute exactly 250 iterations?

2. Explain the error message from this command:

```
$ expr 5 * 6
expr: syntax error
```

How should the problem be corrected?

3. What is the difference between the **while** and **until** statements?

4. How can you program an infinite loop?

5. What determines how many times a **for** loop executes?

6. What does the shell assume when the “`in string ...`” clause is omitted in a **for** statement?
7. What is the difference between the `$*` and `$@` shell special variables?
8. What is the difference between the **break** and **continue** statements? In what context are they normally used?

8.7 Lab: Looping Constructs

This lab will give you experience with the shell's looping constructs.

8.7.1 Phone database: revisited

Modify the Phone Database program from your previous lab to repeatedly display the menu (and execute the choice) until the user indicates to quit.

8.7.2 File tally

This program is not terribly complicated to write; students should have little trouble completing it.

There is one “gotcha” in this lab. Probably the easiest way to determine the file type is to use a series of **if/elif** statements. It is important that the check for symbolic links be done before the other file types, otherwise **test** will follow the link and report *that* file type instead. Note that even though symbolic links must be tested first, there's no reason that the program output must list them first, too. For example:

```
$ tally *
Ordinary Files      8
Directories        4
Symbolic Links      1
Block Device        0
Char. Device        0
Other Special       0
-----
Total Files         13
```

Write a program called *tally* which examines the files specified as command-line arguments and displays a total for each common UNIX file type.

Display a histogram for each file type in addition to the count.

Note: Test for symbolic links first so the test doesn't follow the link and report that file type.

8.7.2.1 Sample usage

```
$ tally *

Ordinary Files      8      *****
Directories        4      ****
Symbolic Links      1      *
Block Device        0
Char. Device        0
Other Special       0
-----
Total Files         13
```

8.8 Relevant Commands

break	Exits from the enclosing for , while , or until -loop, if any. If <i>n</i> is specified, it breaks <i>n</i> levels.
continue	Resumes the next iteration of the enclosing for , while , or until -loop. If <i>n</i> is specified, it resumes at the <i>n</i> -th enclosing loop.
expr	Evaluates arguments in an expression
for	Executes <i>command-list</i> once for each <i>string</i> value provided
until	Executes <i>command-list2</i> as long as <i>command-list1</i> exits with a non-zero exit status
while	Executes <i>command-list2</i> as long as <i>command-list1</i> exits with a zero (0) exit status

8.9 Answers to Review Questions

1. To program a loop to execute exactly 250 iterations, you could use a for loop with 250 strings specified, but that would be very tedious to code. An easier approach would be to use a while loop with a counter variable. For example: `COUNTER=0 while [ $COUNTER -lt 250 ] do ... COUNTER=`expr $COUNTER + 1` ... done`
2. The error message occurred because the shell expanded the * wildcard to filenames before the `expr` command executed. These filenames were unexpected by `expr`, so it displayed an error message. The correction to this problem is to quote the * character. (See “Exercise: Unexpected errors,” on page A-49 of this appendix for a similar problem.)
3. A while statement executes the body of a loop as long as a command-list exits with a true (0) exit status. An until statement executes the body of a loop as long as a command-list exits with a false (non-zero) exit status.
4. You can easily program an infinite loop with a while or until statement, used in conjunction with the true, false, or null commands: `while true while : until false do do do ... .. done done done`
5. A for loop executes once for each string specified after the “in” keyword. If this keyword is omitted, “in “\$@”” is assumed, therefore the loop executes \$# times.
6. When the “in string ...” clause is omitted in a for statement, the shell assumes “in “\$@””.
7. The difference between the \$* and \$@ shell special variables is apparent only when the variable is quoted: \$* refers to \$1 \$2 \$3 \$4 ... \$@ refers to \$1 \$2 \$3 \$4 ... "\$*" refers to "\$1 \$2 \$3 \$4 ... " "\$@" refers to "\$1" "\$2" "\$3" "\$4" ...
8. The break statement terminates the current loop or, if nested, the innermost n loops. This causes the command which follows the next done statement to execute (or n done statements away). The continue statement causes the remaining statements in the current loop (or, if nested, the innermost n loops) to be skipped. Because these statements should not always execute, they are used in conditional contexts (e.g., in an if or case statement) to control a loop construct.

9

Shell Functions

Instructor Guide

Instructor Guide

Lesson 9

Shell Functions

Suggested lesson duration

This lesson should take approximately 45 minutes to complete, including the lab.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 9-1

Instructor Guide

Shell Functions

- Describe the execution of a shell function
- Pass arguments to a shell function
- Use the **return** statement

sp0001 © The Santa Cruz Operation, Inc.

Introduction

Program code that is frequently accessed need not be physically duplicated in your program. If this common code is *repeatedly* accessed, a loop can be used to avoid duplication, but this approach does not work for code that is accessed only *occasionally*.

A different approach might be to place the common code in a separate command file (a “sub-program”) and call it as needed. A shell program may certainly call another shell program and, as expected, the subprogram will execute in a subshell. This approach can have unwanted consequences, however.

- The subprogram will be unable to affect the main program’s environment (e.g., variables, working directory).
- The subprogram will have no access to variables unless they have been exported by the main program (or by another ancestor).
- Program efficiency may decline because the system must go to the expense of creating a new process to execute the subprogram.

A third approach is to use a *shell function*, which automatically gets executed by the current shell. This lesson describes how to write and use functions within your programs.

Objectives

This lesson describes the shell function construct, and how to write and use them. By the end of this lesson, you will be able to:

- Describe the execution of a shell function
- Pass arguments to a shell function and use them within the function body
- Use the **return** statement to exit a function

AV 9-1: Shell Functions

This first slide describes the objectives covered in this lesson.

AV 9-2

Shell Functions

```
$ WHO ()  
> {  
>     who | sort  
> }  
  
$ who  
jeff      tty02      Oct 19 09:46  
donna     tty04      Oct 24 09:11  
steve     tty02      Oct 05 13:43  
bonnie    tty03      Oct 19 08:19  
donna     tty09      Oct 24 09:11  
  
$ WHO  
bonnie    tty03      Oct 19 08:19  
donna     tty04      Oct 24 09:11  
donna     tty09      Oct 24 09:11
```

sp0002 © The Santa Cruz Operation, Inc.

9.1 Shell Functions

A shell function is a body of code that is run directly by the current shell. This code could be placed in a command file and called as a shell script, but doing so would cause it to execute in a subshell.

9.1.1 Overview

Because functions do not execute in a subshell, they have several advantages over standard scripts:

- They can affect the current shell's environment, such as the current directory and values contained in user variables (a function cannot affect the positional parameters, however).
- They can access unexported variables of the current shell.
- They run faster because a process does not need to be created to execute the code; the current shell executes it.

9.1.2 Syntax

AV 9-2: Shell Functions

This slide shows a sample shell function that is declared on an interactive command line.

The general syntax of a shell function is:

```
name ()
{
    command-list
    ...
}
```

where *name* is an identifier which uses the same syntax rules of formation as shell variables.

The space after *name* and the newline after the parentheses [*()*] are optional. The newline before the right curly brace [*}*] is required unless the brace is preceded by a semicolon [*;*].

The *command-list* is either a single command, a list of commands, or a pipeline. Although not required, it is conventional to indent *command-list* to improve the readability of the construct. This sequence only defines the function. The code does not execute until the function is called (interactively or from a shell script).

AV 9-3

Viewing Functions

```
$ set
HOME=/usr/steve
..
TERM=ansi
WHO(){
who | sort
}

$ type WHO
WHO is a function
WHO(){
who | sort
}
```

sp0903

© The Santa Cruz Operation, Inc.

AV 9-4

Exporting Functions

```
$ cat func_test
:
# test whether the function "WHO" is available
echo "About to call WHO..."
WHO

$ func_test
About to call WHO...
bonnie      tty3      Oct 19 08:19
donna       tty4       Oct 24 09:11
donna       tty9       Oct 24 09:11
jeff        tty02      Oct 19 09:46
steve       tty2       Oct 05 13:43

$ sh func_test
About to call WHO...
func_test: WHO: not found
```

sp0904

© The Santa Cruz Operation, Inc.

9.1.3 Viewing and removing functions

AV 9-3: Viewing Functions

This slide shows shell functions embedded in the output of the **set** command as well as the use of **type** to view function information. Note that the **type** command does not produce the same output in the Bourne and Korn shells. If running **ksh**, the content of the function is not displayed.

Use the built-in **set** command to view which functions are currently set and what code they contain. The functions will be shown with the shell variables that **set** also displays.

To view a particular function, you can use the built-in **type** command, which can identify a command as a function and display its code.

Use the built-in **unset** command to remove any unwanted functions.

9.1.4 Exporting functions

AV 9-4: Exporting Functions

This slide demonstrates the one difference between executing a script explicitly rather than implicitly—whether shell functions are made available to the subshell.

Note also that functions will not be available if a script uses the hashpling mechanism, whether it is invoked explicitly or implicitly.

You cannot use the **export** command to cause a shell function to be placed in the environment of subsequent subshells.

Functions are automatically placed in the subshell's environment if it is *implicitly* created. In other words, if *all* of these conditions are true, then the current shell's functions are available for use within the subshell:

- The script file has execute (“x”) permission.
- It does *not* use the hashpling mechanism to explicitly identify the shell which will execute the script:

```
:
# This script begins with the null command
# instead of a comment, so the Bourne shell
# will execute it.
```

- It is invoked with its name alone:

```
$ name
```

Functions are *not* placed in the subshell's environment if it is *explicitly* created. In other words, if either of these conditions is true, then the current shell's functions will *not* be available for use within the subshell:

- It is invoked with the **sh** command:

```
$ sh name
```

- Or it uses the hashpling mechanism to explicitly identify the shell which will execute the script:

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

9.1.5 Shell functions and the dot (.) command

The built-in dot (.) command causes the current shell to read and execute commands in a named script file.

The script file need not have “executable” statements; it may contain only variable and function definitions.

You can create *function libraries* for those functions which you use in many applications.

- Each library file contains only shell variable and function definitions.
- Your shell script can use the functions in the library by reading the file with the dot command.

This exercise should help students realize that they can create their own functions for use in their interactive environment. They should already be aware of the shell start-up file, *.profile*.

Exercise: Shell functions

Where can you define shell functions so that they are available in your login shell?

AV 9-5

Function Arguments

```
$ CDP () { cd $1; pwd; }
$ pwd
/usr/steve/bin

$ CDP ../../lib
/usr/lib

$ del ()
> {
> echo Trying to delete $*
> rm -i "$@"
> }

$ del [lmr]*
Trying to delete letter memo report
letter: ? n
memo: ? y
report: ? n

$ set arg1 arg2 arg3
$ echo $*
arg1 arg2 arg3
```

sp0905

© The Santa Cruz Operation, Inc.

9.2 Passing Arguments

AV 9-5: Function Arguments

This slide shows some shell functions that use parameters for arguments.

Arguments can be used with shell functions just as they can with ordinary shell scripts:

- Arguments come into the function as positional parameters (\$1, \$2, \$3, and so on).
- If no command-line arguments are provided with the function call, then the positional parameters in the function are set to null strings.

NOTE: The original positional parameter values of the calling environment are inaccessible in the body of the function—even if no command-line arguments are provided with the function call.

Upon return from the function, the positional parameters are restored to their original values.

GuruNotes

In XENIX, the positional parameters are *not* restored to their original values upon return from a function. It is necessary to save the values before calling a function and restore them later. For example:

```
OLDPARAMS="$*" # Save old values
function_call
set $OLDPARAMS # Restore old values
```

Note that this could be problematic if, say, the original \$2 value was "mary ellen". The space cannot be protected when restoring the values—quotes around \$OLDPARAMS would also protect the spaces between \$1 and \$2.

Of course, the parameters could be saved and restored with these statements:

```
OLD1="$1" # Save old $1 value
OLD2="$2" # Save old $2 value
OLD3="$3" # Save old $3 value
function_call
set "$1" "$2" "$3" # Restore old values
```

This approach, however, does not lend itself well to programs which can accept an arbitrary number of parameters. The above example assumes that only three parameters are provided and need to be saved.

AV 9-6

Using the return Statement (Part 1)

```
$ cat myfunctions
getyn()
{
while echo "${*}? [y/n] ==> \c" >&2
do
    read REPLY DUMMY
    case "$REPLY" in
        [yY]* ) return 0 ;;
        [nN]* ) return 1 ;;
        * ) echo "\07Please enter <y> or <n>..." ;;
    esac
done
}

abort()
{
    # set default error message"
    [ $# -eq 0 ] && set "Aborting program. See ya!"
    echo "\07*** $" >&2
    exit 2
}
```

sp0906

© The Santa Cruz Operation, Inc.

AV 9-7

The return Statement (Part 2)

```
$ cat rm+mkdir
#!/bin/sh
# Remove a directory file and replace it with a new one
# Syntax: $0 dirname

. myfunctions

if getyn "Delete $1 file?"
then
    if getyn "Cannot be undone. Are you sure?"
    then
        rm -r $1
    else
        abort
    fi
fi

echo "Please enter new name for directory file ==> \c"
read NAME DUMMY
getyn "Is $NAME correct?" || abort "Please try again another time..."
echo "$NAME" | grep '[A-Z]' && abort "Please use lower case characters only"

mkdir "$NAME"
```

sp0907

© The Santa Cruz Operation, Inc.

9.3 The return Statement

9.3.1 Overview

Recall that a shell script can be terminated with the built-in **exit** command.

The **exit** command can also be used in a shell function, but it must be used with care. Because functions are executed by the current shell, **exit** terminates the current shell as well as the function.

