IBM

Korn and Bash Shell Programming

(Course code AL32)

Student Notebook

ERC 1.0

IBM Certified Course Material

Trademarks

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

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

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

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

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

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

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

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

October 2007 edition

Contents

Trademarks

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

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

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

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

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

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

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

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

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

Course description

Korn and Bash Shell Programming

Duration: 5 days

Purpose

This course will teach you how to use shell scripts and utilities for practical system administration of the IBM RISC System/6000.

Audience

Support staff of AIX for RISC System/6000.

Prerequisites

An understanding of programming fundamentals: variables, flow control concepts such as repetition and decision. A working knowledge of AIX including the use of the vi editor, find and grep commands. Students without this experience should attend *AIX Version 5 Basics Plus*.

Objectives

After completing this course, students should be able to:

- Distinguish Korn and bash shell specific features
- Use utilities such as sed and awk to manipulate data
- Understand system shell Scripts such as /etc/shutdown
- Write useful shell Scripts to aid system administration

Contents

- Basic shell concepts
- Flow control in a shell Script
- Functions and typeset
- Shell features such as arithmetic and string handling
- Using regular expressions
- Using sed, awk and other AIX utilities

Agenda

Course Times: 9:00 - 17:00 (16:00 on the last day)

Day 1

Course and Student Introductions
Unit 1 Basic Shell Concepts
Lab 1 Using Shell Basics
Lunch
Lab 1 (Cont)
Unit 2 Variables
Lab 2 Variables
Unit 3 Return Codes and Traps

Day 2

Lab 3 Testing
Unit 4 Flow Control
Lunch
Lab 4 Shell Programming Constructs
Unit 5 Shell Commands
Lab 5 Shell Commands and Features

Day 3

Lab 5 (Cont)
Unit 6 Arithmetic
Lab 6 Shell Arithmetic
Lunch
Unit 7 Shell Types, Commands, and Functions
Lab 7 Typeset and Functions

Day 4

Unit 8 More on Shell Variables
Lab 8 More on Shell Variables
Unit 9 Regular Expressions and Text Selection
Lunch
Unit 9 (Cont)
Lab 9 Regular Expressions and Data Selection
Unit 10 The sed Utility

Day 5

Lab 10 The sed Utility
Unit 11 The awk Program
Lab 11 Using awk
Lunch
Lab 11 (Cont)
Unit 12 Good Practices and Review
Close

Unit 1. Basic Shell Concepts

What this unit is about

This unit introduces the Korn and Bash shells and environments.

What you should be able to do

After completing this unit, you should be able to:

- Recognize file types
- Identify metacharacters
- Use various quoting mechanisms
- Redirect file input and output

How you will check your progress

Accountability:

- Checkpoint questions
- Hands-on exercises

Unit Objectives

After completing this unit, you should be able to:

- Describe the AIX shells
- Use the AIX filesystem
- Create a shell script
- Use metacharacters
- Use I/O redirection
- Use pipes and tees
- Group commands
- Run background processes
- Use shell job control
- Use command line recall and editing

Figure 1-1. Unit Objectives AL321.0

Notes:

Shells

- What is a shell?
  – User interface to AIX
  – Command interpreter
  – Programming language

- AIX shells:

– Bourne	- bsh	
– Bourne-Again	- bash	
– C	- csh	
– Distributed	- dsh	
– Korn (88)	- ksh	
– Korn (93)	- ksh93	
– POSIX	- psh	
– Restricted	- rsh	
– Trusted	- tsh	
– Default	- sh	(links to ksh in AIX V4 and V5)

Figure 1-2. Shells AL321.0

Notes:

Any of the AIX shells can be the initial login shell for a user. Each has different features and syntax. Shells have some built-in commands which we will cover in later units. The AIX operating system provides a number of useful commands that are available from all shells. Examples of these will appear in this and later units.

The Korn shell adds C shell features to the Bourne shell to produce the most user-friendly and powerful shell. It is also faster than the other shells. The Korn shell is more recent than the other shells, but retains backward compatibility with the Bourne shell. David G. Korn wrote the Korn shell at AT&T's Bell Labs (now Lucent) where it is now widely used.

Bourne shell is the oldest shell; it was written at AT&T's Bell Labs by Steven Bourne.

Another shell that is commonly found on open platforms and Linux in particular is the Free Software Foundation GNU Bourne Again SHell (bash). This is a Bourne shell compatible rewrite but with many extensions and additional features, similar to the Korn shell.

The C shell has a completely different syntax to Bourne shell. It provides some advanced features such as job-control and command-line editing. It was written by Bill Joy at the University of California at Berkeley. It's primary use is as an interactive shell and is not usually used in writing shell scripts.

The dsh, or Distributed shell, distributes commands among the nodes of a cluster. It uses a daemon to gather state info of the nodes. A small script then gets this information and then remotely executes the commands via rsh or ssh.

POSIX is Portable Operating System Interface — Xopen. The IEEE POSIX 1003.2 shell and Utilities Language Committee report is the Open Systems definition of a shell. The Korn shell conforms to this document. A POSIX shell is implemented under AIX Version 4 and 5 as a link to the Korn shell.

The Restricted shell provides a limited subset of the commands in Bourne shell:

- You can't change your working directory
- You may not run operating system commands unless they are in the working directory
- The command search path cannot be changed
- Redirection is not allowed

The Trusted shell is a subset of the Korn shell, but it is AIX-specific, and is one of the enhanced security features of AIX Version 3:

- Only “trusted” and shell built-in commands can be executed
- The internal field separator characters cannot be reset
- Functions may not be defined
- There is no command history
- The command search path is fixed in a special start-up profile file (/etc/tsh_profile)

The default login shell for each user (in */etc/passwd*) is the */bin/ksh* Korn shell. The Bourne shell is the default login shell for older UNIX systems, and early versions of AIX.

The default shell is */bin/sh*. For AIX Version 3 this was a link to the */bin/bsh* Bourne shell program. In AIX Version 4 and 5 it is a link to /bin/ksh the Korn shell.

This course will concentrate on the Korn and Bash shells, pointing out differences from the Bourne shell.

This Course

- AIX 5 loads with the 88 Korn Shell, the 93 Korn Shell, the Bourne Shell, and the Bash Shell (and more).
- root may create different users who log into different shells.
- The default shell in AIX 5 is 88 Korn Shell.
- This course focuses on the 88 Korn Shell. The slight differences in the other three shells will be noted on the slide or in the student notes.
- Available logins are:
  – team01, team02, team03, team04, team05
  – bash01, bash02, bash03, bash04, bash05
  – ksh9301, ksh9302, ksh9303, ksh9304, ksh9305
- The password is the same as your login name
- All exercises are located in /home/workshop:
  – They need to be copied into your $HOME

Figure 1-3. This Course AL321.0

Notes:

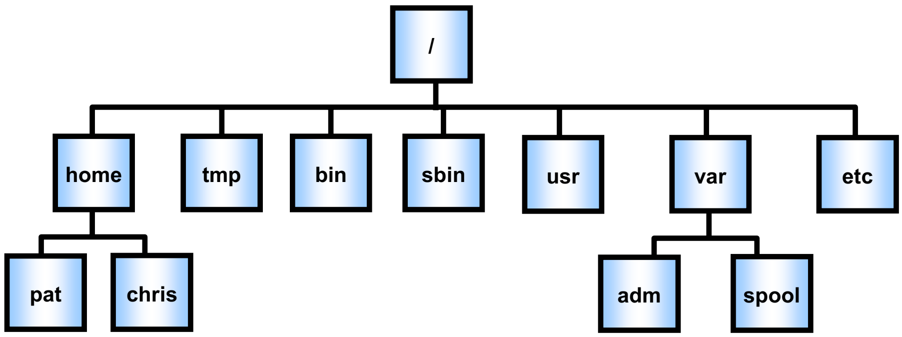
If using the Korn shell 93, bash, or Bourne shell, please check the student notes for slight differences. If any of the shells differ greatly from the Korn shell 88, it will be noted on the slide.

During the first exercise, you will be instructed to copy all the pre-typed exercises to your $HOME. You are able to copy all at once, or as needed -- “on demand” if I may be so bold.

***NOTICE*: The instructor may have special login/password information for your class. Please pay attention when this is discussed!**

Directories

- The filesystem comprises directories in a hierarchical structure



- Refer to the files and directories with a a full or relative path name
- "." represents current dir, ".." represents parent directory

Figure 1-4. Directories AL321.0

Notes:

Each user on the system has a home directory with their portion of the tree underneath: like */home/pat* for user pat. In AIX Version 3.2, */home* replaced */u*.

On the next page, you will see summary table of commands to manipulate the file-system.

Basic File Commands

Command	Argument	Function
`mkdir`	`directory`	Create new directory directory
`rmdir`	`directory`	Delete empty directory directory
`rm`	`file`	Remove a file
`rm -r`	`directory`	Delete directory directory and any sub-directories
`ls`	`directory`	Give a listing of directory - many options: l, R, d, a i, t
`pwd`		Print working directory - where you are in the tree right now
`mv`	`old new`	Rename a file or directory - "new" can be a new file name, or a directory in which to place the file
`cp`	`old new`	Copies a file to a new name
`ln`	`name copy`	Creates another name without copying the contents
`cd`	`directory`	Change working directory to directory

Figure 1-5. Basic File Commands AL321.0

Notes:

Notes:

The current directory is referred to by "." or the "." notation, and is used to specify a relative pathname to a directory or file from the current directory, for example, from ***/home***, ***./chris*** refers to Chris's home directory. Entering ***cd*** with no directory changes the working directory to *your* home directory.

To refer to the parent of the current directory (go up a layer) we use the "**..**" notation, for example, from ***/home/chris***, ***../pat*** is Pat's home directory.

The Korn shell provides ***cd*** and ***pwd*** as built-in commands. AIX provides ***pwd*** as an operating system command. Additional features are provided with the Korn shell ***cd***:

`cd -` changes to the last working directory

`cd old new` replaces the string **old** with **new** in the current directory pathname, and tries to change directory to the resultant path, for example, if /**home**/**pat** is the working directory, ***cd pat chris*** will change to /**home**/**chris**.