To terminate a shell function and *not* the shell, use the built-in **return** command. Like **exit**, **return** terminates with an exit (or “return”) status in the range 0-255.

9.3.2 Syntax

The syntax for **return** is:

```
return [n]
```

If *n* is omitted, \$? is assumed.

Use *n*=0 for successful termination and a non-zero value in the range 1-255 to indicate failure.

Like scripts, you should terminate all functions with a known return status rather than letting it “fall off the edge” and terminate implicitly at the end.

AV 9-6: Using the return Statement (Part 1)

This slide shows the contents of the file *myfunctions*, which contains definitions for two functions: **getyn** and **abort**.

AV 9-7: The return Statement (Part 2)

This slide shows a program that uses the **getyn** and **abort** functions. Note the use of the dot (.) command to read in the file *myfunctions*.

The **getyn** function is a commonly-used function that ensures that a user answers a yes-or-no question with an appropriate response (something beginning with “y” or “n”). Any other response causes the question to be asked again. The **while** loop is an infinite loop because **echo** never fails; the loop (and function) exit when the **return** statement executes. Because the function returns with different values, depending on the response, it is intended to be used in a conditional context.

For example:

```
if getyn "Continue?"
then
    ...
else
    ...
fi
```

or, using the shorthand notation:

```
getyn "Continue?" || exit 1
```

The **abort** function simply terminates the shell by first displaying an error message (with a bell).

9.4 Lesson Summary

A shell function is similar in nature to a shell script, but differs in that:

- It is executed by the current shell rather than a subshell.
- It has direct access to all variables in the current shell environment, whether or not they are exported.
- It can modify variables from the current shell environment.

Review Summary

1. What is the syntax for defining a shell function?
2. How is a shell function called?
3. How can you view or delete a shell function?
4. Can you export a shell function? Explain.
5. How do you use a *function library*?
6. How do you pass arguments to a shell function? How does the function access the arguments?

7. What is the difference between the **exit** and **return** commands?

9.5 Lab: Shell Functions

This lab gives you experience with shell functions in programs and in your login environment.

9.5.1 Everyday functions

Depending upon the hardware available, students may not be able to use the **setcolor** command.

Create a file called `$HOME/.sh_funcs` that contains functions that act like aliases for commands you frequently execute. Set up your `.profile` to automatically read the functions defined in this file. Some example functions might include:

c	Clears the screen
lff	Does a -CF directory listing on a given directory(s) and pipes the output to more
sc	Sets your screen colors and cursor size to your preferences (hardware permitting)
CD	Changes your working directory and your prompt to a given directory

9.5.2 Phone database: revisited

Incorporate into your phone database programs the **getyn** function that was shown earlier in this lesson. It should handle all of the yes/no questions that are asked of the user.

9.5.3 File tally: revisited

Rewrite the file tally program from Section 8.7 on page 8-43 using a function to generate the histograms.

9.6 Relevant Commands

exit	Terminates the current shell
return	Terminates the current shell function
set	Displays the variables and functions set in the current shell. Also sets options (flags) to the current shell and sets positional parameters.
unset	Deletes variables and functions in the current shell

9.7 Answers to Review Questions

1. The syntax for defining a shell function is: `name () { command-list ... }` The left curly brace (`{`) may also appear on the first line after the right parenthesis (`)`. Spaces after name are optional.
2. A shell function is called the same way as you invoke a shell script: on a command line, specify the name of the function (or shell script) and provide it with any needed command-line arguments.
3. Use the `set` command to view a shell function. Use the `unset` command to delete one.
4. You cannot use the `export` command with a shell function. However, it is not needed because shell functions are automatically made available to subshells unless one of the following is true: The hashplng mechanism, which specifies the shell that will execute the script, is in use The subshell for the script was explicitly invoked with the `sh` command
5. A “function library” is a text file that contains the code for your functions. It can be read by the current shell with the `dot` (`.`) command.
6. Pass arguments to a shell function exactly the same way as you pass them to a script: on a command line, specify the name of the function (or shell script) and provide it with any needed command line arguments. The function accesses these arguments with positional parameters.
7. The difference between the `exit` and `return` commands is that `exit` terminates the current shell, whereas `return` only terminates the current shell function.

10

Shell Signal Handling

Instructor Guide

Instructor Guide

Lesson 10

Shell Signal Handling

Suggested lesson duration

This lesson should take approximately 45 minutes, including the lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 10-1

Shell Signal Handling

- Define the term signal
- Identify sources of signals
- Use the shell trap command

sp1001 © The Santa Cruz Operation, Inc.

Instructor Guide

Introduction

When asynchronous events (such as external interrupts from devices) occur on the system, processes can be signalled by the kernel. Typically, when this happens, the process which receives the signal is terminated.

Because this is not always a desirable outcome, programs can use system calls to trap certain signals in order to ignore them or perform some action other than immediate termination. Some of this functionality is available in a shell program, too, with the **trap** command. This chapter discusses the general nature of signals and how you can deal with them in your shell programs.

Objectives

This chapter describes signals and the built-in **trap** command. By the end of this chapter, you will be able to:

- Define the term *signal*
- Identify sources of signals
- Use the shell **trap** command

AV 10-1: Shell Signal Handling

This first slide describes the objectives covered in this lesson.

10.1 Asynchronous Events

An asynchronous event is one whose timing cannot be predicted. As a result, one cannot know:

- When they will occur
- Which process will be executing
- Where in the code the process will be executing

10.1.1 Overview

From a hardware perspective, asynchronous events are:

- Interrupts from external devices
- Exceptions, traps, and faults from the CPU

From a software perspective, the kernel handles the asynchronous event. This handling may involve:

- Retrieving or sending data to an external device
- Allocating or retrieving pages of memory
- Determining which process is responsible for the event and then informing it that an event has occurred
- Panicking the system under severe circumstances

The kernel informs a process that an event has occurred by sending the process a *signal*.

10.1.2 Signals

A *signal* is a software flag that tells a process that some asynchronous event has occurred. The kernel can send different signals to indicate that different events have occurred; each signal is associated with a number.

GuruNotes The kernel sends a signal by setting one bit in the `p_sig` field of the process' Process Table entry. There is a different bit for each signal.

Because there is only one bit (literally!) of information for a given signal, the kernel cannot keep track of:

The number of times a given signal has been sent.

When it was sent in relation to other signals that have also been sent.

Some of the predefined signals and their numbers are:

<i>Name</i>	<i>#</i>	<i>Description</i>
—	0	Shell exit
SIGHUP	1	Hangup
SIGINT	2	Interrupt (rubout)
SIGQUIT	3	Quit
SIGILL	4	Illegal instruction
SIGKILL	9	Kill (cannot be caught or ignored)
SIGTERM	15	Software termination signal from kill

These values are defined in `/usr/include/sys/signal.h`. Technically, “signal” 0 (shell exit) is not a true system signal (it is not defined in `/usr/include/sys/signal.h`), but we will see that the shell can trap this pseudo-signal just like the real signals.

The file `/usr/include/sys/signal.h` is a C language header file that is used by system and application programmers. Although it contains C code, it also contains comment lines which should be reasonably understandable to a novice programmer.

A process commonly terminates upon receipt of most signals.

Some signals—such as SIGQUIT (3)—usually cause a *core* file to be generated when the process terminates.

The core file contains a memory dump of what the process was doing just before it terminated. Without knowing how to use a debugger, this file is usually of little consequence to most users on the system and should be removed to conserve disk space.

SIGKILL (9) is different from most other signals because a process cannot trap or ignore it—it nearly always causes the default action of immediate process termination. Termination occurs right away only if the process is in a state in which it can act upon the signal. Some processes may not be in this state, so their termination is delayed—sometimes indefinitely. Further explanation of this phenomenon is beyond the scope of this course.

GuruNotes

Processes act upon a signal as they return from kernel mode (also called system mode) back to user mode. Normally a process is in user mode, accessing its own code and data, but it enters kernel mode when it makes a system call or when an interrupt or exception occurs.

A process running in kernel mode is not subject to preemption (where it is *required* to give up the CPU for another process)—preemption occurs as the process returns to user mode. A process may, however, choose to *voluntarily* give up the CPU while waiting for an event to occur. This is called *sleeping* and a priority is associated with it. The process is awakened by an interrupt handler or other kernel code typically.

If a signal is sent to a process that is sleeping, the process will wake up and deal with the signal if it is sleeping at a relatively low priority. A process sleeping at a high priority will not be awakened by a signal—even by a SIGKILL (9) signal. It must be awakened before the signal will be dealt with.

10.2 Generating Signals

Although asynchronous in nature, signals can be generated in a more synchronous manner. Some signals are generated from keyboard interrupts and others can be generated with the **kill** command.

10.2.1 Keyboard signals

When you press a key on the keyboard, an interrupt is generated which causes the kernel to retrieve some data from the device—namely, which key was pressed.

The kernel normally passes this information directly to the process so that it can use it, but frequently, the kernel interprets the data before sending it to the process. This occurs when the keyboard interface is in *canonical mode*; the kernel does only minimal preprocessing when in *raw mode*.

Canonical mode is sometimes called “cooked mode.”

Some canonical mode processing includes the interpretation of characters or character sequences. Some of the commonly-used key sequences are:

	Allows command line editing by discarding the previous character
 u	Discards the current line of the standard input
 d	Indicates end-of-file for the standard input
	Sends a SIGINT (2) signal to the foreground process that is using the keyboard
 \	Sends a SIGQUIT (3) signal to the foreground process that is using the keyboard

These may differ depending on your operating system and the shell you are using. For example, SIGINT is generated using  **c** on some systems. Use **stty** to view and reassign the character sequences.

Refer students to the **stty** man pages for more information on reassigning character sequences.

10.2.2 The kill command

The **kill** command sends a signal to one or more processes. The syntax is:

```
kill PID ...  
kill -n PID ...
```

where

PID is the process identification number of the process you want to signal; several processes can be specified on the command line

-n is the signal to be sent; by default, **kill** sends the **SIGTERM** (15) signal

Use the **ps** command to determine the PID of a process. Note that unless you are **root**, you must be the owner to signal a process using **kill**.

The following command lines are equivalent:

```
$ kill 7312  
$ kill -15 7312
```

The following command lines are *not* equivalent:

```
$ kill 7312  
$ kill -9 7312
```

If you intend to terminate a process, use caution when sending the **SIGKILL** (9) signal. This is because a process is unable to trap it (perhaps to perform some cleanup activity before termination). Use other signals first and if that is unsuccessful at stopping the process, then use **SIGKILL** as a last resort.

10.3 The trap Command

The shell has a built-in **trap** command which is used to handle signals within a shell script.

10.3.1 Syntax

The general syntax of the **trap** command is:

```
trap command-list n ...
```

The *command-list* is a single command-line argument that **trap** executes when the shell subsequently receives signal(s) *n*.

- In other words, **trap** refers to its \$1 parameter for the *command-list* that it will execute.
- Quotes are usually needed around *command-list*, especially when it contains multiple words.

If *n* is 0, the *command-list* is executed on exit from the shell.

Note the use of pseudo-signal 0, which indicates that the *command-list* is to be executed upon exit from the shell.

With no arguments, the **trap** command prints the *command-list* associated with each signal number.

The *command-list* is parsed once when the trap is set and once when the trap is taken.

AV 10-2

Catching Signals

```
#!/bin/sh
# Template which shows characteristics of trap command

# At this point, any incoming signal has the default action
...
# Now set a trap for the SIGINT (2) signal
trap 'rm tmp$$; exit 1' 2
# Also set a "cleanup" trap
trap 'rm tmp$$; exit 0' 0
...
# Critical code coming up which should not be interrupted.
# Completely ignore the SIGINT (2) signal
trap '' 2
...
# Out of critical section. It no longer matters what happens if
# SIGINT (2) is sent. Reset signal to system default action.
trap 2
...
# Now at end of program. The trap for "signal" 0 is still in
# effect, so the temporary file will be automatically removed now.
exit 0
```

sp1002

© The Santa Cruz Operation, Inc.

10.3.2 Catching signals

AV 10-2: Catching Signals

This slide shows the skeleton of a shell script where the **trap** command is used several times to handle signals. The comments on the slide explain each **trap** command line.

Use the **trap** command to catch a signal so that if it is subsequently sent to this process, the default action (usually termination) does not take place. In this example, upon receipt of SIGINT (2), SIGQUIT (3), or SIGTERM (15), the shell removes a file and then voluntarily terminates:

```
trap 'rm /tmp/temp$$; exit 1' 2 3 15
```

In this example, the shell removes a file and *continues executing* after receiving SIGINT (2) or SIGQUIT (3):

```
trap 'rm /tmp/temp$$' 2
```

10.3.3 Ignoring signals

You can also use the **trap** command to cause a signal to be ignored by the shell and by the commands it invokes. To ignore a signal, simply specify a null *command-list*.

In this example, the shell ignores SIGINT (2) or SIGQUIT (3) and continues execution:

```
trap '' 2 3
```

To ignore a signal, you can also specify the null command (:) for *command-list*:

```
trap : 2 3
```

10.3.4 Resetting signals

Once you pass a critical section of code, you might not care if special actions need to take place upon receipt of a signal—the default action might be sufficient.

To reset a signal to the system's default action (usually termination), simply omit the *command-list*.

In this example, the shell handles the SIGINT (2) or SIGQUIT (3) with the default action (in this case, termination):

```
trap 2 3
```

10.4 Lesson Summary

Asynchronous events—events whose timing cannot be predicted—occur all the time on the system. The source of these events can be external or internal to the CPU, but the kernel handles the situation. Sometimes, handling will involve informing a process that an event has occurred by sending it a signal.

For most signals, the default action that occurs upon receipt is process termination, but a process does have the ability to trap the signal and do something else. A shell script does this with the built-in **trap** command which can catch a signal, ignore it, or reset it back to the system default action.