A File

- Definition:
  – Collection of data, located on a portion of a disk.
  – Stream of characters or a **byte stream**.

- No structure is imposed on an ordinary file by the operating system.

- Examples:
  – Binary executable code – `/bin/ksh`
  – Text data – `/etc/passwd`
  – C program text – `/home/pat/prog.c`
  – Device special file – `/dev/null`
  – Directory special file – `/home`

```
$ file filename          – to find out which file type
$ strings filename       – if the file type is 'binary'
```


Figure 1-6. A File AL321.0

Notes:

Directories and devices are known as **special files** — the operating system controls their use.

Some other operating systems impose a record structure on all files — AIX does not have this restriction. You can have whatever you like in an ordinary file.

One special file that we'll be using a lot is */dev/null* — this is a bottomless pit where output can be directed if you want to lose it.

The *file* command can be used to find out what type a particular file is, that is, binary executable, C program text, and so forth.

If the file is a binary file, do not use the *cat* command to view it. Use the *strings* command. This command will send at least 4 contiguous, ASCII, printable characters to STDOUT.

AIX File Names

- Should be descriptive of the content
- Are case-sensitive
- Should use only alphanumeric characters:

  UPPERCASE lowercase digits
  # . @ - _

- Should not begin with "+" or "-" sign
- Should not contain embedded blanks or tabs
- Should not contain shell "special" characters:

  * ? > < / ; & ! ~ $ \ |
  [ ] { } ( ) ` ` ' ' " "

Figure 1-7. AIX File Names AL321.0

Notes:

Remember *.filename* files (dot files) are hidden from the normal *ls* command unless you use the *-a* option, or you are root.

Unlike DOS, AIX does not impose limitations on file name structure — you can have a 20 character file name with a *.pat* on the end if that makes your happy.

There is a limit of 256 characters on the length of a shell command line, and 255 characters on file names. As complicated and lengthy commands are sometimes necessary, it is usually wise to avoid very long file names.

What Is a Shell Script?

- A readable text file which can be edited with a text editor
  - `/usr/bin/vi shell_prog`
- Anything that you can do from the shell prompt
- A program, containing:
  - System commands
  - Variable assignments
  - Flow control syntax
  - Shell commands
- And comments
  - `#!/bin/ksh` is not a comment if `#!` is in the first position

Figure 1-8. What Is a Shell Script? AL321.0

Notes:

The first line of a shell script can be read as an instruction to the shell to run the script in a new specified type of shell. This ensures that scripts are correctly run when you have switched your login to another shell type.

#!/usr/bin/ksh, or #!/usr/bin/ksh93 or #!/usr/bin/bash, and so forth, as the first line ensures that the script is always run in the proper shell.

Invoking Shells

`$ ksh`	begins a new 88 Korn shell, interrupting the current one
`$ ksh -c commands`	runs commands in a shell
`$ ksh -r`	starts a restricted shell

terminates the current shell and replaces with new shell

`$ exec ksh`

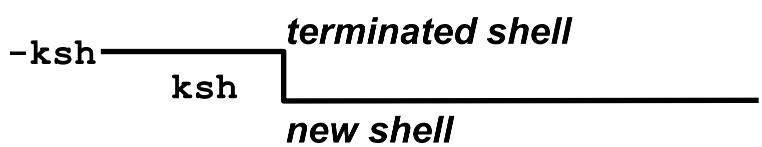


Figure 1-9. Invoking Shells AL321.0

Notes:

There are many options for invoking the Korn shell. These are described fully in Unit 5. The Bourne and Bash shells share the options shown above with the Korn shell.

With the *-c* option, multiword *commands* must be enclosed in quotes, so that they are treated logically as a single word.

A waiting shell is **sleeping** until its new shell signals that it has completed.

The ***exec*** command is a shell built-in command.

To open a new Korn shell version 93, type ***ksh93***.

To open a new Bash shell, type ***bash***.

To open a new Bourne shell, type ***bsh***.

Invoking Scripts

$. prog — (sourced) in current shell environment

$ ksh prog — run **prog** in a new Korn shell

$ prog (or ./prog) — run in a new shell if **prog** is executable

$ exec prog — run **prog** in a new shell to replace the current one

Figure 1-10. Invoking Scripts AL321.0

Notes:

The "." method (sourcing) causes the entire prog file to be read by the shell before it executes any of it. Other methods of invoking scripts execute each line of code as it is read in. Don't forget, if #! is used in position 1, the 'sourcing' gives way to the named shell.

The "." method is used when you want to change your current environment. For example, if the prog script changed any variables, the variables would be changed after the script completes when using ".". If the prog script changes directories, you will be in the new directory when the script completes. You need only "r:" permissions on the script.

The next two methods, *ksh prog* and *prog* run the script in a subshell. If the prog script changed any variables, those new values are reset to old values when the subshell closes. Likewise, if the prog script changed directories, that will not affect the parent shell. When the prog script is finished running, you will be back to your original environment. The prog script will not change any variables or directories when executed this way. You will need both "r" and "x" permissions on the script itself.

The last method, *exec prog*, replaces your current shell.

Korn Shell Configuration Files

Invoking the Korn Shell sources:

`/etc/environment`	Sourced by all AIX processes
`/etc/profile`	Sourced by login shells
`.profile` `.exrc (.vimrc)`	Login shells source these files in the user's home directory
`$ENV` `.kshrc (.bashrc)`	A resource file listed in the ENV environment variable will be sourced by the shell

time

Each new **explicit** Korn shell sources the ENV file again

* If using CDE, `.dtprofile` must be changed to force an execute of `.profile`. If using bash, please refer to student notes.

Figure 1-11. Korn Shell Configuration Files AL321.0

Notes:

If you use the Korn shell as your login shell, your *.profile* file should contain settings for ENV. For example, it is typical to include the following lines in the script:

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

This variable sets up the $HOME/.kshrc file to be executed for all Korn shells - login and subshells. This is the difference between .profile and .kshrc. .profile is only executed once -- when you login. Therefore, any environment you set up there will only be set up in your login shell (except for variables which can be exported). The *.kshrc* file will be executed for all Korn shells, login and sub, so this is where you put things you want permanently part of your Korn shell environment, but that can't be exported, for example, aliases and *set* commands.

For privileged shells, run with the "*-p*" option, the user's *.profile* and ENV files are replaced by */etc/suid_profile*. A privileged shell is automatically invoked if your effective user id (UID) is different from your real UID, or your effective group (GID) is different from your real GID.

The AIX Windows Common Desktop Environment (CDE) provides access to Korn shell windows. Normally these are not login shells. A .dtprofile file will be sourced if found in the home directory. To force it to execute your .profile as well, you must uncomment the DTSOURCEPROFILE=TRUE statement.

The Trusted shell uses */etc/tsh_profile* in place of */etc/profile* and the user's *.profile* file.

The C shell sources *.login* and *.cshrc* files in the user's home directory, instead of */etc/profile* and the users' *.profile* and *.kshrc* files.

Only Korn shells source the ENV file. You invoke an explicit shell when you use the Korn shell directly or explicitly. For example when you use commands like:

```
ksh, ksh prog, ksh -c commands
```

When you run a program (other than by the dot method) that has the special comment *#!/usr/bin/ksh* as its first line, you also invoke an **explicit** shell.

Another common file used is *.exrc*. This file contains commands used to control your vi editor environment. For example:

```
set showmode
set tabstop=4
ab IBM International Business Machines, Inc.
```

in your *.exrc* file. You do not need to use the colon before the command in the *vi* interactive form of the command.

The Bash shell looks for (in this order):

1. /etc/environment for all AIX processes
2. /etc/profile
3. $HOME/.bash_profile
   if not found, then
   $HOME/.bash_login
   if not found, then
   $HOME/.profile

The BASH_ENV variable is slightly different from the ENV variable. A file named $HOME/.bashrc will be executed for any interactive Bash shell, even if the BASH_ENV variable is not set. (This is NOT true with the Korn shell ENV variable). However, if you want $HOME/.bashrc to be executed for non-interactive shells (running shell scripts without the #! line), then you must set the BASH_ENV variable. Also, the .bashrc file is only executed for sub-bash shells. You must force an execute of it (source) for login shells.

Often, this is placed in .bash_profile (or .bash_login or .profile).

Refer to the previous notes on .kshrc to explain the difference of what to put in .bash_profile (.profile) and .bashrc (.kshrc). The Bash shell also provides $HOME/.bash_logout to be executed each time you log out.

What Are Metacharacters?

- Characters with special meaning
  - Three types
    - Wildcard (or expansion)
    - Shell
    - Quoting
  - Shell processes metacharacters before executing a command
  - There are many different shell metacharacters
  - Metacharacters can be mixed
- Wildcard metacharacters can be turned off by shell options

Figure 1-12. What Are Metacharacters? AL321.0

Notes:

Metacharacters do not represent themselves. The three types are a way of classifying the metacharacters. Wildcards are the most commonly used (like *, ?). Korn and Bash shell uses metacharacters, like ? and +. The third type are quotes like double, single and the \ escape character.

Unit 5 shows how wildcard metacharacters can be turned off using shell options.

Wildcard Metacharacters

Metacharacters that form patterns that are expanded into matching filenames from the current directory:

`*`	-	Match any number of any characters
`?`	-	Match any single character
`[abc]`	-	Match a single character from the bracketed list
`[!az]`	-	Match any single character except those listed
`[a-z]`	-	Inclusive range for a list

Character Equivalence Classes can be used in place of range lists, to avoid National Language collation problems:

`[[:upper:]]`	-	Range list of all ***upper case*** letters
`[[:lower:]]`	-	All lower case letters: ***a, b, c,... z***
`[[:digit:]]`	-	Digits: ***0, 1, 2,... 9***
`[[:space:]]`	-	Spacing characters: ***tab, space***, and so forth

Figure 1-13. Wildcard Metacharacters AL321.0

Notes:

Filenames beginning with a "." must be matched explicitly, with a "." as the first character in your pattern.

There are many more Character Equivalence Classes: **[:alpha:], [:alnum:], [:cntrl:], [:graph:], [:print:], [:punct:], [:xdigit:]** and **[:blank:]**. Further description of these is in the AIX Commands Reference manual, under *ksh*, *bsh*, *csh*, and especially *ed*.