Review questions

1. What is an asynchronous event? What causes them?
2. What are some things the kernel does to handle an asynchronous event?
3. What is a *signal*?
4. What is special about the SIGKILL (9) signal? How can you send it to a process? When should it be used?
5. How can you usually generate the SIGINT (2) and SIGQUIT (3) signals? How can you reconfigure this?
6. What does the **trap** command do?

10.5 Lab: Shell Signal Handling

This lab gives you experience with sending and trapping signals within your shell programs.

10.5.1 Logoff cleanup

Modify your login environment so that the screen is automatically cleared every time you logoff.

Review the use of pseudo-signal 0, which indicates that a *command-list* is to be executed upon exit from the shell.

10.5.2 Phone database: revisited

Modify your Phone Database programs from a previous lab so that they ignore the following signals:

SIGINT

SIGQUIT

SIGTERM

10.6 Relevant Commands and Files

Relevant Commands

kill	Sends a signal to one or more processes
stty	Sets terminal characteristics, including the mapping of character sequences for generating signals
trap	Shell built-in command which is used to affect signal handling within a shell script

Relevant Files

<i>/usr/include/sys/signal.h</i>	Header file which defines the signals on the system
----------------------------------	---

10.7 Answers to Review Questions

1. An asynchronous event is one whose timing cannot be predicted. These events are caused by: Interrupts from external devices; and exceptions, traps, and faults from the CPU.
2. Some things the kernel does to handle an asynchronous event include: Retrieving or sending data to an external device Allocating or retrieving pages of memory Determining which process is responsible for the event and then informing it that an event has occurred Panicking the system under severe circumstances
3. A signal is a software flag that tells a process that some asynchronous event has occurred.
4. The SIGKILL (9) signal is special in that the kernel will not allow a process to trap or ignore it. You can send it to a process with the kill command. You should do this only as a last resort, however. Send other signals first so the process can trap them and perform cleanup work if needed.
5. You can usually generate the SIGINT (2) signal from the keyboard by pressing `DEL`, and the SIGQUIT (3) signal by pressing `CTRL \`. Use the stty command to reconfigure this.
6. The trap command allows a shell to ignore, trap (intercept), or reset signals.

11

Efficiency Considerations

Instructor Guide

Instructor Guide

Lesson 11

Efficiency Considerations

Suggested lesson duration

This lesson should take approximately 30 minutes, including the lab exercise.

Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV 11-1

Efficiency Considerations

- Use the **time** command
- Redefine the `PATH` variable
- Use absolute and relative pathnames effectively
- Identify the shell's built-in commands
- Identify efficient use of pipelines

sp1101 © The Santa Cruz Operation, Inc.

Instructor Guide

Introduction

A shell program has a big advantage over a binary program with the same functionality—easier development. For a given task, you can typically write a shell program in a fraction of the time it would take to write a similar program in a compiled language such as C.

The price for this ease of development is speed of execution. Shell programs usually run much slower than binary programs, primarily because more processes are created when they run. This slower speed might make shell programs unacceptable for some applications.

A good programmer can reduce this impact of inherently slower code by writing more efficiently. This lesson discusses several strategies that you can employ to improve the performance of your programs.

Objectives

This lesson discusses techniques to improve the efficiency of a shell program. By the end of this lesson, you will be able to:

- Use the **time** command to gauge command efficiency
- Identify situations where the **PATH** variable should be reset
- Distinguish absolute pathnames from relative paths and identify when each should be used
- Identify the shell's built-in commands
- Identify efficient means to send data through pipelines

AV 11-1: Efficiency Considerations

This first slide describes the objectives covered in this lesson.

AV 11-2

The `time` Command

```
$ time whoami > output

real    1.1
user    0.2
sys     0.2

$ time format /dev/rfd0

real    2:26.12
user    0.06
sys     0.74

$ time tar cv6L /etc/default > tar.log

real    6.67
user    0.06
sys     0.74

$ time (find /etc/default | cpio -ocvBL -O/dev/fd0 > cpio.log)

real    4.08
user    0.01
sys     0.07
```

sp1102

© The Santa Cruz Operation, Inc.

11.1 The time Command

One measure of program efficiency is speed of execution. When attempting to improve this aspect, you will find the **time** command useful in measuring how successful you are.

11.1.1 Syntax

AV 11-2: The time Command

This slide shows sample usage of the **time** command. Note that it writes to the standard error and not the standard output.

The last two examples on the slide compare using **tar** with **cpio**. Because **cpio** needs to get a file list which usually comes in from a pipeline from **find** or another command, the file list generation time must be included for a fair comparison. To time this, the **find/cpio** command line is grouped by parentheses.

Another way to time complex command lines is to create a file (script) with the command line and run:

```
time sh command-file
```

The syntax of the command is:

```
time command
```

All times are reported in seconds.

The Korn shell (**ksh**) and C shell (**csh**) have a built-in version of the **time** command which has the same syntax, but slightly different output.

The output data is influenced by current system load. The same *command* could produce varying execution times when run at different instances.

11.1.2 Overview

The **time** command executes a command line and then displays this information to the standard error:

- The time elapsed during the command (**real**) is measured.
- The time spent executing the command in user mode (**user**) is measured. In user mode, an application executes its own code.
- The time spent executing the command in system mode (**sys**):

- In system mode, an application executes kernel code.
- Kernel code executes when a user makes a system call request and when interrupts and CPU exceptions occur.

11.2 Redefining the PATH Variable

The point of this section is that the PATH variable can be set in a script to search only those directories that it really needs. The order it searches these directories can also be optimized.

One factor that affects command execution is the amount of time it takes for the shell to locate a command. Recall from Section 3.2.2, “Finding commands,” on page 3-15, that the PATH variable is commonly used to find them.

- PATH is used to locate non-built-in commands and commands that do not contain a slash (/) character.
- PATH represents a list of directories that is sequentially searched to find a command. For example:

```
PATH=name1:name2:name3:name4:name5
```

It will take more time to find a command located in *name5* than for one located in *name1*.

- Because PATH is commonly exported, a shell script will inherit its parent’s value—a value which may not meet the needs of the script.
- A script can redefine the PATH variable to a value more appropriate for the set of commands it will execute.
 - It can omit command directories it is not using.
 - It can order the command directories so that the most frequently accessed ones are specified earlier in the search path.
 - If portability is a concern, be careful when redefining the PATH variable. Other UNIX systems may not locate commands in the same directories.
- Recall that when a child process modifies a shell variable, the modification is not reflected in the parent’s environment.

AV 11-3

Absolute vs. Relative Pathnames

```
$ time grep BUF /usr/include/sys/[m-z]*.h > $HOME/data
```

```
real      3.0
user      0.7
sys       0.5
```

```
$ cd /usr/include/sys
```

```
$ time grep BUF [m-z]*.h > $HOME/data
```

```
real      1.2
user      0.7
sys       0.2
```

sp1103

© The Santa Cruz Operation, Inc.

11.3 Absolute vs. Relative Pathnames

The overall point of this section is that neither type of pathname is inherently more efficient than the other. Generally, the more efficient name will be the shorter pathname because it searches through fewer directories.

AV 11-3: Absolute vs. Relative Pathnames

This slide shows examples of two commands that basically do the same thing. One uses full pathnames; the other uses relative names.

Try executing these commands with the shell's **-x** option in effect so that you can observe how the ***** wildcard expands.

Many commands on the system access files when they perform their work. You indicate which files to use by specifying a file pathname.

A pathname can take on one of two forms:

An absolute (or full) path	This pathname begins with the slash (/) character and the path starts at the root of the directory structure (or as specified by the chroot command).
A relative path	This pathname begins with a non-slash (/) character and the path starts at the current directory structure (as specified by the cd command).

Neither type of pathname is inherently more efficient than the other. Generally, the more efficient name will be the *shorter* pathname.

For example, if your current directory is */usr/peter/reports/august*:

- It is quicker to access the *rpt13* file with the relative path *rpt13* than with the absolute path */usr/peter/reports/august/rpt13*.
- It is quicker to access the *july* directory with the relative path *../july* than with the absolute path */usr/peter/reports/july*.
- However, it is quicker to access the *etc* directory with the absolute path */etc* than with the relative path *../../etc*.

Absolute paths to a given file are fixed in length, but depending on what the current directory is, a relative path could be shorter than an absolute one. With the **cd** command, you can take advantage of relative pathnames by changing your current working to a directory that is nearer to the files you will be accessing.

Recall that a child process modifies its environment, such as changing the current working directory, this has no impact on the parent's environment.

AV 11-4

Shell Built-in Commands		
:	for	test
.	getopts	times
break	hash	trap
case	if	type
continue	newgrp	ulimit
cd	pwd	unset
echo	read	umask
eval	readonly	until
exec	return	wait
exit	set	while
export	shift	

sp1104

© The Santa Cruz Operation, Inc.

Instructor Guide

11.4 Built-in Shell Commands

AV 11-4: Shell Built-in Commands

This slide lists the commands that are built-in to the Bourne shell.

Recall that there are three types of commands on the system:

- Shell built-in commands
- Binary commands
- Shell scripts

Of these, the built-in commands generally run the fastest because a process does not need to be created as with binaries and scripts; the shell directly executes them.

You are, of course, limited with what can be done with the built-in commands. Sometimes, however, you have the choice of performing a task with a built-in or non-built-in command. In these cases, you will usually find the built-in command to be more efficient.

This exercise shows a good example of how to perform a task with a built-in or with a non-built-in command. The built-in command performs the task more efficiently.

Instructor Guide

Exercise: Built-in command efficiency

If you want to interactively assign a value to a shell variable, you can use either of these command lines to accomplish this task. Which is more efficient and why?

```
VARIABLE=`line`  
read VARIABLE
```

GuruNotes Before the **test** command was built into the shell, another good example of performing a task with a built-in or with a non-built-in command is:

```
if [ "$REPLY" = yes ]
then
    command-list-1
else
    command-list-2
fi

case "$REPLY" in
    yes ) command-list-1 ;;
    *   ) command-list-2 ;;
esac
```

When **test** (**[**) was not a built-in, a process had to be created to perform the test (a string comparison). The **case** command, however, has always been built-in, and it inherently performs string comparisons.

AV 11-5

Pipeline Efficiency

```
$ wc /usr/dict/words
23739  23739 194186 /usr/dict/words

$ cat pipe1
sort /usr/dict/words | grep sco > data

$ time pipe1
real    11.2
user    0.3
sys     0.2

$ cat pipe2
grep sco /usr/dict/words | sort > data

$ time pipe2
real    0.8
user    0.0
sys     0.0
```

sp1105

© The Santa Cruz Operation, Inc.

11.5 Pipeline Efficiency

Shell pipes are a powerful feature that allow you to combine commands together; the output of one command becomes the input for another.

11.5.1 Data reduction

AV 11-5: Pipeline Efficiency

This slide shows two pipelines that accomplish the same task, but one is more efficient than the other.

The file `/usr/dict/words` used to be (and perhaps still is) used by the **hashmake** (see **spell**) command. On UnixWare 7 systems this file is found in `/usr/ucblib/dict`. It contains a list of words for the spelling dictionary (over 23 thousand words!).

The two sample pipelines are in a script so that **time** can time the whole command line and not just the first command in each pipeline.

Frequently, a *filter* is used in a pipeline. Recall that a filter is a command that is able to read from the standard input and write to the standard output.

Some filters reduce the amount of output they produce relative to the input they were given; others increase the amount. These commands may be called *shrinkers* and *expanders*, respectively.

For example, **grep** displays only those lines from a file that match a given pattern, which is usually a small fraction of the total file. It could be considered a shrinker.

Conversely, **nl** adds line numbers to a file, which serves to increase the amount of output it generates relative to its input. It could be considered an expander.

When given a choice, it is better to place shrinkers earlier in a pipeline and expanders later, to reduce the amount of data that needs to be transferred through the pipeline.

This exercise provides for a discussion of shrinkers and expanders.

Exercise: Shrinkers and expanders

What are some other filter commands that could be classified as *shrinkers*? As *expanders*? Depending on command-line options, which could act as a shrinker or expander?

11.5.2 Unnecessary pipelines

Because pipelines generate multiple processes in order to affect the I/O of your commands, they can be used inefficiently and should be avoided when they provide no additional functionality beyond what a single command can do.

A common waste of a pipeline is to begin with the **cat** command. Unless **cat** is concatenating two or more files—and not just displaying one file—there is no good reason to have it there! With just one file, its net effect at the beginning of a pipeline is to cause the standard input for the next command in the pipeline to refer to this file. The shell can more easily do this redirection with the **<** symbol.

This exercise demonstrates several examples where a pipeline is not needed to accomplish a task.

Exercise: Unnecessary pipelines

Compare the following pairs of command lines and determine whether they are functionally equivalent. If so, which is more efficient?

- `cat message | mail -s Subject joan`

`mail -s Subject joan < message`

- `cat report memo | lp`

`lp report memo`

- `echo 'Time to go!' | write beth`

`write beth << EOF`

`Time to go!`

`EOF`

- `ls $HOME | sort -r`

`ls -r $HOME`

- `ps -fe | grep '^[ ]*paul`

`ps -fu paul`

11.6 Lesson Summary

Although they are relatively quick to develop, shell scripts do not usually execute as quickly as a binary or built-in command. This lesson described some strategies you can use to improve the efficiency of your scripts, including judiciously changing your `PATH` variable and current directory; using built-in commands wherever possible; and efficiently using pipelines to send data.

Review questions

1. What do the `real`, `user`, and `sys` times indicate in the output of the **`time`** command?
2. Why do shell scripts often define the `PATH` variable even though they inherit a value from the parent?
3. Which is more efficient: absolute or relative pathnames? Explain.
4. Why are built-in commands usually more efficient than other types of commands?
5. Why is it more efficient to place a *shrinker* command at the beginning of a pipeline than an *expander*?

11.7 Relevant Commands

time	Displays the elapsed time for a command to execute
-------------	--

11.8 Answers to Review Questions

1. The “real,” “user,” and “sys” times in the output of the time command indicate: The time elapsed during the command (real) The time spent executing the command in user mode (user) The time spent executing the command in system mode (sys)
2. A script often redefines the PATH variable to a value more appropriate for the set of commands it will execute. This allows it to omit command directories it is not using, and to order the command directories so that the most frequently accessed ones are specified earlier in the search path.
3. Neither absolute nor relative pathnames are inherently more efficient than the other. The path that will be more efficient is the one that searches through the fewer number of directories.
4. Built-in commands are usually more efficient than other types of commands because a new process does not have to be created to execute the command.
5. It is more efficient to place a “shrinker” command at the beginning of a pipeline than an “expander” because less data will be sent through the pipeline. As a result, the commands later in pipeline will have less data to process.

A

***Answers to Lab
Exercises***

Instructor Guide

Instructor Guide

Appendix A

Answers to Lab Exercises

Introduction

This appendix provides answers to the lab exercises that were presented in the lessons of this course. For maximum benefit, do not refer to this section as you are attempting these exercises. Rather, you should work through them on your own, asking for assistance from your instructor where needed.

The solutions presented here may be useful to you upon review of the materials outside the classroom environment. Also bear in mind, however, that because of the richness of utilities within UNIX, there may be many possible “correct” answers to the problems. Your solution may not exactly match the model solutions presented here.

A.1 Lesson 1: Course Introduction

This lesson has no lab exercises.

A.2 Lesson 2: Shell Overview

This lesson has no lab exercises.

A.3 Lesson 3: Command Execution

This lesson has no lab exercises.

A.4 Lesson 4: Basic Scripts

This section provides suggested answers to the lab exercises in Lesson 4.

Lab: Your login environment

```

:
#    @(#) profile 23.3 91/10/24
#
#    Copyright (C) 1989-1991 The Santa Cruz Operation, Inc.
#    All Rights Reserved.
#    The information in this file is provided for the exclusive use of
#    the licensees of The Santa Cruz Operation, Inc.  Such users have the
#    right to use, modify, and incorporate this code into other products
#    for purposes authorized by the license agreement provided they include
#    this notice and the associated copyright notice with any such product.
#    The information in this file is provided "AS IS" without warranty.
#
# .profile    -- Commands executed by a login Bourne shell
#

PATH=$PATH:$HOME/bin:.      # set command search path
MAIL=/usr/spool/mail/`logname` # mailbox location
export PATH MAIL

# use default system file creation mask

eval `tset -m ansi:ansi -m :`?${TERM:-ansi} -r -s -Q`

#####
# The above lines came from the standard .profile which is given to  #
# new users.  The following lines satisfy the instructions of the lab #
# exercise.                                                           #
#####

# Set screen to green border, blue foreground, yellow background
setcolor -o green blue yellow

# Update personal log with current date and time
echo 'Last logged in: `c`' >> login.log
date >> login.log

# Automatically invoke a mailer to read any new mail that may have arrived
mail

# Enable messages to be sent to your screen via the write command
echo Enabling messages
mesg y

```

Lab: System status

```
#!/bin/sh
# Syntax:  system_status
# This script displays status information on the system.  The
# generated report contains status information for the following:
#
#      * Who is logged onto the system
#      * Which filesystems are mounted
#      * Which processes are running on the system
#      * Printer queue status
#
# Note: See the more man pages for an explanation of the -w and -d options.

# Who is logged onto the system
echo 'The following users are logged in:'
who | more -wd

# Which filesystems are mounted
echo '\n\nThe following filesystems are mounted:'
df -v | more -wd

# Which processes are running on the system
echo '\n\nThe following processes are on the system:'
ps -e | more -wd

# Printer queue status
echo '\n\nPrinter queue information:'
lpstat -t | more -wd
```


A.5 Lesson 5: Shell Variables

This section provides suggested answers to the lab exercises in Lesson 5.

Lab: Phone database

Add phone book entries

```
#!/bin/sh
# Syntax: add
#
# This program adds a new entry to a telephone
# database which prompts the user for the following information:
#
#      * Name
#      * Department
#      * Phone extension
#      * Login name
#
# This information is added as a single line with <Tab>-separated
# fields to the text file $HOME/phone.data. For example:
#
#      Andrew Tonkin      Tech Support      8649      andyt
#
# Once the data has been added to the database file, the file is sorted.

DATAFILE=$HOME/phone.data

# Prompt user for information about the new entry
echo 'Full name of user ==> \c'
read NAME
echo 'Department ==> \c'
read DEPT
echo 'Phone Extension ==> \c'
read PHONE
echo 'Login name of user ==> \c'
read LOGINNAME

# Add name to database file
echo $NAME'\t'$DEPT'\t'$PHONE'\t'$LOGINNAME >> $DATAFILE
echo $NAME added to $DATAFILE

# sort the database file (by user's [first] name)
# see sort for description of -o option
sort $DATAFILE -o $DATAFILE
```

Instructor Guide

Advanced add

```
#!/bin/sh
# Syntax: add.guru NAME DEPT EXT LOGIN
```

```
#
# This version of the add program adds a new entry to a telephone
# database from the following information supplied as positional
# parameters:
#
#      * Name ($1)
#      * Department ($2)
#      * Phone extension ($3)
#      * Login name ($4)
#
# This information is added as a single line with <Tab>-separated
# fields to the text file $HOME/phone.data. For example:
#
#      Andrew Tonkin      Tech Support      8649      andyt
#
# Once the data has been added to the database file, the file is sorted.

DATAFILE=$HOME/phone.data

# Verify user information for the new entry
echo Full name of user: $1
echo Department: $2
echo Phone Extension: $3
echo Login name of user: $4

# Give user an opportunity to abort if information is incorrect
echo '\nIs this correct? Press <Return> to continue, <Delete> to abort'
read DUMMY

# Add name to database file
echo $1'\t'$2'\t'$3'\t'$4 >> $DATAFILE
echo $1 added to $DATAFILE

# sort the database file (by user's [first] name)
# see sort for description of -o option
sort $DATAFILE -o $DATAFILE
```

Search phone book entries

```
#!/bin/sh
# Syntax: ext
#
# This program searches the telephone database and displays the
# line(s) that match a pattern provided by the user.
#
# The user is prompted for a pattern (which can be the name,
# department, extension, or login name) and then performs the search.
DATAFILE=$HOME/phone.data

# Prompt user for information about the new entry
echo 'Enter pattern to search for (e.g., Name, Phone, Dept, Login)\n==> \c'
read PATTERN

# search the database file
# see grep for description of -i option
```

```
grep -i $PATTERN $DATAFILE
```

Advanced search

```
#!/bin/sh
# Syntax: ext.guru
#
# This version of the ext program searches the telephone database and
# displays the line(s) that match a pattern provided by the user as
# a positional parameter.
#
DATAFILE=$HOME/phone.data

# search the database file
# see grep for description of -i option
echo searching for the pattern $1...
grep -i $1 $DATAFILE
```

Lab: A friendly find

```
#!/bin/sh
# Syntax: search file at directory
#
# This ia a user-friendly find command, which allows users to enter:
#
#     search lostfile at start
#
# instead of the more cumbersome command line:
#
#     find start -name lostfile -print
#
echo Searching for the file $1, beginning at directory $3
find $3 -name $1 -print
```

Advanced options

```
#!/bin/sh
# Syntax: search.guru
#
# This is a user-friendly find command, which allows users to enter:
#
#     search.guru
#
# instead of the more cumbersome command line:
#
#     find start -name lostfile -print
#
# The version of the script prompts the user for needed information rather
```

```
# than looking for it on the command line.

echo 'Enter filename to search for ==> \c '
read FILE
echo 'Enter directory to start search ==> \c'
read DIR

echo Searching for the file $FILE, beginning at directory $DIR
find $DIR -name $FILE -print
```

A.6 Lesson 6: Quoting

This section provides suggested answers to the lab exercises in Lesson 6.

Lab: Command-line prompt

```
# Modify your command line prompt so that upon login, it displays
# your machine name and login name in the following format:
#
#      <machine|loginname>
#
# Use command substitution to determine the names rather than
# hard-code particular names. This will allow you to use the same
# command line for different users on different machines. There
# should be a <Space> character following the prompt.
#
# The following line, added to your .profile, will change your prompt
# as indicated above

PS1="<`uname`|`logname`> "
```

Lab: Phone database: revisited

```
#!/bin/sh
# Syntax: add
#
# This version of the program adds a new entry to a telephone
# database by prompting the user for the information which is then
# stored in positional parameters:
#      $1: Name
#      $2: Department
#      $3: Phone extension
#      $4: Login name

DATAFILE=$HOME/phone.data

# Prompt user for information about the new entry
echo 'Full name of user ==> \c'
set "`line`" # Set value of $1
echo 'Department ==> \c'
set "$1" "`line`" # Set value of $2, retaining $1
echo 'Phone Extension ==> \c'
set "$1" "$2" "`line`" # Set value of $3, retaining $1 and $2
echo 'Login name of user ==> \c'
set "$1" "$2" "$3" "`line`" # Set value of $4, retaining $1, $2, and $3

# Verify user information for the new entry
echo Full name of user: $1
echo Department: $2
echo Phone Extension: $3
echo Login name of user: $4
```

```
# Give user an opportunity to abort if information is incorrect
echo '\nIs this correct? Press <Return> to continue, <Delete> to abort'
read DUMMY

# Add name to database file
echo "$1\t$2\t$3\t$4" >> $DATAFILE
echo $1 added to $DATAFILE

# sort the database file (by user's [first] name)
# see sort for description of -o option
sort $DATAFILE -o $DATAFILE
```

Lab: Shell escape

```
#!/bin/sh
# This shell program prompts the user for a command line and then
# executes it. It is able to handle user input such as:
#
#      ps -e | more
#      who > logfile
#      echo $TERM

echo "Command? ==> \c" >&2
eval `line`
```

Lab: Recursive grep

```
#!/bin/sh
# Syntax: grepall pattern start-directory
#
# This script searches for a pattern in all files in a given
# directory and also in all subdirectories from that point on down
# the directory tree structure. The output includes the matching
# line(s) in the file as well as the filename itself.
echo "Searching for files that contain the pattern $1, starting at $2"
find "$2" -exec grep "$1" /dev/null '{}' \;

# Okay, that last statement was rather terse. What did it do? The
# find command recursively searches for files that match a given
# criterion. In this example, no criteria were provided, so all
# files starting at $2 were selected by find.
#
# The -exec option causes find to repeatedly execute a command,
# once for each file that it finds (in this example, the grep
# command). Here, grep is searching for the pattern specified by
# $1 in two files: /dev/null and '{}'. The '{}' notation is
# substituted by find to the current filename that it has found.
#
# /dev/null (the "bit bucket" special file) is used to force grep
# to display the filename. When grep is invoked, it displays the
# filename only if it is provided with more than one name. Here,
# /dev/null is simply used as a "dummy" filename where, of course,
# grep will be unable to find the pattern.
#
# The \; notation tells find that there are no more "sub-
# arguments" to the -exec argument. It is quoted so that it is not
# interpreted by the shell; find needs to see it on the command
# line.
```

A.7 Lesson 7: Decision Constructs

This section provides suggested answers to the lab exercises in Lesson 7.

A friendly find: revisited

```
#!/bin/sh
# Syntax: search [ filename [directory] ]
#
# This version of the search program can accept 0, 1, or 2 command
# line arguments.
#
# 0 arguments: Begin the search at the current directory. Search
# for a filename as specified in the environmental variable
# SEARCHFILE. If the variable is NULL or not set, prompt the user
# for a filename.
#
# 1 argument: Begin the search at the current directory. Search for
# a filename as specified in the argument.
#
# 2 arguments: these specify both the start directory and the filename.
# Determine the number of positional parameters
case $# in
0 )   # No arguments given.

        # check for $SEARCHFILE environmental variable and set if
        # it is null or not set already
        if [ -z "$SEARCHFILE" ]
        then
            echo "Enter filename to search for ==> \c"
            read SEARCHFILE
        fi

        # Begin search in current directory
        STARTDIR=.
        ;;
1 )   # One argument given.

        # $1 refers to file.
        SEARCHFILE="$1"

        # Begin search in current directory
        STARTDIR=.
        ;;
2 )   # Two arguments given.
        # $1 refers to file.
        SEARCHFILE="$1"

        # $2 refers to starting directory.
        STARTDIR="$2"
        ;;
* )   # Too many arguments. Provide error message.
```



```

        echo "Syntax: $0 [ filename [directory] ]" >&2
        exit 1
    ;;
esac

echo "Searching for the file $SEARCHFILE, beginning at directory $STARTDIR"
find "$STARTDIR" -name "$SEARCHFILE" -print
exit 0

```

Lab: Phone database: revisited

```

#!/bin/sh
# Syntax: phone
# This script modifies the Phone Database programs from a previous
# lab to use a menu interface.
#
# Specifications
#
# The following specifications are implemented in the program:
#
# A user can choose a given menu item by choosing either the
# corresponding number or by entering the first letter of the menu
# choice text. For example, a user can search the database by
# entering 1, S, or s.
#
# Menu choices 1 and 2 incorporates the functionality already
# written in the previous labs.
#
# Menu choice 3 displays the entire database with the more
# command.
#
# Menu choice 4 verifies that the user really wants to purge the
# file (i.e., truncated, but not removed).

DATAFILE=$HOME/phone.data

echo 'TELEPHONE DATABASE MENU
-----
1. Add name to database
2. Search database
3. Display entire database
4. Purge the database
5. Quit

Enter choice ==> \c'

read CHOICE DUMMY

case "$CHOICE" in
1 | a | A ) # Add name to database
    # Prompt user for information about the new entry
    echo 'Full name of user ==> \c'
    read NAME
    echo 'Department ==> \c'
    read DEPT
    echo 'Phone Extension ==> \c'

```

```
    read PHONE
    echo 'Login name of user ==> \c'
    read LOGINNAME

    # Add name to database file
    echo "$NAME\t$DEPT\t$PHONE\t$LOGINNAME" >> $DATAFILE
    echo $NAME added to $DATAFILE

    # sort the database file (by user's [first] name)
    # see sort for description of -o option
    sort "$DATAFILE" -o "$DATAFILE"
    ;;

2 | s | S ) # Search database
    echo 'Enter pattern to search for ==> \c'
    read PATTERN

    # search the database file
    # see grep for description of -i option
    grep -i "$PATTERN" "$DATAFILE"
    ;;

3 | d | D ) # Display the entire database
    more "$DATAFILE"
    ;;

4 | p | P ) # Purge the database
    echo "Are you sure you want to purge the database? [y/n] \c"
    read ANSWER DUMMY
    if [ "$ANSWER" = y -o "$ANSWER" = Y ]
    then
        echo "Purging database..."
        : > "$DATAFILE" # Truncate file via I/O redirection
    else
        echo "Database not purged."
    fi
    ;;

5 | q | Q ) # Quit
    echo "Have a nice day. Goodbye."
    exit 0
    ;;

* )
    echo "Unknown response. Aborting program."
    exit 2
    ;;

esac

exit 0
```

A.8 Lesson 8: Loop Commands

This section provides suggested answers to the lab exercises in Lesson 8.