Commands and utilities such as *grep*, *sed* and *awk* also use pattern matching metacharacters and Character Equivalence Classes. **These have similar functions but are not identical (see units 9, 10 and 11):**

*****	to match any number of the preceding character (so it must always follow something),
.	the dot matches any single character, rather than the **?**,
[^ab]	with a **^** in place of a **!** to signify an exclusion list,
^	can be used to signify the beginning of a line,
$	will signify the end of a line.

Sample Directory

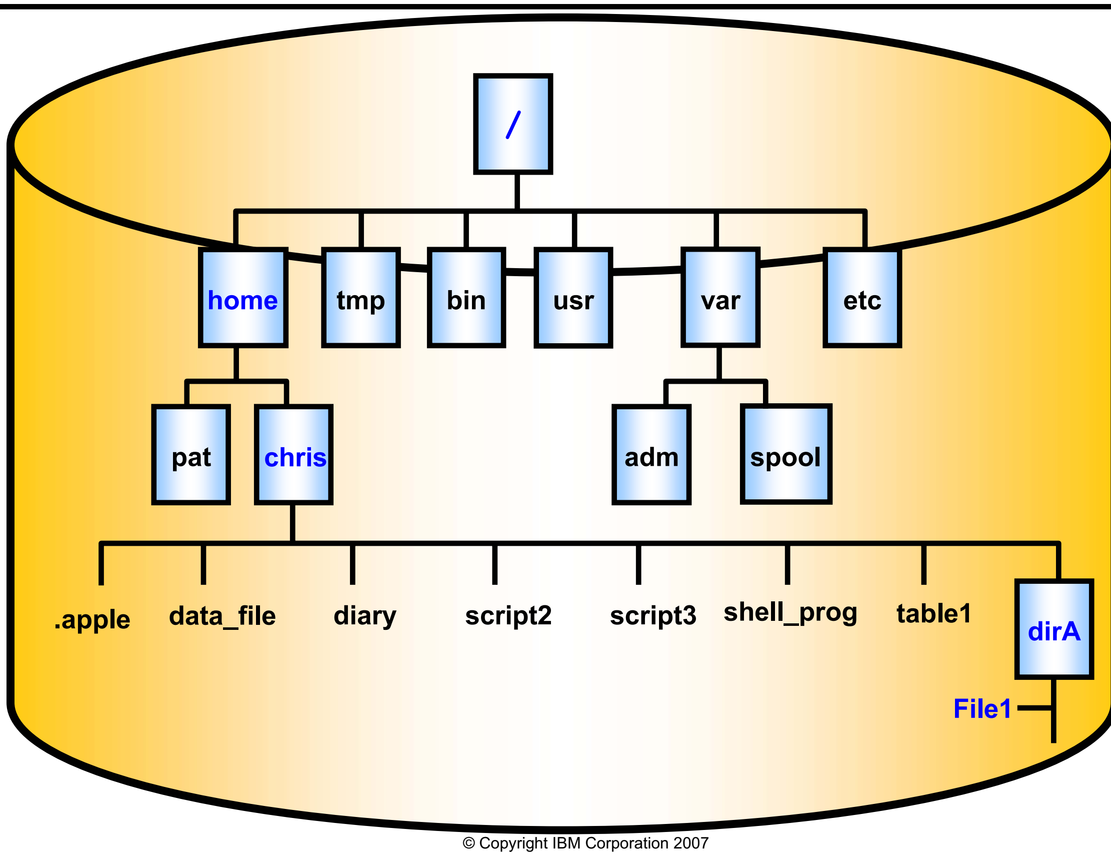


Figure 1-14. Sample Directory AL321.0

Notes:

These files will be used for the examples of metacharacter file name expansion on following pages.

Expansion Examples

`$ rm d*y`	removes the `diary` file
`$ file script*`	identifies `script2` and `script3`
`$ head script[345]`	displays the top lines of `script3`
`$ more script[3-6]`	displays `script3` screen by screen
`$ tail script[!12]`	displays the last lines of `script3`

Now, it's your turn...

```
$ touch ?a*

$ pg [st][ah]*

$ ls d*

$ lpr [a-z]*t[0-9]
```


Figure 1-15. Expansion Examples AL321.0

Notes:

Assume the current directory is */home/chris*.

Remember, the wildcard expands **before** the command runs.

More Shell Metacharacters

The Korn shell can match multiple patterns

`*(pattern|pattern...)`	zero or more occurrences
`?(pattern|pattern...)`	zero or one occurrence
`+(pattern|pattern...)`	one or more occurrences
`@(pattern|pattern...)`	exactly one occurrence
`!(pattern&pattern...)`	anything except

One or more patterns, separated with "`|`" for "or", "`&`" for "and"

Examples:

`*([0-9])`	0 or more ***consecutive*** digits
`?(warning)`	0 or 1 occurrence of "***warning***"
`+([[:upper:]]|[a-z])`	1 or more ***consecutive*** letters
`@([0-9]|abc)`	1 digit or "***abc***"
`!(err*|fail*)`	Word cannot start with "***err***" or "***fail***"

Figure 1-16. More Shell Metacharacters AL321.0

Notes:

These will process in the shell, as in while using *ls* or *cd* or *lpr*, etc.

Quoting Metacharacters

Stops normal shell metacharacter processing, including metacharacter expansion

- To form strings

"double quotes"	remove the special meaning of all shell metacharacters ***except*** for the $, ` (backquote), and \

- To form literal strings

'single quotes'	remove any special meaning of the characters ***within*** the single quotes

- For a literal character

\character	removes the special meaning of the character ***immediately following*** the \

Figure 1-17. Quoting Metacharacters AL321.0

Notes:

Try these examples: (assume you are the user 'chris' and you have a file called "table1" in the current directory).

1. echo $HOME t*
   \/home/chris table1
2. echo '$HOME ta*'
   *$HOME ta**
3. echo "$HOME t*"
   *\/home/chris t**

In #2, we used single quotes. Single quotes tell the shell to ignore the special meaning of all metacharacters between the quotes. We get everything back literally.

Why did #3 expand the variable, and not the wildcard? Double quotes make the shell ignore the special meaning of all metacharacters **except** for the $, ' (backquote), and \. In #3, the double quotes allow the $ to expand, but the * is NOT an exception listed above, so it will not expand.

Process I/O

- Every process has a file descriptor table associated with it

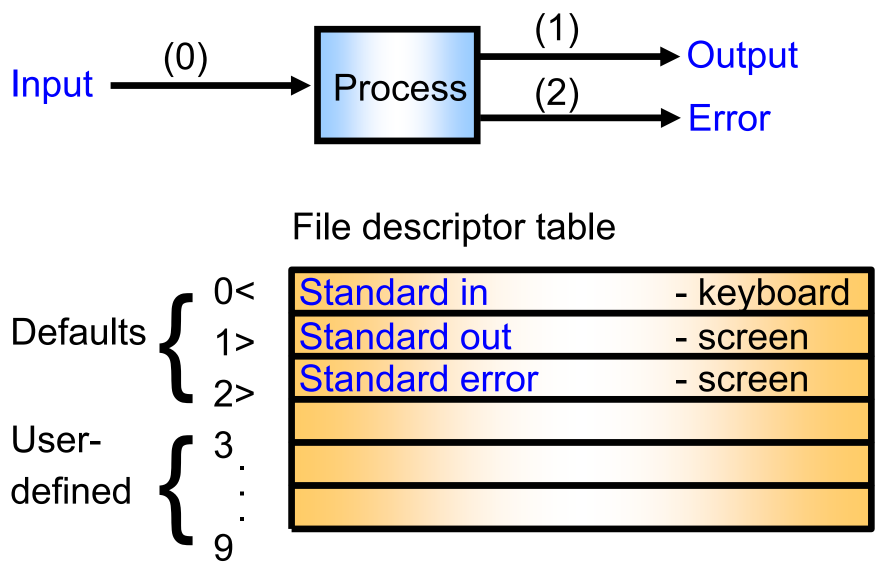


Figure 1-18. Process I/O AL321.0

Notes:

You can define how the file descriptors 3 to 9 are handled. You might want to use descriptor 3 to output to a named file, while 4 outputs to a printer device file. Remember that your screen is addressed through its device file, for example, */dev/tty0*, for both reading of input and displaying of output.

Remember that the device file */dev/tty* always refers to your keyboard or screen.

The defaults for the first three file descriptors can be changed as we will see next.

Input Redirection

Redirecting standard input from a file: **<**

command < filename

```
$ mail marty
Subject: Hello
A letter to see if you are still with us.
<Ctrl-d>
$ _
$ mail -s "Hello" marty < letter
$ _
```

Input may also be given inline. This is called a ***HERE*** document.

```
command << END
text
...
END
```


Figure 1-19. Input Redirection AL321.0

Notes:

In the first example, the user is creating a note to send to the user marty. In the second example, the file *letter* has previously been created using an editor such as */usr/bin/vi.* The file descriptor “`0`” is changed so that input is taken from the named file. It is possible to write "*0*<", but the file descriptor number is usually omitted.

HERE documents are usually seen in scripts. You could use the HERE document syntax for the first *mail* example. The “>” in front of each HERE document line is the shell secondary prompt; shell prompts are configurable (see unit 2 for example). This will also work in a shell script, allowing input to come from the text of the script between the END markers rather than from a file.

Note that the final *END* marker is on a line by itself. You could use any string of characters to mark the ends, but the words *END* or *EOF* or *HERE* seem appropriate. A space must separate the chosen marker from “<<“.

If “-” follows the “<<“, that is “`<<- END`”, leading tabs are ignored in the input text. A “\” will prevent substitutions from taking place. Otherwise, you can refer to variables and substitute command values.

Output Redirection

Redirecting standard output to a file: **>**

command > filename

```
$ ls /home/chris
data_file script2 script3 shell_prog table1
$ _

$ ls /home/chris > listing
$ _
```

Redirecting standard error output to a file: **2>**

command 2> filename

```
$ cat /home/chris/printout
cat: 0652-050 Cannot open printout.
$ _

$ cat /home/chris/printout 2> errors
$ _
```


Figure 1-20. Output Redirection AL321.0

Notes:

In this example, the files *listing* and *errors* are created, or overwritten if the file already exists.