Lab: Phone database: revisited

```
#!/bin/sh
# Syntax: phone
# This script modifies the Phone Database programs from a previous
# lab to use a menu interface. This version repeatedly displays
# the menu (and executes the choice) until the user indicates to quit.
#
# Specifications
#
# The following specifications are implemented in the program:
#
# A user can choose a given menu item by choosing either the
# corresponding number or by entering the first letter of the menu
# choice text. For example, a user can search the database by
# entering 1, S, or s.
#
# Menu choices 1 and 2 incorporates the functionality already
# written in the previous labs.
#
# Menu choice 3 displays the entire database with the more
# command.
#
# Menu choice 4 verifies that the user really wants to purge the
# file (i.e., truncated, but not removed).

DATAFILE=$HOME/phone.data

while :      # Create an infinite loop
do
    cat <<- EOF

    TELEPHONE DATABASE MENU
    -----
    1. Add name to database
    2. Search database
    3. Display entire database
    4. Purge the database
    5. Quit
EOF

    echo 'Enter choice ==> \c'

    read CHOICE DUMMY

    case "$CHOICE" in
    1 | a | A ) # Add name to database
        # Prompt user for information about the new entry
        echo 'Full name of user ==> \c'
        read NAME
```

```

        echo 'Department ==> \c'
        read DEPT
        echo 'Phone Extension ==> \c'
        read PHONE
        echo 'Login name of user ==> \c'
        read LOGINNAME

        # Add name to database file
        echo "$NAME\t$DEPT\t$PHONE\t$LOGINNAME" >> $DATAFILE
        echo $NAME added to $DATAFILE

        # sort the database file (by user's [first] name)
        # see sort for description of -o option
        sort "$DATAFILE" -o "$DATAFILE"
        ;;

2 | s | S ) # Search database
        echo 'Enter pattern to search for ==> \c'
        read PATTERN

        # search the database file
        # see grep for description of -i option
        grep -i "$PATTERN" "$DATAFILE"
        ;;

3 | d | D ) # Display the entire database
        more "$DATAFILE"
        ;;

4 | p | P ) # Purge the database
        echo "Are you sure you want to purge the database? [y/n] \c"
        read ANSWER DUMMY
        if [ "$ANSWER" = y -o "$ANSWER" = Y ]
        then
            echo "Purging database..."
            : > "$DATAFILE" # Truncate file via I/O redirection
        else
            echo "Database not purged."
        fi
        ;;

5 | q | Q ) # Quit
        echo "\nHave a nice day.  Goodbye."
        exit 0
        ;;

* )
        echo "\n\007Unknown response.  Try again."
        ;;

esac

done # end of while loop

```

Lab: File tally

```

#!/bin/sh
# Syntax: tally file...
#

```



```
then
    echo "Directories\t$NUM_D\t%c"
    COUNT=0
    while [ $COUNT -lt $NUM_D ]
    do
        echo '*\c'
        COUNT=`expr $COUNT + 1`
    done
    echo # Carriage return
fi

if [ $NUM_L -ne 0 ]
then
    echo "Symbolic Links\t$NUM_L\t%c"
    COUNT=0
    while [ $COUNT -lt $NUM_L ]
    do
        echo '*\c'
        COUNT=`expr $COUNT + 1`
    done
    echo # Carriage return
fi

if [ $NUM_B -ne 0 ]
then
    echo "Block Device\t$NUM_B\t%c"
    COUNT=0
    while [ $COUNT -lt $NUM_B ]
    do
        echo '*\c'
        COUNT=`expr $COUNT + 1`
    done
    echo # Carriage return
fi

if [ $NUM_C -ne 0 ]
then
    echo "Char. Device\t$NUM_C\t%c"
    COUNT=0
    while [ $COUNT -lt $NUM_C ]
    do
        echo '*\c'
        COUNT=`expr $COUNT + 1`
    done
    echo # Carriage return
fi

if [ $NUM_O -ne 0 ]
then
    echo "Other Special\t$NUM_O\t%c"
    COUNT=0
    while [ $COUNT -lt $NUM_O ]
    do
        echo '*\c'
        COUNT=`expr $COUNT + 1`
    done
    echo # Carriage return
fi
echo "-----\nTotal Files\t$#"
```

A.9 Lesson 9: Shell Functions

This section provides suggested answers to the lab exercises in Lesson 9.

Lab: Phone database: revisited

```
#!/bin/sh
# Syntax: phone
# This script modifies the Phone Database programs from a previous
# lab to use a menu interface. This version repeatedly displays
# the menu (and executes the choice) until the user indicates to quit.
#
# Specifications
#
# The following specifications are implemented in the program:
#
# A user can choose a given menu item by choosing either the
# corresponding number or by entering the first letter of the menu
# choice text. For example, a user can search the database by
# entering 1, S, or s.
#
# Menu choices 1 and 2 incorporates the functionality already
# written in the previous labs.
#
# Menu choice 3 displays the entire database with the more
# command.
#
# Menu choice 4 verifies that the user really wants to purge the
# file (i.e., truncated, but not removed).

getyn()
{
    while echo "${*}? [y/n] == \c" >&2
    do
        read REPLY DUMMY
        case "$REPLY" in
            [yY]* )return 0 ;;
            [nN]* )return 1 ;;
            * )echo "\07Please enter <y> or <n>..." ;;
        esac
    done
}
DATAFILE=$HOME/phone.data

while :      # Create an infinite loop
do
    cat <<- EOF

    TELEPHONE DATABASE MENU
    -----
    1. Add name to database
    2. Search database
    3. Display entire database
    4. Purge the database
```

```
5. Quit
EOF

echo 'Enter choice ==> \c'

read CHOICE DUMMY

case "$CHOICE" in
1 | a | A ) # Add name to database
    # Prompt user for information about the new entry
    echo 'Full name of user ==> \c'
    read NAME
    echo 'Department ==> \c'
    read DEPT
    echo 'Phone Extension ==> \c'
    read PHONE
    echo 'Login name of user ==> \c'
    read LOGINNAME

    # Add name to database file
    echo "$NAME\t$DEPT\t$PHONE\t$LOGINNAME" >> $DATAFILE
    echo $NAME added to $DATAFILE

    # sort the database file (by user's [first] name)
    # see sort for description of -o option
    sort "$DATAFILE" -o "$DATAFILE"
    ;;

2 | s | S ) # Search database
    echo 'Enter pattern to search for ==> \c'
    read PATTERN

    # search the database file
    # see grep for description of -i option
    grep -i "$PATTERN" "$DATAFILE"
    ;;

3 | d | D ) # Display the entire database
    more "$DATAFILE"
    ;;

4 | p | P ) # Purge the database
    if getyn "Are you sure you want to purge the database"
    then
        echo "Purging database..."
        : > "$DATAFILE" # Truncate file via I/O redirection
    else
        echo "Database not purged."
    fi
    ;;

5 | q | Q ) # Quit
    echo "\nHave a nice day. Goodbye."
    exit 0
    ;;

* )
    echo "\n\007Unknown response. Try again."
    ;;

esac
done # end of while loop
```


Lab: File tally: revisited

```
#!/bin/sh
# Syntax: tally file...
#
# This program examines the files specified as command line arguments
# and displays a total for each common UNIX file type.
# This version of the program also displays a histogram in addition to
# the count. The histogram is implemented with a shell function

# Verify command line syntax
if [ $# -lt 1 ]
then
    echo "Syntax: $0 file..." >&2
    exit 1
fi

# Initialize counter variables
NUM_F=0      # Number of ordinary files
NUM_D=0      # Number of directory files
NUM_L=0      # Number of symbolic link files
NUM_B=0      # Number of block special device files
NUM_C=0      # Number of character special device files
NUM_O=0      # Number of other special files

# Define HISTOGRAM function
HISTOGRAM ()
{
    # $1 = character string which identifies file type
    # $2 = number of files of a given type

    if [ $2 -ne 0 ]
    then
        echo "$1\t$2\t\c"
        COUNT=0
        while [ $COUNT -lt $2 ]
        do
            echo '*\c'
            COUNT=`expr $COUNT + 1`
        done
        echo # Carriage return
    fi
}

# Process each file
for FILE
do
    # Note: test for symlink must occur before other tests because other
    # tests will follow the link to the file and determine *its* type
    if [ -L "$FILE" ]          # Test for symbolic link
    then
        NUM_L=`expr $NUM_L + 1`
    elif [ -f "$FILE" ]        # Test for ordinary file
    then
        NUM_F=`expr $NUM_F + 1`
    elif [ -d "$FILE" ]        # Test for directory file
    then
```

```
        NUM_D=`expr $NUM_D + 1`
    elif [ -b "$FILE" ]           # Test for block device special file
    then
        NUM_B=`expr $NUM_B + 1`
    elif [ -c "$FILE" ]           # Test for char. device special file
    then
        NUM_C=`expr $NUM_C + 1`
    else                           # Default for other file type
        NUM_O=`expr $NUM_O + 1`
    fi
done

# Print totals (if non-zero count)
HISTOGRAM "Ordinary Files" $NUM_F
HISTOGRAM "Directories" $NUM_D
HISTOGRAM "Symbolic Links" $NUM_L
HISTOGRAM "Block Device" $NUM_B
HISTOGRAM "Char. Device" $NUM_C
HISTOGRAM "Other Special" $NUM_O
echo "-----\nTotal Files\t$#"
```

A.10 Lesson 10: Shell Signal Handling

This section provides suggested answers to the lab exercises in Lesson 10.

Lab: Logoff cleanup

```
# Place the following line in your .profile to cause the screen
# to clear automatically upon logout

trap clear 0
```

Lab: Phone database

```
#!/bin/sh
# Syntax: phone
# This script modifies the Phone Database programs from a previous
# lab to use a menu interface. This version repeatedly displays
# the menu (and executes the choice) until the user indicates to quit.
#
# Specifications
#
# The following specifications are implemented in the program:
#
# The following signals are ignored by this script:
#     SIGINT (2)
#     SIGQUIT (3)
#     SIGTERM (15)
#
# A user can choose a given menu item by choosing either the
# corresponding number or by entering the first letter of the menu
# choice text. For example, a user can search the database by
# entering 1, S, or s.
#
# Menu choices 1 and 2 incorporates the functionality already
# written in the previous labs.
#
# Menu choice 3 displays the entire database with the more
# command.
#
# Menu choice 4 verifies that the user really wants to purge the
# file (i.e., truncated, but not removed).

# Set traps
# Ignore the following signals in this script:
#     SIGINT (2)
#     SIGQUIT (3)
#     SIGTERM (15)
trap '' 2 3 15

getyn()
{
    while echo "${*}? [y/n] == \c" >&2
```

```

do
    read REPLY DUMMY
    case "$REPLY" in
        [yY]* )return 0 ;;
        [nN]* )return 1 ;;
        * )echo "\07Please enter <y> or <n>..." ;;
    esac
done
}
DATAFILE=$HOME/phone.data

while :      # Create an infinite loop
do
    cat <<- EOF

    TELEPHONE DATABASE MENU
    -----
    1. Add name to database
    2. Search database
    3. Display entire database
    4. Purge the database
    5. Quit
EOF

    echo 'Enter choice ==> \c'

    read CHOICE DUMMY

    case "$CHOICE" in
        1 | a | A ) # Add name to database
            # Prompt user for information about the new entry
            echo 'Full name of user ==> \c'
            read NAME
            echo 'Department ==> \c'
            read DEPT
            echo 'Phone Extension ==> \c'
            read PHONE
            echo 'Login name of user ==> \c'
            read LOGINNAME

            # Add name to database file
            echo "$NAME\t$DEPT\t$PHONE\t$LOGINNAME" >> $DATAFILE
            echo $NAME added to $DATAFILE

            # sort the database file (by user's [first] name)
            # see sort for description of -o option
            sort "$DATAFILE" -o "$DATAFILE"
            ;;

        2 | s | S ) # Search database
            echo 'Enter pattern to search for ==> \c'
            read PATTERN

            # search the database file
            # see grep for description of -i option
            grep -i "$PATTERN" "$DATAFILE"
            ;;

        3 | d | D ) # Display the entire database

```

```
        more "$DATAFILE"
        ;;
4 | p | P ) # Purge the database
    if getyn "Are you sure you want to purge the database"
    then
        echo "Purging database..."
        : > "$DATAFILE" # Truncate file via I/O redirection
    else
        echo "Database not purged."
    fi
    ;;
5 | q | Q ) # Quit
    echo "\nHave a nice day.  Goodbye."
    exit 0
    ;;
* )
    echo "\n\007Unknown response.  Try again."
    ;;
esac

done # end of while loop
```

A.11 Lesson 11: Efficiency Considerations

This section has no lab exercises.

B

Programming Tools

Instructor Guide

Instructor Guide

Appendix B

Programming Tools

Suggested lesson duration

This lesson is optional; it should take approximately 1 hour to complete. There is no lab exercise.

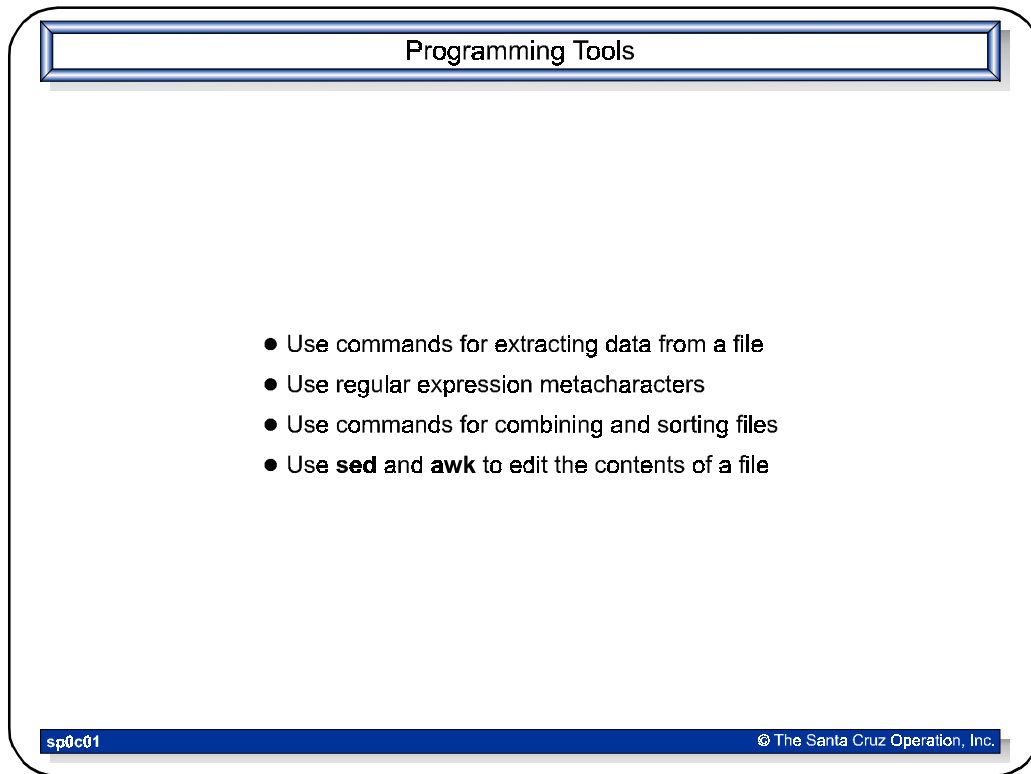
Pre-lesson classroom preparation

None.

Instructor preparation

None.

AV B-1



Programming Tools

- Use commands for extracting data from a file
- Use regular expression metacharacters
- Use commands for combining and sorting files
- Use **sed** and **awk** to edit the contents of a file

sp0c01 © The Santa Cruz Operation, Inc.

Instructor Guide

Introduction

You can write a shell script to perform a wide variety of routine tasks. Often, your scripts will need to use commands that manipulate the contents of files, either by extracting data or adding new data to them.

This appendix describes several commands that perform these functions, including **head**, **tail**, **cut**, **uniq**, **grep**, **cat**, **paste**, **sort**, and **sed**. Some of these commands perform pattern matching with *regular expressions*; this mechanism is discussed as well.

Objectives

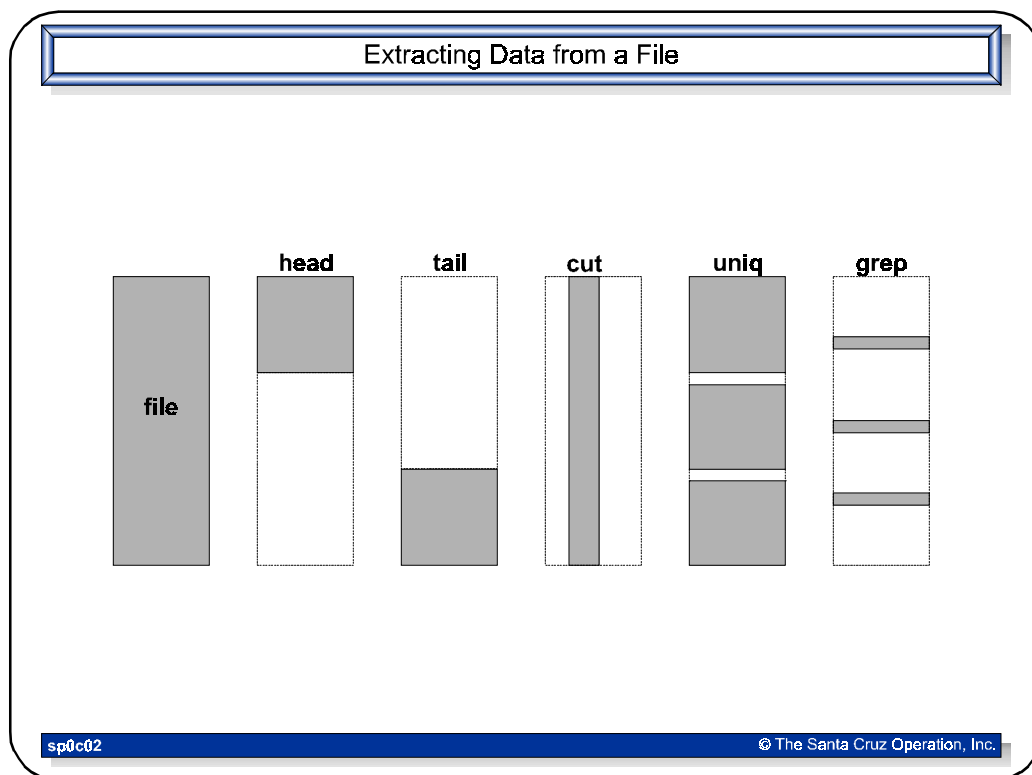
This appendix describes commands that extract and add data to files. By the end of this appendix, you will be able to:

- Use commands for extracting data from a file
- Use regular expression metacharacters with the **grep** and **sed** commands
- Use commands for combining and sorting files together
- Use **sed** and **awk** to edit the contents of a file

AV B-1: Programming Tools

This first slide describes the objectives covered in this appendix.

AV B-2



B.1 Extracting Data from Files

Several commands are available that allow you to break up a file into smaller pieces to deal with only that data you need. These commands include **head**, **tail**, **cut**, **uniq**, and **grep**.

AV B-2: Extracting Data from a File

This slide is a graphical representation of how the **head**, **tail**, **cut**, **uniq**, and **grep** commands operate on a file.

B.1.1 The head Command

The **head** command displays the first *count* lines of a file. It can be used as a filter—it writes to the *stdout* and if no *file* is specified, it reads from the *stdin*.

B.1.1.1 Syntax

The syntax is:

```
head -count [ file ... ]
```

where *count* specifies the number of lines to output (default=10).

B.1.1.2 Example

To display the first three lines from the */etc/passwd* file:

```
$ head -3 /etc/passwd
root:x:0:1:Superuser:/:/bin/ksh
daemon:x:1:1:System daemons:/etc:
bin:x:2:2:Owner of system commands:/bin:
```

B.1.2 The tail Command

The **tail** command displays the end of a file, beginning *+count* lines from the top, or *-count* lines from the end. It can be used as a filter—it writes to the *stdout* and if no *file* is specified, it reads from the *stdin*.

B.1.2.1 Syntax

The syntax is:

```
tail [ -lbc ] ±[count] [ file ... ]
```

where:

count Specifies the number of lines to output starting at the top (+) or bottom (-) of the file (by default, *count*=10)

lbc Specifies *count* units in lines (**l**), bytes (**b**), or characters (default=**l**)

B.1.2.2 Examples

To display the last line from the */etc/conf/cf.d/mdevice* file:

```
$ tail -1 /etc/conf/cf.d/mdevice
vw    I    icS    vw    0    106    8    128    -1
```

To display lines 3-4 from the */etc/passwd* file:

```
$ head -4 /etc/passwd | tail -2
bin:x:2:2:Owner of system commands:/bin:
sys:x:3:3:Owner of system files:/usr/sys:
```

The tail command can also be used with the **-f** option to continually print lines of a file as it is growing. This is useful for monitoring the growth of a file that is being written by some other process. For example:

```
$ tail -f /usr/adm/syslog
```

```
May  6 09:29:06 envis sendmail[3506]: JAA03504: to=root, ctladdr=root
(0/3), delay=00:00:00, xdelay=00:00:00, mailer=local, stat=Sent
May  6 09:31:55 envis /usr/dt/bin/rpc.ttdbserverd[3564]: svc_run()
May  6 09:31:56 envis /usr/dt/bin/rpc.ttdbserverd[3567]: svc_run()
May  6 09:32:36 envis /usr/dt/bin/rpc.ttdbserverd[3577]: svc_run()
May  6 09:32:36 envis /usr/dt/bin/rpc.ttdbserverd[3573]: svc_run()
May  6 09:32:36 envis /usr/dt/bin/rpc.ttdbserverd[3579]: svc_run()
May  6 09:32:36 envis /usr/dt/bin/rpc.ttdbserverd[3580]: svc_run()
May  6 09:44:0  envis syslogd: restart
May  7 13:03:0  envis syslogd: restart
May  7 13:03:11 envis unix: Novell IPX/TPI Socket Router 4.10a
May  7 13:03:12 envis unix: Novell IPX Lan Router 4.10a
May  7 13:03:11 envis in.snmpd[394]: t_open of /dev/ipx failed: Resource
temporarily unavailable.
May  7 13:03:13 envis rpcbind: Could not open connection on nbclts: system
error
```

B.1.3 The cut Command

The **cut** command extracts fields of data from each line of the standard input or from each line of the specified files. It is a useful tool for cutting columns of data from a file in table format, such as the file */etc/passwd*.

The **cut** command can be used as a filter—it writes to the *stdout* and, if no *file* is specified, it reads from the *stdin*.

B.1.3.1 Syntax

The syntax is:

```
cut [ options ] field-list [ file ... ]
```

where:

options Common options include:

- **-c**—Specifies character positions
- **-f**—Specifies fields

- **-dc**—Used with the **-f** option, the **-d** option specifies that the following character *c* is the field delimiter. The default is tab.

field-list A comma-separated list of character or field numbers, in increasing order. A dash (-) indicates inclusive ranges. Examples:

<code>-f 2,5-10</code>	(fields 2 and 5-10)
<code>-c -4,10</code>	(characters 1-4 and 10)
<code>-c 2-</code>	(characters 2 through the end-of-line)

B.1.3.2 Examples

Display the first and fifth fields from */etc/passwd*, where each field is colon-separated:

```
cut -d: -f1,5 /etc/passwd
```

Display only the names of everyone currently logged in:

```
who | cut -f 1
```

Strip the first two characters from each line in *myfile* and save the results:

```
cut -c 3- myfile > tmp$$; mv tmp$$ myfile
```

B.1.4 The **uniq** command

The **uniq** command reads an input file, compares adjacent lines, and checks for uniqueness. By default it will delete repeated identical lines. It can be used as a filter—with output going to the *stdout* and reading input from the *stdin* by specifying a dash (-) for the input filename.

This command only finds repeated *adjacent* lines. The **sort** command may be useful for preparing a file prior to its use.

B.1.4.1 Syntax

The syntax is:

```
uniq [ options ] [ inputfile [ outputfile ] ]
```

Some of the *options* that **uniq** accepts include:

- u** Causes only those lines that are *not* repeated to be output
- d** Causes one copy of those lines that *are* repeated to be output
- c** Causes each line to be preceded by a count of the number of times it occurred

Other options allow **uniq** to skip portions of the line during the comparison. Refer to the manual pages for details.

By default, *inputfile* refers to the *stdin*; use dash (-) to explicitly indicate this. By default, *outputfile* refers to the *stdout*.

B.1.4.2 Examples

To show the contents of an inventory file named *inventory*:

```
$ cat inventory
CD-ROM
CD-ROM
17" monitor
tape
17" monitor
15" monitor
CD-ROM
```

To show which items make up the inventory:

```
$ sort inventory | uniq
15" monitor
17" monitor
CD-ROM
tape
```

To show the number of items in the inventory:

```
$ sort inventory | uniq -c
  1 15" monitor
  2 17" monitor
  3 CD-ROM
  1 tape
```

B.1.5 The `grep` command

The **grep** command searches input files for lines that match a pattern and normally writes these lines to the *stdout*. If more than one file is searched, the name of the file in which each match occurs is also written to the *stdout* along with the matching line.