It is permissible to write `command 1> filename`, but the `1` is usually omitted. However, for redirecting error output, the `2` is mandatory.

To redirect other I/O descriptors, use the syntax `n>`, where `3<=n<=9`

Note that the number in the error message is unique for each type of message and product.

Output Appending

Appending standard output to a file: **>>**
command >> filename

```
$ wc -l /home/chris/script3
   42 /home/chris/script3
$ _

$ wc -l /home/chris/script3 >> line_count
$ _
```

Appending standard error output to a file: **2>>**
command 2>> filename

```
$ wc -c /home/chris/characters
wc: 0652-755 Cannot open characters.
$ _

$ wc -w /home/chris/words/ 2>> errors
$ _
```


Figure 1-21. Output Appending AL321.0

Notes:

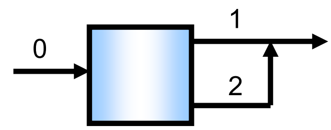
The *line_count* file is appended to — the original contents remain intact.

It is permissible to write `command 1>> filename`. Again, appending to other I/O descriptors uses the `n>>` syntax.

Association

File descriptors can be joined, so that they output to the same place

```
command > file 2>&1
```



Redirects standard error to join with standard out

What do you think these command do?

```
$ cat message_file 2>&1 > errors_file
$ cat message_file 1>&2
```


Figure 1-22. Association AL321.0

Notes:

The order of association is significant. If we had put *command 2>&1 > file*, the error output would appear at the default destination for the standard output, while the standard output goes to the file.

The cmd 1>&2 syntax is used often in shell scripting in order to create your own error messages in your shell scripts.

Consider this: Your user runs your shell script (named prog) and sends error messages to an error file (ex: prog 2> errors). You want to echo to the screen that the user did something incorrect, ex: echo “You did not provide enough arguments”. If you want your message to go to the users *errors* file, you must use the 1>&2 syntax. (that is, echo “you did not provide enough arguments” 1>&2)

Setting I/O or File Descriptors

The built-in shell command `exec` allows you to

- Open
- Associate
- Close

file descriptors

`$ exec n>of`	Opens output file descriptor ***n*** to file "***of***"
`$ exec n<if`	Opens input file descriptor ***n*** to read file "***if***"
`$ exec m>&n`	Associates output file descriptor ***m*** with ***n***
`$ exec m<&n`	Associates input file descriptor ***m*** with ***n***
`$ exec n>&-`	Closes output file descriptor ***n***
`$ exec n<&-`	Closes input file descriptor ***n***

Figure 1-23. Setting I/O or File Descriptors AL321.0

Notes:

Once executed, each of the above settings remains active for the duration of the shell. Settings for file descriptors 0, 1 and 2 remain active in subsequent shells. They are reset by using *exec* to run a replacement shell or command.

There is no way to list the current configuration of file descriptors for the shell.

Setting I/O Descriptor Examples

To open file descriptor 3 for output to Lee's ***out*** file and file descriptor 4 to Lee's ***err*** file

```
$ exec 3> /home/lee/out
$ exec 4> /home/lee/err
$ date >&3
$ ls /home/lee 2>&4
```

To associate output to file descriptor 3 with file descriptor 4

```
$ exec 3>&4
$ wc -l /home/lee/script3 >&3
$ wc -l /home/lee/table1 >&4
```

To close file descriptors 3 and 4

```
$ exec 3>&-
$ exec 4>&-
```


Figure 1-24. Setting I/O Descriptor Examples AL321.0

Notes:

File descriptor 3 is redirected by the association step, so that output to file descriptor 3 is logged in Lee's *err* file — rather than the original *out* file destination. At the end of the example, Lee's *out* file contains only the date command output. Lee's *err* file contains both the listing of Lee's home directory and *wc* command outputs.

Pipes

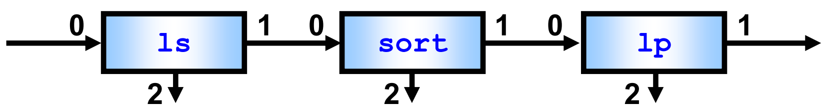
Commands can be joined, so one inputs into the next

```
command1 | command2 | command3
```

Gives a command ***pipeline***

```
$ ls /home/robin | sort -r  |  lp
```

sorts the file list into reverse order, and prints it



Pipelines may have a branch using the ***tee*** command file descriptor

```
$ ls /home/francis | tee raw_list | sort -r | lp
```

saves the unsorted list in the file raw_list

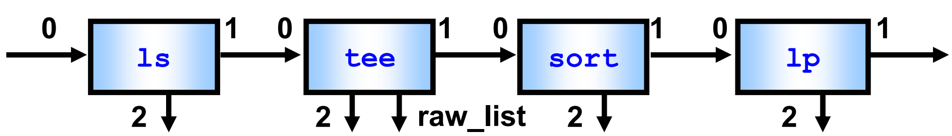


Figure 1-25. Pipes AL321.0

Notes:

A command which takes input from its standard input and outputs to standard output after processing is called a filter. All but the last command in a pipeline is run in a subshell.

There is a 32 KB limit on the amount of data passing along the pipeline. If a command generates more than 32 KB of output it must sleep until the next command processes some of the data; then it can awaken.

Commands can be sequenced with semi-colons, but there is no interaction between them:

```
cd /home/robin ; pwd
```

The tee command is quite useful particularly if you want to view output and keep it for later use. To append to an existing file with tee, use the *-a* option.

By default, only standard output goes over the pipe. Standard error still comes to the screen (unless redirected). How can it be set up so the standard error goes over the pipe also (for instance to record in a log file)?

Answer: **cmd 2>&1 | tee logfile**

Command Grouping { } and ( )

To combine the output of several commands: `{ }` or `( )`

```
{ command ; command ... ; }
```

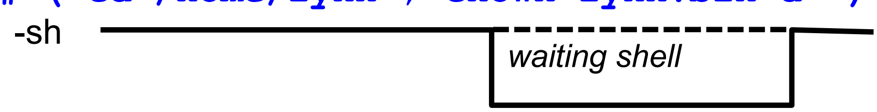
- Runs commands in the current shell
- Directory (or environment) changes remain in effect


```
# { cd /home/lynn ; chown lynn:bin s* ;}
```

```
( command ; command ... )
```

- Does not change your current environment

```
# ( cd /home/lynn ; chown lynn:bin d* )
```



This leaves the working directory unchanged on completion

Figure 1-26. Command Grouping { { and ( ) AL321.0

Notes:

{ } Shell built-in commands can be run in the current shell if they appear in “{ }” parentheses. Because it is ***not*** a subshell, changes to the environment ***do*** affect the main shell.

Input and output redirection can be applied to the grouped commands after the parentheses, for example:

```
{ command1 ; command2 ;} > /dev/null 2>&1
```

The semicolons are mandatory following each command within the { }.

( ) Even shell built-in commands can be run in the subshell if they appear in “( )” parentheses. As it is a subshell, changes to the environment do not affect the main shell.

The semicolons allow the commands to appear on the same line, you could have new lines instead:

```
( command1
command2 )
```

Background Processing

Execute command in the background: **&**

```
$ sleep 999  &
```

Waiting for the end...

```
$ date
Mon Dec 31 11:59:59 EST 2007
$ wait
```

When all background processes have finished

```
$ _
```

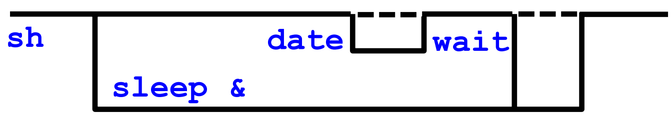


Figure 1-27. Background Processing AL321.0

Notes:

You can specify a process id number or shell job number to wait for instead of waiting for all background processes. The wait command is a shell built-in command. It completes with the same exit status as the background task. Wait can also wait for a specific job to complete and return its status. We shall learn about a command's exit status in Unit 3.

You will not be notified that the job is completed until the next time you press ENTER.

Shell Job Control

The shell assigns job numbers to background or suspended processes

- The **jobs** command lists your current shell processes and their job ids
- **<Ctrl-z>** suspends the current foreground job
- **bg** runs a suspended job in background
- **fg** brings to foreground a suspended or background job
- Jobs can be stopped with the **kill** command
- The **disown** command can be used in ksh93

kill, **fg** and **bg** work with the following arguments:

pid	process ID
%job_id	job ID
%% **%+**	current job
%-	previous job
%command	match a command name
%?string	match string in command line

Figure 1-28. Shell Job Control AL321.0

Notes:

The *jobs* command has three options:

-l	Lists process ids along with the job ids,
-n	Lists only jobs that have stopped or exited since last notified,
-p	Lists only the process group.

The disown command is a built in command in ksh93. It allows you to disassociate a background job from the current shell. The effect is that the job is not killed when the shell exits. It can be compared to the nohup command, but is used after the job is already running. Syntax: disown %job#

Job Control Example

```
$ cc -o RUNME program_in.c
...    After some time running this long compilation...
Ctrl-z
[2] + 5692 Stopped (SIGTSTP)    cc -o RUNME program_in.c
$ jobs
+ [2] Stopped (SIGTSTP)         cc -o RUNME program_in.c
- [1] Running                   sleep 999 &
$ bg %+
[2] cc -o RUNME program_in.c
$ jobs
+ [2] Running                   cc -o RUNME program_in.c
- [1] Running                   sleep 999 &
$ kill %cc
[2] + 5692 Terminated           cc -o RUNME program_in.c
$ fg %1
sleep 999
$ _
Completing the sleep in the foreground...
$ jobs
$ _
```


Figure 1-29. Job Control Example AL321.0

Notes:

Command Substitution

Command substitution allows you to use the output of a command or group of commands:

- In a variable assignment
- In part of an argument list

Bourne, Korn, and Bash	`variable=`command``
-- or --	
Korn and Bash	`variable=$(command)`

Nesting is possible but can be ***EXTREMELY*** confusing:

```
var=`cmd1 \`cmd2 \\\`cmd3\\\` \` `
          -- or --
var=$(cmd1 $(cmd2 $(cmd3) ) )
```


Figure 1-30. Command Substitution AL321.0

Notes:

Bourne, Bash, and Korn shells have available the first form using grave accents, more usually called back quotes. The second is Korn shell specific syntax. Clearly nesting is easier with the Korn shell form.

If you use a *case* statement with the Korn Shell `$( ... )` command substitution, you must use the optional “*(* “in front of each pattern. Be careful to leave spaces around brackets, to avoid confusion with the double parens form of the *let* command -- as in (( )) .

Substituted commands run in subshells, but you can use redirection in place of a command.

Command substitution is helpful in generating reports “real time”.

Ex:

print “Today is $(date)”
print “There are $(who|wc -l) users on the system”
print “There are $(ps -ef|wc -l) processes running”

Command Substitution Examples (1 of 2)

Here is command substitution in action...

```
$ d=$(date)
$ print $d
Fri Feb 29 02:29:00 EST 2008
$ _
$ print "Contents of a file" > tmp_file
$ c=`cat tmp_file`
$ r=$( < tmp_file )                no command, no Sub-Shell
$ print "Cat: $c"
Cat: Contents of a file
$ print "<: $r"
<: Contents of a file
$ _
```


Figure 1-31. Command Substitution Examples (1 of 2) AL321.0

Notes:

Inside the backquotes (grave accents), a backslash normally only removes the special meaning of: \, ' or *$*.

Between backquotes that are themselves double quoted, the backslash also removes the special meaning of a double quote, for example:

```
var="output $(print \"text to print\") "
```

Command Substitution Examples (2 of 2)

Can you explain ***exactly*** what is happening here?

```
$ print "Most recent file: $(ls -t | head -1)"
Most recent file: tmp_file
$ _
$ print "Today is $(date)"
Today is Sat July 07 07:07:07 EDT 2007
$ _
```


Figure 1-32. Command Substitution Examples (2 of 2) AL321.0

Notes:

The first example will list all files, sorted by time, and then show only the top line, or file name, after the “: “.

The second example executes the “date” command and uses the output within the print statement.

Command Line Editing and Recall

vi option for the Korn Shell and emacs for the Bash Shell give:

- Command line editing
- Command recall

```
$ set -o vi or set -o emacs
```

For vi simply press **ESC** to enter editing mode:

- **h** to move the cursor left
- **l** to move the cursor right
- **-** or **k** fetches commands from the history file
- **+** or **j** if you go too far back
- Plus other **vi** commands to perform line editing

For emacs:

- Arrows work, DELete and BackSpace work, else **<Ctrl-b>**, **<Ctrl-f>**, **<Ctrl-d>**
- Up arrow to fetch previous command
  (Check out the Student Notes for more fun stuff!)

Figure 1-33. Command Line Editing and Recall AL321.0

Notes:

Appendix A, at the back of your notes, contains a detailed reference for the "*vi*" command. Below are the special "*vi*" sub-commands that work with `"set -o vi"` editing of a command line:

\ Filename completion.
Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended.
If the match is unique, a "/" is appended if the file is a directory and a space is appended if the file is not a directory.

* Appends an asterisk to the current word and attempts filename generation.
If no match is found, it rings the bell.
Otherwise, the word is replaced by the matching pattern and input mode is entered.

= Lists the file names that match the current word as if an asterisk were appended to it.

_	(Underscore) Optionally preceded by a *Count*, for example, "*5*_". Causes the *count*th word of the previous command line to be appended and input mode entered. The last word of the previous command line is used if *Count* is omitted.
/	Command search Searches command history for this string. Use "n" to go to the next, "N" to go to the previous.
@Letter	Searches the *alias* list for an *alias* named Letter. If an *alias* of this name is defined, its value is placed into the input queue for processing.
#	Sends the line after inserting a "#" in front of the line. Useful for causing the current line to be inserted in the history without being executed.
<Ctrl-c>	Terminates the `"set -o vi"` edit.
<Ctrl-j>	(New line) Executes the current line, regardless of the mode.
<Ctrl-l>	Line feeds and prints the current line. Has effect only in control mode.
<Ctrl-m>	(Return) Executes the current line, regardless of the mode.

Any other command in vi:

set -o emacs (used as default in most Bash shells)

<Tab>	filename completion
<Tab><Tab>	filename completion list
<Ctrl-b>	move back one character
<Ctrl-f>	move forward one character
<Ctrl-d>	delete 1 character forward
<Del>	Delete 1 character behind
<Esc> ~	username completion
<Esc> $	variable completion
<Esc> @	hostname completion
<Esc> !	command completion
<Esc>	completion

Any other command in emacs.

Question: In what file would you put “set -o vi” or “set -o emacs” to make it permanent?

Answer: .kshrc or .bashrc (whatever your ENV or BASH_ENV variable is set to). NOT .profile because then command line editing would only work in your login shell.

Checkpoint

1. What type of file is `/dev/tty3`?
2. How could we find out a file type?
3. How can we get `.kshrc` to run in an explicit Korn Shell?
4. How can we specify the first character in a file name to be uppercase?
5. How can we ignore error messages from a command?
6. How do you make the normal output of a command appear as error output?
7. How can we group commands, in order to redirect the standard output from all of them?
8. What will `kill 1` do?
9. If you have submitted a job to run in foreground, how could you move it to background?
10. How would you set up a command line recall facility?

Figure 1-34. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

Unit Summary

- AIX shells
- Hierarchical file-system
- Filenames and types
- Shell scripts
- Invoking shells
- Shell metacharacters: Expansion and quoting
- Redirection -- **<** and **<<** input, **>** and **>>** output, **2>** and **2>>** error
- Setting file descriptors
- Pipes and tees
- Command grouping
- Background processes
- Shell job control
- Shell command editing

Figure 1-35. Unit Summary AL321.0

Notes:

Unit 2. Variables

What this unit is about

This unit describes how to set and reference variables. In addition, we present positional parameters and variable inheritance.

What you should be able to do

After completing this unit, you should be able to:

- Set and reference variables
- Access positional parameters
- Analyze variable inheritance

How you will check your progress

- Checkpoint questions
- Machine exercises

Unit Objectives

After completing this unit, you should be able to:

- Set variables
- Reference variables
- Use positional parameters
- Shift arguments
- Set positional parameters
- Use shell parameters
- Understand inheritance
- List shell variables
- List environment variables

Figure 2-1. Unit Objectives AL321.0

Notes:

Setting Variables

To assign a value to a variable: `name=value`

```
$ var1=Fri
$ _
```

To "unset" the value to a variable:

```
$ unset var1
```

To protect a variable against further changes:

```
readonly name=value
      - or -
typeset -r name=value

$ readonly var1=Sun
$ var1=Mon
ksh: var1: This variable is read only
$ _
$ readonly -p
```

displays full list

Figure 2-2. Setting Variables AL321.0

Notes:

There are no spaces around the “=”. Variable assignments remain in effect for the duration of the shell.

It is a good idea not to use uppercase names for your variables; the shell does, and there could be conflicts. There are no shell limitations on the length of a variable name, or the length of its contents.

A *readonly* variable cannot be assigned a new value or be *unset*. The shell itself can change *readonly* variables, for example, if you make any shell-set variable *readonly*. The command is a shell built-in. To initialize a *readonly* variable, set the value when declaring the variable. The *typeset* command is a shell builtin (not available in all shells). There will be more in later units. You cannot assign values with the *readonly* command in the Bourne shell.

With no further arguments, both *readonly* and *typeset -r* list the variables that are readonly.

With AIX Version 4, a new option *readonly -p* gives a list of readonly variables in the format “`readonly var=val`”.

Referencing Variables

To reference a variable, prefix name with a **$**

```
$ print $var1
Fri
$ _
```

To separate a variable reference from other text use: **${ }**

```
$ print The course ends on $var1day
The course ends on
$ print The course ends on ${var1}day
The course ends on Friday
$ _
```


Figure 2-3. Referencing Variables AL321.0

Notes:

The *print* command is a Korn shell builtin command. You can get the same functionality by using either the */bin/echo* command provided by the AIX operating system, or the *echo* command builtin to the shells. In bash, you must use echo.

Unset variables have no value, and so nothing is printed when you reference them in a *print* command.

Positional Parameters

Parameters can be passed to shell scripts as arguments on the command line

```
$ params.ksh arg1 arg2
```

- "arg1" is positional parameter number 1
- "arg2" is positional parameter number 2
- Others are unset

They are referenced in the script by:

- `$1` to `$9` for the first nine
- `${10}` to `${n}` for all after the first nine

Figure 2-4. Positional Parameters AL321.0

Notes:

In the Bourne shell you cannot reference more than nine arguments at once.

If you want to pass arguments that begin with a “-” or “+”, you can use the convention that “--” marks the end of options for a command or script. You will see how to use this in Unit 5 with the option processing command *getopts*. For example:

```
params.ksh -- -arg1 +arg2 arg3
```

This will prevent “*-arg1*” being treated as an option rather than an argument.

Setting Positional Parameters

In a shell script the **set** command can:

- Change the values of positional parameters
- Unset positional parameters previously set

```
$ cat first.ksh
print $1 $2 $3
set apple banana
print $1 $2 $3

$ first.ksh a b c
a b c
apple banana

$ _
```


Figure 2-5. Setting Positional Parameters AL321.0

Notes:

set is a shell built-in command. Here, parameter 3 was cleared (or unset) by the use of the *set* command.

The shell command *unset* can be used to clear a variable from memory and remove it:

```
unset var1
```

or

```
unset -v var1
```

AIX Version 4 introduced the *-v* option for *unset*. This option corresponds to the POSIX standard recommendation.

Variable Parameters

Shell scripts set a number of other shell parameters:

$# The number of positional parameters set

$@ Positional parameters in a space separated list

$* Positional parameters in a list separated by the first Internal Field Separator (the default is a space)

In double quotes, $@ and $* behave differently:

"$@" = `"$1" "$2" "$3" . . .`

"$*" = `"$1 $2 $3 . . . "`

Figure 2-6. Variable Parameters AL321.0

Notes:

The *IFS* (Internal Field Separator) variable contains the Field Separator characters. In most shells these characters default to Space, Tab, and Newline.

We shall see more of *IFS* later.