This command can be used as a filter—output goes to the *stdout*; input comes from a named file, but defaults to the *stdin* when no input file is specified.

B.1.5.1 Regular expressions

The patterns **grep** uses are either fixed strings or basic *regular expressions*, which are described in the **regex** manual pages. These regular expressions are used by other commands, including **vi** and **sed**.

The simplest form of regular expression is a string of characters. More complicated regular expressions use metacharacters in the string; such a regular expression will match a variety of strings.

A regular expression matches a zero or greater length string where the characters in the string correspond to the pattern. The search begins at the start of a string and ends when the first matching sequence is found. If the expression allows there to be a variable number of characters in the matched string, the longest leftmost matching string is found.

AV B-3

Using Metacharacters with `grep`

```
$ grep foo* letter
your foot injury
football games are today
april fools day

$ grep foo? letter
your foot injury

$ grep ^foo letter
football games are today

$ grep '[Hh]enr(y|ietta)' letter
Hello Henry
and as I was saying to Henrietta
```

sp0c03

© The Santa Cruz Operation, Inc.

Note that regular expression metacharacters are *not* the same as the filename expansion metacharacters used by the shell, although some do have similar meaning. It may be necessary to quote a regular expression metacharacter on a command line to prevent interpretation by the shell. These are some of the metacharacters that can be used to form a regular expression:

* (asterisk)	Matches zero or more occurrences of the previous single character (or subexpressions). For example, the expression <code>XA*Y</code> matches <code>AAAA</code> in the string <code>XAAAAAY</code> , and it matches the null string in <code>XY</code> , however, it does not match the string <code>XABCY</code> or <code>XAAAA</code> .
. (dot)	Matches any single character except a null character. It is analogous in function to the <code>?</code> shell metacharacter. The combined expression <code>. *</code> matches anything and is analogous in function to the <code>*</code> shell metacharacter.
[<i>list</i>] (brackets)	Matches any single character from the specified <i>list</i>. It is analogous in function to the <code>[...]</code> shell metacharacter. • For example, the expression <code>A[BCD]E</code> matches the strings <code>ABE</code>, <code>ACE</code>, or <code>ADE</code>, however, it does not match the string <code>ABCDE</code>. • The expression <code>A[BCD]*E</code> matches the strings <code>ABE</code>, <code>ACE</code>, or <code>ADE</code>—as well as <code>ABDCDDCCBCBE</code> and <code>AE</code>—however, it does not match the string <code>ABCDHE</code> or <code>ABCD</code>. • The <i>list</i> may specify an ascending range(s) of characters. For example, <code>[0-9]</code> or <code>[A-Za-z]</code>. • If the first character in the <i>list</i> is a caret (^), this matches any single character <i>not</i> in the specified <i>list</i>.
\$ (dollar)	Anchors the preceding expression to the end of the line
^ (caret)	Anchors the following expression to the beginning of the line. If both caret and dollar are used, the regular expression must match a complete line.

AV B-3: Using Metacharacters with grep

This slide shows examples using grep with regular expression metacharacters.

B.1.5.2 grep syntax

The grep syntax is:

```
grep [ options ] pattern [ file ... ]
```

where some of the *options* recognized by **grep** include:

- c** A count of matching lines is the only output.
- h** The name of the file containing the matching line is omitted from the output. It is used when searching multiple files.
- i** Pattern matching is case insensitive.
- l** Only the names of files with matching lines are displayed.
- n** Each line is preceded by its relative line number in the file.
- v** All lines *but* those which match are displayed.
- x** Displays only exact matches of an entire line.

B.1.5.3 Examples

Display the lines in the files */etc/conf/cf.d/mtune* and */etc/conf/cf.d/stune* that determine the size of the kernel open file table:

```
$ grep NFILE /etc/conf/cf.d/?tune
/etc/conf/cf.d/mtune:NFILE      0      100      64000
/etc/conf/cf.d/stune:NFILE           200
```

Identify when the user **steve** logged in:

```
$ who | grep '^steve '
steve      ttyp20      Oct 10 10:53
```

Display the user **anne**, but not the users **annette** or **maryanne** from the file */etc/passwd*:

```
$ grep '^anne:' /etc/passwd
anne:x:201:50:Anne Sample:/usr/anne:/bin/sh
```

Display lines that have *at least* four characters and end with a vowel:

```
$ grep '....[aeiou]$_' colors
```

```
aqua
```

```
blue
```

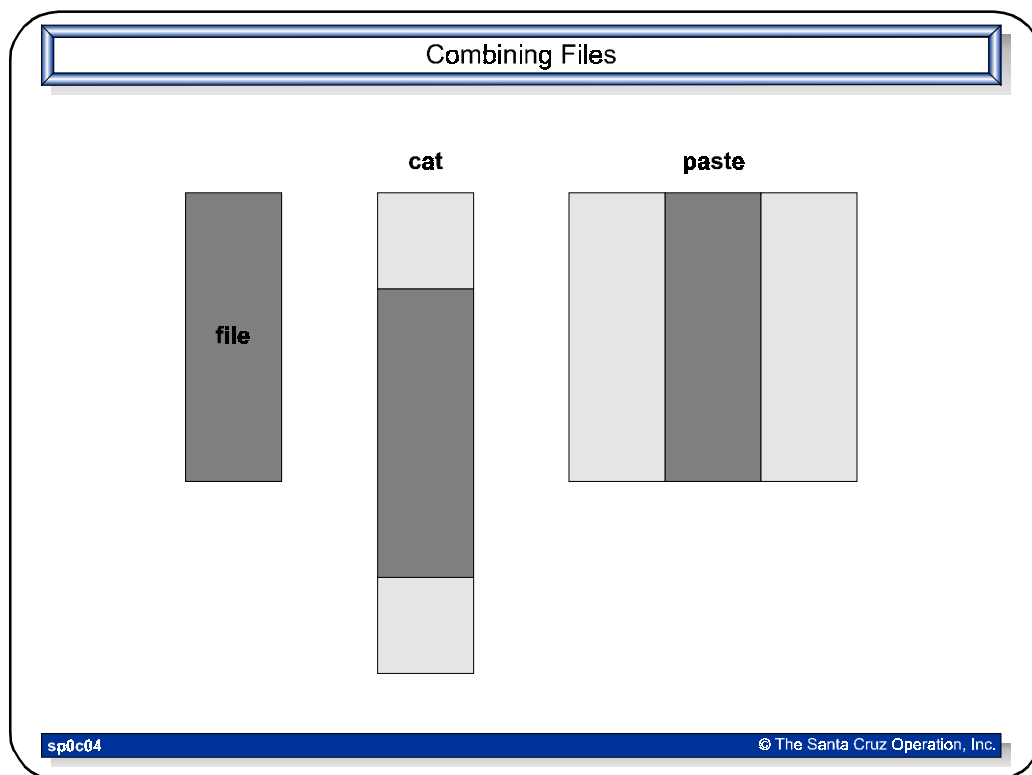
```
magenta
```

Display lines that have *exactly* five characters and *do not* end with a vowel:

```
$ grep '^....[^aeiou]$_' colors
```

```
green
```

AV B-4



Instructor Guide

B.2 Combining Files

Some commands allow you to combine files rather than break them apart. The **cat** command combines them vertically, whereas **paste** does so horizontally.

AV B-4: Combining Files

This slide is a graphical representation of how the **cat** and **paste** commands operate on files.

B.2.1 The cat command

The **cat** command reads each file in sequence and writes it on the *stdout*. If no input file is given, or if a single dash (-) is given, **cat** reads from the *stdin*.

B.2.1.1 Syntax

The syntax is:

```
cat [options] [ file ... ]
```

where some of the options include:

- s Suppresses warnings about nonexistent files
- u Causes the output to be unbuffered
- v Causes non-printing characters (with the exception of tabs, newlines, and form feeds) to be displayed

B.2.1.2 Examples

Concatenate *file1* and *file2*, saving the results in *file3*:

```
$ cat file1 file2 > file3
```

Read the *stdin* and save in *myfile*; view the results:

```
$ cat > myfile
No need for an editor
to create such a short file!
CTRL d
$ head myfile
No need for an editor
to create such a short file!
```

B.2.2 The paste command

The **paste** command concatenates corresponding lines of input files, treating each file as a column of a table and joining them together horizontally (parallel merging). It is the counterpart of **cat** which concatenates files vertically, that is, one file after the other.

Output goes to the *stdout*. If dash (-) is used in place of an input filename, **paste** reads a line from the *stdin*.

B.2.2.1 Syntax

The syntax is:

```
paste [ options ] file ...
```

where *options* are:

- d list** Replace the default tab as the line concatenation character by one or more alternate characters specified in *list*. The *list* is used circularly—when exhausted, it is re-used.
- s** Merge subsequent lines rather than one from each input file

B.2.2.2 Examples

These examples use the specified files for data:

```
$ cat color
red
green
blue
yellow
$ cat money
penny
nickel
dime
quarter
$ cat size
small
medium
large
```

Paste files using the default concatenation character (tab):

```
$ paste color size money
red      small  penny
green    medium nickel
blue     large  dime
yellow                   quarter
```

Paste files using a different concatenation character:

```
$ paste -d: color size money
red:small:penny
green:medium:nickel
blue:large:dime
yellow::quarter
```

Paste subsequent lines of the files using a different concatenation character:

```
$ paste -s -d+ color size money
red+green+blue+yellow
small+medium+large
penny+nickel+dime+quarter
```

List the files in */etc/default*, five names per line, using different concatenation characters:

```
$ ls /etc/default | paste -d',;+' - - - - -  
Stp,accounts;archive+authsh,backup  
boot,cc;cleantmp+codeset,cron  
device.tab,dumpdir;filesys+filesys-,format  
goodpw,idleout;inet+issue,it  
lang,lock;login+lpd,mail  
man,mapchan;mapkey+merge,micnet  
mnt,msdos;nbconf+netroff,passwd  
purge,pwr;restor+restore,scsihas  
slot,su;tape+tar,tcp  
usemouse,whois;+,
```

B.3 Sorting File Contents

The **sort** command sorts lines of one or more named files and writes the result to the standard output. If more than one file is specified, the sorted output of each file is combined.

By default, **sort** orders each input line lexicographically (like a dictionary), but this can be changed using command-line options.

B.3.1 Syntax

The basic **sort** syntax is:

```
sort [ options ] [ +field ] [ file ... ]
```

where:

- | | |
|------------------|---|
| -o file | Specifies the output file to use instead of the standard output |
| -M | Sorts months (e.g., “JAN” < “FEB” < “MAR”...) |
| -r | Reverses the order |
| -d | Uses “dictionary” order; only alphanumeric and space characters are significant |
| -b | Ignore leading blanks |
| -f | Treat upper and lower case letters as equals |
| -t c | Use <i>c</i> as the field separator character |
| -k keydef | Defines keys for restricting sorting to specified fields. The basic syntax for keydef is: |

start-field,end-field[flag]

NOTES: The **sort** command can be used to write unique lines using the **-u** option, but the **unique** command is the recommended method for generating unique output. Using **sort** might leave unwanted characters at the end of the sorted output.

The **-d** option does not imply **-f**. That is, when specifying “dictionary” order, characters are sorted in the order: numeric, lower case, upper case. Upper and lower case letters are *not* treated as equal.

B.3.2 Examples

The following examples assume the input file *phone-list*:

\$ cat phone-list

Norm MacDonald	4019
Uve Denvers	4211
Mitch Grommer	4842
Darius Mills	4165
Pam Macilhenny	4300
Greg Clos	4410

This example uses the default settings and sorts the file by the first field:

\$ sort phone-list

Darius Mills	4165
Greg Clos	4410
Mitch Grommer	4842
Norm MacDonald	4019
Pam Macilhenny	4300
Uve Denvers	4211

To sort using the last name, skip the first field by specifying **+1**:

\$ sort +1 phone-list

Greg Clos	4410
Uve Denvers	4211
Mitch Grommer	4842
Norm MacDonald	4019
Pam Macilhenny	4300
Darius Mills	4165

To sort by extension number and save to the file, *ext*:

\$ sort -b +2 phone-list > ext

Norm MacDonald	4019
Darius Mills	4165
Uve Denvers	4211
Pam Macilhenny	4300
Greg Clos	4410
Mitch Grommer	4842

To print the file */etc/passwd* using the numeric user ID (the third colon-separated field):

```
$ sort -t : -k 3,3n /etc/passwd
```

```
daemon:x:1:12:0000-Admin(0000):/:  
bin:x:2:2:0000-Admin(0000):/usr/bin:  
sys:x:3:3:0000-Admin(0000):/:  
adm:x:4:4:0000-Admin(0000):/var/adm:  
uucp:x:5:5:0000-uucp(0000):/usr/lib/uucp:  
mail:x:6:6:Mail Processes:/etc/mail:  
lp:x:7:9:0000-LP(0000):/var/spool/lp:/usr/bin/sh
```

To sort the file */etc/passwd* first by the numeric group ID (the fourth colon-separated field), then by the user ID:

```
$ sort -t : -k 4,4n -k 3,3n /etc/passwd
```

```
chris:x:100:1::/home/chris:/bin/ksh  
pw:x:102:1::/home/pw:/OpenServer/bin/sh  
uwpossh:x:103:1::/home/uwpossh:/usr/bin/posix/sh  
uwossh:x:104:1::/home/uwossh:/OpenServer/bin/sh  
bin:x:2:2:0000-Admin(0000):/usr/bin:  
root:x:0:3:0000-Admin(0000):/sbin/sh  
sys:x:3:3:0000-Admin(0000):/:  
adm:x:4:4:0000-Admin(0000):/var/adm:
```

B.4 Editing Files with sed

The stream editor, **sed**, allows you to perform non-interactive editing tasks, making it particularly suited to running sequences of editing commands on files from within a shell script.