Some Shell Parameters

Shell parameters that remain fixed for the duration of the script:

$0	The (path)name used to invoke the shell script
$$	The **P**rocess **ID** (PID) of current process (shell)

Parameters set as the script executes commands:

$!	The PID of the last background process
$?	The return code from the last command executed

Figure 2-7. Some Shell Parameters AL321.0

Notes:

As *$0* remains fixed for the duration of a shell script, it is not affected by the *shift* command seen earlier. It is the pathname used to invoke the script.

$- Shell options used to invoke the shell, for example, -r

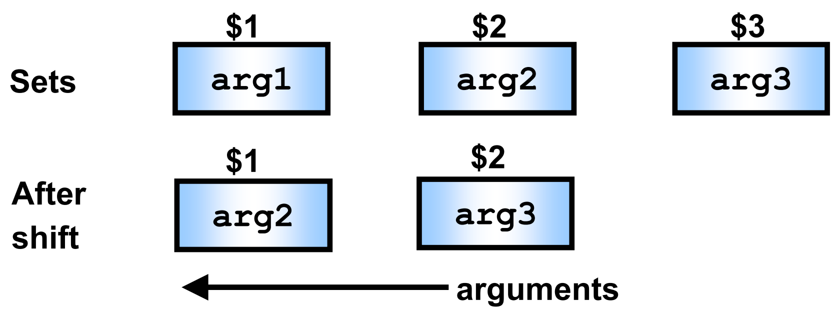
If you see "*ism*" in the *Shell Options,* these are the usual default options for a command login shell. The option letters mean the shell is in interactive mode, it uses STDIN for commands, and it has job control (m=monitor) enabled respectively. We shall see all of the options in Unit 5.

We shall see more of *$?* in the next unit. You should note that PID is a very common abbreviation used in documentation and commands.

Shifting Arguments

In a shell script the **shift** command moves arguments to the left:

```
$ params.ksh arg1 arg2 arg3
```



- Discarding the first or leftmost argument
- Decrementing the number of positional parameters
- Allowing Bourne shell to reference more than 9 arguments

Figure 2-8. Shifting Arguments AL321.0

Notes:

You can specify a number of parameters for *shift*, for example,

```
shift 3
```

moves three parameters to the left, discarding the leftmost three. The shell provides *shift* as a built-in command.

The shift command is very helpful inside of loops, which we will see later.

Parameter Code Example

So, let's put all of it into action in a shell script.

```
$ cat second.ksh
print $$
print $0
print "$# PPs as entered"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
shift
print $0
print "$# PPs after a shift"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$@"
print 'Set "$@" - parameters in double quotes'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$*"
print 'Set "$*" - parameters space separated'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"

$ _
```


Figure 2-9. Parameter Code Example AL321.0

Notes:

On the next page we shall see what this does.

Parameter Output Example

Here's what it does.

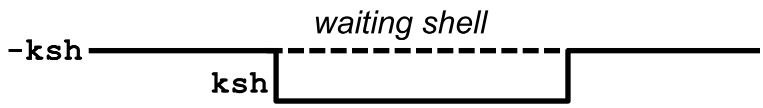
```
$ second.ksh Atlanta NYC "Chicago and D.C."
4687
second.ksh
3 PPs as entered
PP1=Atlanta PP2=NYC PP3=Chicago and D.C. PP4=
second.ksh
2 PPs after a shift
PP1=NYC PP2=Chicago and D.C. PP3= PP4=
Set "$@" - parameters in double quotes
PP1=NYC PP2=Chicago and D.C. PP3= PP4=
Set "$* "- parameters space separated
PP1=NYC Chicago and D.C. PP2= PP3= PP4=
$ _
```


Figure 2-10. Parameter Output Example AL321.0

Notes:

This Shell and the Next

What happens to variables when you spawn a Subshell?



Unless you export variables, they will not be passed on.

$ set	to list all variables and values
$ export var	export variable var so that it will
- or -	be inherited by subshells, or
$ typeset -x var	use typeset in the Korn shell
$ declare -x var	use declare in the Bash shell
$ export	to list variables that are exported,
- or -	other variables will be unset in a
$ typeset -x	subshell

Figure 2-11. This Shell and the Next AL321.0

Notes:

Attributes of variables are also inherited — like a *readonly* attribute for example.

In the Korn shell you can use the *export* command to set variable values and export them in one step: For example,

```
$ export var=value
```

or

```
$ typeset -x var=value
```

With AIX Version 4 "`export -p`" gives a list of exported variables in the format "`export var=val`".

The *set* command also reports variable settings in single quotes.

The *env* command performs a similar function to the "*export*" built-in command above, but it is an external operating system command.

You will see more about *typeset* in later units.

Inheritance Example - The export Command

Let's see inheritance in action...

`$ x=324`	We can set a variable **x**
`$ print "$$: X=$x"`	in our current shell
`4589: X=324`	
`$ ksh`	In a subshell, **x** is unset
`$ print "$$: X=$x"`	- there is no value to print
`4590: X=`	
`$ _ Ctrl-d`	Returning to the main shell...
`$ print "$$: X=$x"`	
`4589: X=324`	**x** will have its value restored
`$ export x`	If we export **x**, a subshell
`$ ksh`	can inherit the value of **x**
`$ print "$$: X=$x"`	
`4591: X=324`	
`$ x=3`	If we change **x** from the
`$ _ Ctrl-d`	subshell the change does
`$ print "$$: X=$x"`	not affect the main Shell
`4589: X=324`	

Figure 2-12. Inheritance Example - The export Command AL321.0

Notes:

Important points to note here are:

- To use a value in a script or subshell, it **must** be *export*ed.
- You can **never** pass a value back (or up) from a subshell to a calling shell with an exported variable.
- Unset or unexported variables have a NULL (string) value.

What do you think would happen if we opened a second subshell AFTER we set x=3? Would the value pass down to the second subshell?

Korn Shell Variables

Korn Shell sets certain variables each time they are referenced:

SECONDS	seconds since Shell invocation
RANDOM	random number in the range 0 to 32767
LINENO	current line number within a Shell Script or function
ERRNO	system error number of the last failed system call – a system-dependent value!

Figure 2-13. Korn Shell Variables AL321.0

Notes:

Every variable above holds integer values. None of the above are exported by default.

Notice that each variable name is in uppercase. Shell variable names are generally uppercase. To avoid conflicts, you should avoid using uppercase variable names.

You can set *SECONDS* to an initial value, so that subsequent references yield that value plus the number of seconds since shell invocation, for example,

```
$ SECONDS=35
```

You can initialize the *RANDOM* number sequence by assigning a value to the variable, for example,

```
$ RANDOM=$$
```

You can clear the ERRNO variable by assigning the value zero to it, that is,

```
$ ERRNO=0
```

Other shell variables (which we shall see next) also lose their special meanings if they are *unset*.

Environment Variables

Several variables define the environment of a Shell:

CDPATH	a search path for the **cd** command
HOME	your home directory
IFS	input field separators (***space, tab, newline***)
PATH	the system command search path
PS1	the primary Shell command prompt
PS2	a secondary prompt for multi-line entry
PS3	prompt for the **select** command
PS4	debug prompt for **ksh** with the **-x** option
PWD	the current working directory
OLDPWD	previous working directory for **cd -**

Figure 2-14. Environment Variables AL321.0

Notes:

The shell sets default values for *PATH*, *PS1* and *PS2*. The shell normally does not set a value for *SHELL*. The AIX login process sets the value for *TERM;* this is taken from the Object Data Manager (ODM).

You can customize the shell prompts.

- In PS1 "!" is replaced by the command number
- Use single quotes to include shell set variables

$ PS1=!' $SECONDS : '

Shell defaults for PS1 through PS4 are:

```
PS1='$ '
PS2='> '
PS3='? '
PS4='+ '
```

Korn Environment Variables (1 of 2)

Korn Shell specific features require environment variables:

COLUMNS	screen width
LINES	screen length
SHELL	the pathname of the shell
TERM	the terminal type (selects terminfo file)
ENV	program/script to be sourced for each new shell
FCEDIT	an editor for the **fc** command
FPATH	a search path for function definition files
HISTFILE	your history file
HISTSIZE	limit of history commands accessible

Figure 2-15. Korn Environment Variables (1 of 2) AL321.0

Notes:

None of the above are exported by default.

`COLUMNS` defaults to 80, `LINES` to 24. Both of these variables control window editing and, as we shall see in Unit 4, the *select* command.

By default, `ENV` is not set.

`HISTFILE` implicitly defaults to `$HOME/.sh_history`, while `$HISTSIZE` has the value 128.

`LC_COLLATE` is normally set to "*En_GB*" or "*en_GB*" in the UK, and "*En_US*" or *"C(POSIX)"* in America.

Unit 5 describes the *fc* command, and Unit 7 the function of `FPATH`.

Korn Environment Variables (2 of 2)

LC_COLLATE	sorting sequence for pattern ranges
MAIL	the name of your mail file
MAILCHECK	mail check frequency (default 600 seconds)
MAILMSG	the "you have new mail" message
PPID	the parent process ID
REPLY	set by **select** command and the **read** command if no argument is given
EDITOR	the editor for command line editing
VISUAL	a visual editor – overrides **EDITOR**

Figure 2-16. Korn Environment Variables (2 of 2) AL321.0

Notes:

The shell sets default values for *IFS* and *MAILCHECK*. The *login* program sets up the *HOME* variable. The shell normally does not set a value for *MAIL*.

MAILCHECK holds an integer value, *unset* removes the special meaning.

TMOUT holds an integer value. The shell default value of zero means no timeout. The Korn shell waits one minute before dying after issuing a warning message and a beep.

We shall see more of *PS3* and *REPLY* in Unit 4; *OPTARG* and *OPTIND* in Unit 5.

The Bourne shell provides further environment variables:

NLFILE	file with extended character set details
NCLTAB	sort collating sequence
SHACCT	command history for use by system accounting
TIMEOUT	minutes to Bourne shell timeout — which is without warning!

Korn Shell 93 Variables

- There are several additional variables and variable meanings in ksh93. Here are a few:

`TMOUT`	also used to timeout of select menu
`.sh.version`	identifies version of the shell -- use **`${ }`**

Figure 2-17. Korn Shell 93 Variables AL321.0

Notes:

Use ${ } with .sh.version.

Bash Environment Variables

- Bash variables are the same unless noted here:
- **BASH_ENV** instead of **ENV** program to be sourced for each new interactive shell
- **PS1** has additional features (see below)
- Some additional variables in bash:

BASH_VERSION	version number for the instance of bash
HOSTNAME	name of current host
HOSTTYPE	describes machine bash is running on
SHLVL	shell level - how deeply you are nested

Figure 2-18. Bash Environment Variables AL321.0

Notes:

In Korn shell, we can set PS1='$PWD =>' to have our working directory reflected in the prompt. In bash, we can also use the following prompt string customizations:

\d	date
\H	hostname
\h	hostname up to first "."
\T	time in 12 hour HH:MM:SS
\t	time in HH:MM:SS
\u	username
\h	hostname
\n	newline
\w	current working directory
\W	basename of current working directory

Examples: PS1="\w=> "

PS1="\n[\u@\h \w]\n\$ " Can you describe this?

Checkpoint

1. How could we use positional parameter 3 in a shell script?
2. Which variable contains the number of positional parameters?
3. How can we change the value of a variable set in a different process?
4. What is the variable **IFS**?
5. How can we reset **PS1** to show the current directory?
6. By setting a variable, how can we have a command recall facility?

Figure 2-19. Checkpoint AL321.0

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.

Unit Summary

- Setting variables
- Referencing variables
- Using positional parameters
- Shifting arguments
- Setting positional parameters
- Using shell parameters
- Understanding inheritance
- Shell variables
- Environment variables

Figure 2-20. Unit Summary AL321.0

Notes:

Unit 3. Return Codes and Traps

What this unit is about

This unit provides the students with the opportunity to review basic testing concepts and explore shell scripting using return codes, signals, and traps.

What you should be able to do

After completing this unit, you should be able to:

- Identify conditional execution statements
- Analyze return codes and signals
- Test variables or files for specified conditions
- Handle signals in a script with traps

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Recognize return values
- Identify exit codes
- Identify conditional execution
- Use the `test` command
- Understand compound expressions
- Examine file `test` operators
- Use numerical expressions
- Understand string expressions
- Understand shell `test` operators
- Use shell `[[ ]]` expressions
- Handle signals
- Understand sending signals
- Understand catching signals

Figure 3-1. Unit Objectives AL321.0

Notes:

Return Values

Each command, pipeline, or group of commands returns a value to its parent process.

`$?` contains the value of the return code

- **zero** means success
- **non-zero** means an error occurred

The single value returned by a pipeline is the return code of the last command in the pipeline.

For grouped commands – that is, `( )` or `{ }` – the return code is that of the last command executed in the group.

Figure 3-2. Return Values AL321.0

Notes:

The “0” does not mean “zero errors” -- it simply means the previous command was successful.

Exit Status

A shell script provides a return code using the **exit** command.

`$ print $$`	check the shell process id
`879`	
`$ ksh`	start a new subshell
`$ print $$`	and check its process id
`880`	
`$ exit`	quit the subshell
`$ print $?`	and print the return code
`0`	
`$ print $$`	
`879`	
`$ ksh`	begin another subshell
`$ print $$`	
`890`	
`$ exit 101`	exit with a value to set
`$ print $?`	the return code
`101`	
`$ print $$`	
`879`	
`$ _`	

Figure 3-3. Exit Status AL321.0

Notes:

The *exit* command is a shell built-in command.

Conditional Execution

A return code (or exit status) can be used to determine whether or not to execute the next command.

- If command1 **is** successful execute command2

```
command1 && command2

$ rm -f file1 && print file1 removed
```

- If command1 **is not** successful execute command2

```
command1 || command2

$ who|grep marty || print Marty logged off
```


Figure 3-4. Conditional Execution AL321.0

Notes:

The *-f* option to the *rm* command prevents interactive questions being displayed when file permissions do not allow read or write for the named file. The command returns status "0" only if the named file is deleted.

The operating system command *who* lists the users logged on to the system. The *grep* operating system command searches standard input for the pattern specified. Only if a match is found will it return an exit status "0" (the return code).

The test Command

The test command is used for expression evaluation

```
test expression
        - or
[ expression ]
```

- Returns zero if the expression is true
- Returns non-zero if the expression is false

The Korn and Bash shells provide an improved version

```
[[ expression ]]
```

- Easier syntax
- Includes same functionality as `test`
- Additional operators
- Shell expansions prevented

Figure 3-5. The test Command AL321.0

Notes:

Test operators form expressions that we shall see later.

The keywords *true* and *false* have their obvious meanings.

If you use metacharacters with *test* or `[ ]` they will be expanded: with `[[ ]]` they are only expanded if they appear as a pattern in a string expression; refer to shell `[[ ]]` Expressions later in this unit.

The Korn and Bash shells provide additional operators for use with the *test* command compared to the Bourne shell, as well as further operators for use with the `[[ ]]` syntax.

File Test Operators

File status can be examined using several operators.

Operator:	*True if ...:*
-s file	file has a size greater than zero
-r file	file exists and is readable
-w file	file exists and is writable
-x file	file exists and is executable
-u file	file exists and has the SUID bit set
-g file	file exists and has the SGID bit set
-k file	file exists and has the SVTX sticky bit set
-e file	file exists
-f file	file exists and is an ordinary file
-d file	file exists and is a directory
-c file	file exists as a character special file
-b file	file exists and is a named pipe file
-p file	file exists and is a named pipe file
-L file	file exists and is a symbolic link

Figure 3-6. File Test Operators AL321.0

Notes:

Note a file will appear to be writable even though it is within a read-only file system. Only the file access control list is examined, not the file system status.

An executable directory file is a directory that can be searched; you may *cd* to the directory.

The operator "-e" was added with AIX Version 4.

The above tests can be done any of the following three ways:

```
test -s file
[ -s file ]
[[ -s file ]]
```

Numeric Expressions

For arithmetic expressions and integer values use:

Expression:	True if ...:
`exp1 -eq exp2`	`exp1` is equal to `exp2`
`exp1 -ne exp2`	`exp1` is not equal to `exp2`
`exp1 -lt exp2`	`exp1` is less than `exp2`
`exp1 -le exp2`	`exp1` is less than or equal to `exp2`
`exp1 -gt exp2`	`exp1` is greater than `exp2`
`exp1 -ge exp2`	`exp1` is greater than or equal to `exp2`

Figure 3-7. Numeric Expressions AL321.0

Notes:

Numerical values are compared using the above operators. If variable *x* has been assigned a numerical value, you test *x* as follows:

```
$ x=2
$ test $x -eq 1
$ [ $x -eq 2 ]
$ [[ $x -eq 3 ]]
```


String Expressions

To examine strings use one of the following:

Expression:	*True if ....:*
`-n str`	`str` is non-zero in length
`-z str`	`str` is zero in length
`str1 = str2`	`str1` is the same as `str2`
`str1 != str2`	`str1` is not the same as `str2`

Figure 3-8. String Expressions AL321.0

Notes:

Character strings are compared using the above operators. If variable *TERM* has been assigned a character string, you test *TERM* as follows:

Examples:

```
[ -n $TERM ]
test -n $TERM
[[ -n $TERM ]]
```

To avoid syntax errors from *test* or from the shell, you usually surround the $variable with double quotes — as in "$*TERM*". This avoids problems testing with NULL strings in particular.

More Shell Test Operators

The shell provides a number of additional **`test`** operators.

Expression:	*True if ....:*
`file1 -ef file2`	**`file1`** is another name for **`file2`**
`file1 -nt file2`	**`file1`** is newer than **`file2`**
`file1 -ot file2`	**`file1`** is older than **`file2`**
`-t des`	**`file`** descriptor **`des`** is open and associated with a terminal device

Figure 3-9. More Shell Test Operators AL321.0

Notes:

You can use metacharacters in filenames.

More examples:

-O file	file exists and its owner is the effective user id
-G file	file exists and its group is the effective group id
-S file	file exists as a socket special file

Shell [[ ]] Expressions

When using the shell **`[[ ]]`** syntax there are a few extra expressions.

Expression:	*True if ...:*
`str = pattern`	**`str`** matches pattern
`str != pattern`	**`str`** does not match pattern
`str1 < str2`	**`str1`** is before **`str2`** in the ASCII collation seq.
`str1 > str2`	**`str1`** is after **`str2`** in the ASCII collation seq.
`-o opt`	option **opt** is on for this shell

You may use shell metacharacters in the patterns.

Figure 3-10. Shell [[ ]] Expressions AL321.0

Notes:

Examples:

```
[[ abc == *c ]]     true
[[ abc != ?c? ]]    true
[[ abc < def ]]     true
```

Remember that shell metacharacters may be used in patterns.

Also, due to locale settings, some string comparisons may not give the answers you expect. This is particularly true if `LANG` is not set to `en_US`.

Although “=” does work, it is considered obsolete in ksh93, the “==” is the most recent preferred syntax.

Compound Expressions

For the `[ ]` or `test` command

`exp1 -a exp2`	binary **and** operation
`exp1 -o exp2`	binary **or** operation
`! exp`	logical negation
`\(  \)`	used to group expressions

For the `[[  ]]` syntax

`exp1 && exp2`	true if both expressions are true - the second is **only evaluated** if the first is true
`exp1 || exp2`	true if either expression is true - the second is **only evaluated** if the first is false
`! exp`	logical negation
`(  )`	used to group expressions

Figure 3-11. Compound Expressions AL321.0

Notes:

Notice that with *test* or `[ ]` you need to escape shell metacharacters (like parentheses). Compound expressions are valuable with multiple test operators and tests.