For example, **sed** can replace all occurrences of a given word in a file with a different word, or delete all lines containing a specified string from a file.

B.4.1 Syntax

The basic syntax for **sed** is:

```
sed [ -e ] edit-command [ -f script-file ] [ inputfile ... ]
```

where:

-e	Indicates that the next argument is an editing command (may be omitted if there are no other -e or -f options specified)
-f script-file	Indicates that the next argument is the name of the file containing editing commands (typed one to a line)

The **sed** command reads the input a line at a time, carries out whatever editing changes are specified, then writes the changed lines to the standard output.

This simple example uses **sed** to change all occurrences of “red” to “blue” in the file *colors*:

```
$ cat colors
```

```
This is red.
```

```
This is red, too.
```

```
$ sed -e "s/red/blue/g" colors
```

```
This is blue.
```

```
This is blue, too.
```

The **-n** option is used to suppress the default output. As in the above example, the substitution is made in *colors* but this time, nothing is written to the standard output:

```
$ sed -n -e "s/red/blue/g" colors
```

```
$
```


AV B-5

Using sed Scripts

```
$ cat greeting
/Hello/s/Mike/John/g
$ sed -f greeting < inputfile > outputfile

$ cat inputfile
Hello Mike;
I am pleased to be able to say
a big "Hello" to you, Mike.
Regards, CEO

$ cat outputfile
Hello John;
I am pleased to be able to say
a big "Hello" to you, John.
Regards, CEO
```

sp0c05

© The Santa Cruz Operation, Inc.

B.4.2 Using sed scripts

AV B-5: Using sed Scripts

This simple example illustrates how **sed** is used with script files.

A **sed** script consists of editing commands, one per line, of the form:

```
[ address ] function [ arguments ]
```

where:

address Specifies the line or lines upon which **sed** will act. An address is:

- A decimal number that counts input lines cumulatively across files
- A \$ that addresses the last line of input
- A context address—a regular expression enclosed within a pair of non-alpha-numeric characters, usually slash(/). See Section B.1.5.1, “Regular expressions,” on page B-12.

A command line with two addresses separated by a comma selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second.

If address is omitted, **sed** processes each input line.

function Specifies the editing command **sed** is to perform. It may require *arguments*.

arguments Specifies the text affected by the function

The example **sed** script in AV B-5, *greeting*, searches each line that contains the string “Hello” (the address) and uses the **sed** search function (**s**) to replace each occurrence of “Mike” with “John” (arguments to **s**) on each line.

AV B-6

Using **sed** Functions (Part 1)

Change the first occurrence of "Unix" to "UNIX" on lines 1-15 of *myfile*:

```
$ sed '1,15s/Unix/UNIX/' myfile
```

Change all occurrences of "Unix" to "UNIX" on all lines of *myfile*:

```
$ sed 's/Unix/UNIX/g' myfile
```

Delete (substitute with null) the last two characters on a line, but only on lines that begin with a digit character:

```
$ sed '/^[0-9]/s/..$//' myfile
```

Delete line 5 of *myfile*, saving the results:

```
$ sed 5d myfile > newfile
```

Delete lines 8 through the end-of-file (note, this is equivalent to the command line `head -7`; results go to the *stdout*):

```
$ sed '8,$d' myfile
```

Delete lines that do not begin with a capital letter:

```
$ sed '/^[^A-Z]/d' myfile
```

Print line 5 of *myfile*, saving the results:

```
$ sed -n 5p myfile > newfile
```

Print lines that do not end with a period:

```
$ sed -n '/\.$/p' myfile
```

sp0c06

© The Santa Cruz Operation, Inc.

AV B-7

Using **sed** Functions (Part 2)

Append a border line (a string of asterisks) and blank line after any line that has the string "Plan" or "plan". Also delete lines containing the word "mail".

```
$ cat sedfile
/[Pp]lan/a\
*****\

/mail/d
$ finger -l | sed -f sedfile
```

Insert two lines of text at the beginning of *myfile*:

```
sed '1i\
These two lines are inserted \
at the top of the file' myfile
```

Replace lines 10-20 of *myfile* with new lines:

```
sed '10,20c\
These two lines replace\
lines 10-20 of the file' myfile
```

Replace blank lines with a border line (a string of asterisks). This example shows output being sent to the printer:

```
finger -l | sed '/^$/c\
*****' | lp
```

sp0c07

© The Santa Cruz Operation, Inc.

B.4.3 Using sed Functions

All **sed** commands require a function, which is named by a single character. These are some of the functions that **sed** supports:

s	Substitute text as specified by the <i>address</i> and <i>argument</i>
d	Delete lines specified by <i>address</i>
p	Explicitly print lines as specified by the <i>address</i> ; normally uses the -n option
i	Insert entire lines of text as specified by the <i>address</i> . This function requires an <i>argument</i> .
a	Append entire lines of text after those lines specified by the <i>address</i> and <i>argument</i> . The syntax is the same as the i function.
c	Change (i.e., replace) entire lines of text as specified by the <i>address</i> and <i>argument</i> . The syntax is the same as the i function.

Refer to the manual pages for more information on these and other functions.

AV B-6: Using sed Functions (Part 1)

This slide shows examples using the **sed** functions **s**, **d** and **p**. Note the need for the backslashes for line continuation; the single quotes are needed so the shell doesn't interpret the backslashes (and newlines) itself).

AV B-7: Using sed Functions (Part 2)

This slide shows examples using the **sed** functions **i**, **a** and **c**.

B.5 awk Basics

This section is a brief introduction to **awk**. It covers only the most basic concepts and includes some simple examples to help students write one-line programs to perform a handful of tasks. It will also provide a starting point for understanding the **awk** code included in many system shell scripts.

If the instructor decides to present this material, it is recommended for use after Lesson 5 is completed (at the earliest).

The **awk** programming language is designed for processing and reporting on the contents of text files. Although you can use **awk** to write large, complex programs, simple one-line **awk** programs are useful for handling many common tasks.

Like **sed**, **awk** reads input one line, or *record*, at a time. A record is a sequence of characters ending with a newline character. Each record is then split into *fields*—each field is a string of non-blank, non-tab characters.

A set of **awk** functions act on the input data and the results are sent to the standard output.

While **sed** is most useful for change the contents of a file or stream of data, **awk** is primarily used when special formatting is needed that cannot easily be accomplished with **sed**.

B.5.1 Syntax

The syntax for using **awk** on the command line is:

```
awk 'pattern-action statement' [ input_files ]
```

where the basic structure of the *pattern-action statement* is:

```
pattern {action}
```

For example, this **awk** program includes both a pattern and an action to print the telephone extension of any employee named “Mitch” listed in the file *phone-list*:

```
$ awk '/Mitch/ {print}' phone-list
Mitch Grommer          4842
```

Note that “Mitch” is surrounded by the slash character. The literal string “Mitch” is the simplest form of regular expression which must be enclosed within slashes to distinguish it from an **awk** command.

Either the pattern or the action can be omitted. If there is no action with a pattern, the matching line is printed. This example is equivalent to the last:

```
$ awk '/Mitch/' phone-list
Mitch Grommer      4842
```

If there is no pattern with an action, the action is performed for every input line. For example, this **awk** program simply prints every line in the file *phone-list*:

```
$ awk {print} phone-list
Mitch Grommer      4842
Norm MacDonald     4019
.....
```

B.5.2 Field variables

Since most of the work done with **awk** involves processing records, **awk** provides *field variables* for identifying fields within a record:

`$1, $2, $3 $4, ...`

The entire record is stored in the variable `$0`.

For example, this one-line program prints (to the standard output) the first and third fields of every line in *phone-list* that contains the string “Pam”:

```
$ cat phone-list
```

```
Norm MacDonald     4019
Pam Laughlin       4423
Uve Denvers        4211
Mitch Grommer      4842
Darius Mills       4165
Pam Macilhenny     4300
Greg Clos          4410
```

```
$ awk '/Pam/{print $1, $3}' phone-list
Laughlin           4423
Macilhenny         4300
```

B.5.3 Built-in Variables

In addition to the field variables, `$1, $2, ...`, there are internal variables built-in to **awk**. The basic set of built-in variables includes:

NF The total number of fields read

NR The total number of records

FILENAME Name of current input file

FNR Record number in current file

An example using the NR variable is a program which numbers the lines in a text file:

```
awk '{print NR $0}' /etc/passwd
```

B.5.4 User-defined variables

As with other programming languages, you can define your own variables to store data.

For example, the variable `sum` is used to accumulate the value of the third field (`$3`):

```
{sum = sum + $3}
```

In general, **awk** initializes variables with the string value `""` (null) and the numeric value 0.

B.5.5 Operators

The **awk** programming language includes a number of comparison operators and regular expressions which can be used to specify patterns.

These include the six relational operators:

`==` Equal to

`!=` Not equal to

`<` Less than

`<=` Less than or equal to

`>` Less than

`>=` Less than or equal to

This **awk** program uses the `==` relational operator to compare the first field to a text string and print the line(s) which match:

```
$ awk '$1 == "Mitch"' phone-list
Mitch Grommer        4842
```


This program prints all lines that begin with the letters “N” through “Z”:

```
$ awk '$1 >= "N"' phone-list
Norm MacDonald      4019
Pam Laughlin        4423
Pam Macilhenny      4300
Uve Denvers         4211
```

Other operators include those used with regular expressions:

~	Matches
!~	Does not match

For example, this program checks the *phone-list* file to see if extension 4300 is in use:

```
$ awk '$3 ~ /4300/ {print $1 $2}' phone-list
PamMacilhenny
```

B.5.6 Regular expressions

Patterns can also be created using the regular expressions provided by **awk**. Common regular expressions include:

<i>char</i>	Any non-meta character
<i>\char</i>	Literal character
^	Beginning of a string
\$	End of a string
.	Any character but newline
[s]	Any character in set s
[^s]	Any character <i>not</i> in set s

For example, this program prints each line in *phone-list* whose second field (last name) starts with the letter “D”:

```
$ awk '$2 ~ /^D/' phone-list
Uve Denvers         4211
```

To print all lines that have a first field beginning with “D” or “G”:

```
$ awk '$1 ~ /^[DG]/' phone-list
Darius Mills      4165
Greg Clos         4410
```

B.5.7 Formatting awk output

For more carefully formatted output, **awk** provides the **printf** statement. When used in awk program, the basic syntax is:

```
{ printf "format" [ arg ] }
```

where *format* is comprised of:

Literal characters	Character strings copied to the output stream
Conversion Specifications	Fetches arguments and converts if specified:
	%s no conversion
	%d written as decimal number
	%c written as character string
Escape sequences	\b backspace
	\r carriage return
	\n newline
	\t tab

For example, **printf** is used to format the output of the *sales* file to print all Q1 information by territory, with sales numbers in decimal format:

```
$ cat sales
Americas Q1 44500 1200
Americas Q2 8660 2450
Aisa Q1 8650 6637
Aisa Q2 10730 1269
Australia Q2 29200 400
Australia Q1 38400 14
Europe Q1 50900 920
Europe Q2 3286 26

$ awk '$2 ~ /Q1/ {printf "%10s %9d.00 %9d.00\n", $1, $2, $3}' sales
Americas 44500.00 1200.00
Europe 50900.00 920.00
Aisa 8650.00 6637.00
Australia 38400.00 14.00
```

B.5.8 Simple awk programs

Programs combining **awk** commands can be created in separate files, or run directly on the command line.

Often two special patterns, BEGIN and END, are used to control initialization and wrap-up in an **awk** program.

For example, the following **awk** program, **q1-report**, uses BEGIN to put column headings on the output. The program's third **printf** statement, which is executed for each line in the file *sales* containing the string "Q1," formats the output into a table, neatly aligned under the column headings. The END action prints the totals.

```
$ cat q1-report
```

```
BEGIN
    FS = "\t"
    printf "%30s", "First Quarter Sales\n"
    printf "%10s %12s %12s\n",
        "Territory", "Product A", "Product B\n" }
/Q1/ { printf "%10s %9d.00 %9d.00\n", $1, $3, $4
        sum1 = sum1 + $3; sum2 = sum2 + $4 }
END { printf "\n%10s %9d.00 %9d.00\n", "TOTAL", sum1, sum2 }
```

```
$ awk -f q1-report sales
```

```
    First Quarter Sales
Territory    Product A    Product B

    Aisa      8650.00      2450.00
Americas     44500.00     1200.00
Europe       50900.00      920.00
    Aisa      8650.00     6637.00
Australia    38400.00      14.00

    TOTAL     151100.00    11221.00
```

B.6 Appendix Summary

SCO UNIX provides commands that allow you to make various modifications to your files. This appendix describes commands used to extract data from a file (**head**, **tail**, **cut**, **uniq**, **grep**), combine files (**cat**, **paste**), **sort** them, and edit them using the **sed** and **awk** commands.

B.7 Relevant Commands

awk	Processes text files and writes results to the <i>stdout</i>
cat	Reads one or more files and writes them to the <i>stdout</i>
cut	Cuts out columns from a table or fields from each line of one or more files and writes them to the <i>stdout</i>
grep	Searches one or more input files for lines that match a pattern and writes them to the <i>stdout</i>
head	Writes the first <i>count</i> lines of one or more files to the <i>stdout</i>
paste	Concatenates corresponding lines of input files, writing the result to the <i>stdout</i>
sed	Edits text files and writes results to the <i>stdout</i>
sort	Sorts and merges files
tail	Writes the end of one or more files to the <i>stdout</i>
uniq	Reads an input file, compares adjacent lines, deletes repeated lines, and writes the result to the <i>stdout</i>