Examples:

```
test $# -eq 2 -a $? -eq 0
[ $# -eq 2 -a $? -eq 0 ]
[[ $# -eq 2 && $? -eq 0 ]]
```

Practice Test

```
$ [[ -s /etc/passwd || -r /etc/group ]]
$ print $?              True or False?

$ test -f /etc/motd -a ! -d /home
$ print $?              True or False?

$ x="005"
$ y=" 10"
$ test "$y" -eq 10
$ print $?              True or False?

$ [ "$x" = 5  ]
$ print $?              True or False?

$ [[  -n "$x"  ]]
$ print $?              True or False?

$ test  -S  /dev/tty0
$ print $?              True or False?

$ [[  1234 = +([0-9])  ]]
$ print $?              True or False?
```


Figure 3-12. Practice Test AL321.0

Notes:

Signals

- The kernel sends ***signals*** to processes during their execution
  - Certain system events issue signals when they
    - Run out of paging space
    - Receive special key sequences like **`<Ctrl-c>`**
  - The **`kill`** command sends a specific signal to a process

Figure 3-13. Signals AL321.0

Notes:

To terminate a foreground process you can press the Interrupt key sequence (normally <Ctrl-c>). Your input causes the relevant *signal* to be sent to your foreground process by the system.

The *kill* command is the only way to terminate a background process.

What You Can Do with Signals

Signals sent to processes may be:

• Caught	the process deals with it
• Ignored	nothing happens
• Defaulted	use default *handlers*

Figure 3-14. What You Can Do with Signals AL321.0

Notes:

Signals are a form of simple interprocess communication. If a process takes default action on a signal, this normally means terminate (die!). If you do not want the default you can either ignore or trap the signal.

The Kill Command

- To send a signal to a process:

  kill -sig pid -or- **kill -s sig pid**

- To list all defined signals

  kill -l **(lowercase "ell")**

- To list a specific signal

  kill -l # **(replace # with a number)**

- To list the signal that caused an exit error

  kill -l $?

Figure 3-15. The Kill Command AL321.0

Notes:

The current process group means all processes started from, and including, the current login shell. The *-s sig* and *-l $?* options were introduced with AIX Version 4.1.

To signal the current process group:

kill -sig 0 -or- kill -s sig 0

To send a signal to all of your processes, except those with PPID 1 (do not use if you are root):

kill -sig -1 -or- kill -s sig -1

The full signal list is held in /usr/include/sys/signal.h.

We know in many cases the default action is for the process to die upon receipt of the signal. However, some signals are ignored. A list of useful signals follows on the next pages.

Note: The output of kill -l in ksh93 is not as verbose as ksh(88).

Signal List (1 of 2)

Here is a list of some useful signals.

Signal:	Event:
0 EXIT	issued when a process or function completes (shell specific)
1 HUP	you logged out while the process was still running – sent to sub-shells too
2 INT	interrupt pressed **<Ctrl-c>**
3 QUIT	quit key sequence pressed **<Ctrl-\>**
9 KILL	special 'force' signal, cannot be ignored
15 TERM	default kill command signal
18 TSTP	process suspend **<Ctrl-z>**

Figure 3-16. Signal List (1 of 2) AL321.0

Notes:

The INT (2) signal key sequence may vary with terminal type. For early versions of AIX and IBM-3151 ASCII terminals it is <Ctrl-c>; other common sequences are <Ctrl-BackSpace> and <Delete>.

The default key configurations for a terminal can be changed through *smit* — terminal attributes — or by using the *stty* command for the session. To change the *QUIT* sequence to <Ctrl-t>: `$ stty quit ^t`

Signal names include a "SIG" prefix to the signal codes listed above, that is, *SIGDANGER*. By default background processes stop if they attempt to read from a terminal. To set this behavior for background processes that attempt to write to a terminal, use:

```
$ stty tostop
```

You should avoid the `KILL` signal except as a last resort. If you send a `KILL` to a process it can never be caught so it is impossible to perform cleanup actions (like removing lock files etc.). Signals KILL (9), SEGV (11), STOP (17) and SAK(63) may not be trapped under AIX V3 or V4.

Signal List (2 of 2)

Signal:		*Event:*
19	CONT	continue if stopped – issued by **`kill`** to a suspended process before TERM or HUP
29	PWR	power failure imminent – save data now!
33	DANGER	paging space low
63	SAK	you pressed **<Ctrl-x>** and **<Ctrl-r>** the SAK sequence

Figure 3-17. Signal List (2 of 2.) AL321.0

Notes:

A reserved key sequence, called the secure attention key (SAK), allows a user to request a trusted communication path which is part of TCB (Trusted Computing Base).

Catching Signals with Traps

The **trap** command specifies any special processing you want to do when the process receives a signal:

To process signals

```
$ trap 'rm /tmp/$$; print signal!; exit 2' 2 3
```

To ignore signals

```
$ trap '' INT QUIT
```

To reset signal processing

```
$ trap - INT QUIT
```

- or -

```
trap 2 3
```

To list traps set

```
$ trap
```


Figure 3-18. Catching Signals with Traps AL321.0

Notes:

The shell *trap* command allows your script to catch specific signal.

You should use single quotes to enclose the action to protect it from shell expansions, although double quotes may also work. Single quotes are preferred because the shell scans the action twice; once when it prepares to run the trap command, and once when the shell executes the trap. Think about when you want variables, and so forth, to expand. In the shell signal names or numbers may be used, but names are more portable. For the Bourne shell only numbers are allowed.

The syntax of the trap command is:

```
trap "actions to do instead of signals default actions" sig1 sig2 sig3 ..
```

The signals trapped can be system or user initiated. Once a signal is set to be ignored, subshells also ignore that signal, and cannot then trap the signals themselves.

Notice that you need to explicitly use *exit* if you want to terminate the script from within a *trap*. Otherwise, after the trap executes, control is passed back to the next command in the script.

Trap Example

```
#!/usr/bin/ksh
# ps_monitor
# monitor processes using ps -elf at intervals
# of 30 seconds for 2 minutes.  If interrupted,
# a summary report is produced by executing
# psummary.
#
trap 'print $0: interrupt received ;
        ./psummary ;
        exit' 2 3 15
ps -elf > /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
trap - 2 3 15
```


Figure 3-19. Trap Example AL321.0

Notes:

Which directory does the `trap` command use for the `./psummary` command/script? Do you think a relative or full path name would be best in this situation?

Checkpoint

1. How can you tell whether a command you have just entered was successful?

2. How can you test if file *datafile* is non-empty?

3. How can you check if you have been logged on for more than 20 minutes, and if so, print out a suitable message?

4. How could you log off, using the kill command?

5. If you are a DBA is this a desirable command to terminate the <oracle_server>? **`kill -KILL <oracle_server>`**

6. What does this command do? **`trap echo you did <Ctrl-c> 2`**

7. How could you get <Ctrl-c> to log you off?

Figure 3-20. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

Unit Summary

- Return values
- Exit status
- Conditional execution
- The `test` command
- Compound expressions
- File `test` operators
- Numerical expressions
- String expressions
- Shell test operators
- Shell `[[ ]]` expressions
- Signals
- Sending signals – `kill` command
- Catching signals – `trap` command

Figure 3-21. Unit Summary AL321.0

Notes:

Unit 4. Flow Control

What this unit is about

This unit presents flow control using conditional loops and decision making.

What you should be able to do

After completing this unit, you should be able to:

- Generate if-then-else statements
- Generate while/until loops
- Understand and use for loops
- Create case and select constructs
- Leave loops prematurely

How you will check your progress

Accountability:

- Checkpoint questions
- Machine exercises

Unit Objectives

After completing this unit, you should be able to:

- Generate the `if - then - else` construct
- Generate conditional loops with `until` and `while`
- Understand specific value iteration with `for`
- Use multiple choice pattern matching with `case`
- Use the `select` command for menus
- Use `break` and `continue` in Loops
- Identify Doing Nothing – the `null` Command

Figure 4-1. Unit Objectives AL321.0

Notes:

The Simple *if - then - else* Construct

```
if expression1
then
   commands to be executed if expression1 is
      true

else
   commands to be executed if all expressions
      are false
fi
```

```
if  [[ "$UID" -lt "100" ]]
then
   export PS1="# "
else
   export PS1="$ "
fi
```


Figure 4-2. The Simple *if - then - else* Construct AL321.0

Notes:

In one of its simplest forms, the if-then-else construct is as easy as it sounds.

In the example above, the script will check to see if your User ID is between 0 and 99 and if it is, the prompt will be a pound sign, or that of a ‘system user’. Otherwise, the prompt will be that of a ‘regular user’.

The Full *if - then - else* Construct

```
if [[ "$1" = "-a" ]]
then
   author=y ; control=n
   version=n
elif "$1" = "-c"
then
   author=n ; control=y
   version=n
elif "$1" = "-v"
then
   author=n ; control=n
   version=y
else
   author=n ; control=n
   version=n
fi       < /usr/local/defaults
```


Figure 4-3. The Full *if - then - else* Construct AL321.0

Notes:

You do not always need an *else* part, but there can be only one. Any number of *elif ... then* segments may be included.

As soon as a true expression is found, the corresponding block of commands is executed. Then the flow of the program will continue after the closing *fi* statement. The return value of the construct is that of the last command block executed, or true if none was executed.

if Example

Here is a simple if construct:

```
#!/usr/bin/ksh
# Usage: goodbye username
#
if [[ $# -ne 1 ]]
then
      print  "Usage is:  goodbye username"
      print  "Please try again."
      exit 1
fi
rmuser $1
print "O.K., $1 is removed."
```

When we run "goodbye", this is what we get ...

```
$ goodbye
Usage is: goodbye username
Please try again.
$ goodbye pete
O.K., pete is removed.
$ _
```


Figure 4-4. if Example AL321.0

Notes:

We have used the shell `[[ ]]` syntax for the expressions above, but it could just as easily have been the older `[ ]` or *test* command. In fact any command, or even group of commands, could be used as an expression. Metacharacters are expanded and variable references are allowed. It is the return value of the expression that is used to decide true or false: zero = true.

Conditional Loop Syntax

```
until cc prog.c
do
        vi prog.c
done
```

```
while [ "$x" -lt 3 ]
do
  lsps -a >> ./statfile
  df >> ./statfile
  let x=x+1
done    # should be "< ./statfile" here
```


Figure 4-5. Conditional Loop Syntax AL321.0

Notes:

The expression after the *until* can be any command. The return code of the command will be checked to decide if the loop should continue.

The *while* loop will be executed only if the expression evaluates true. An *until* loop follows the reverse logic — executing only if the expression is false.

Both *until* and *while* return the value of the last loop command executed, or true if no loops were executed. The program continues after the *done* statement.

If the "open file" instruction (< file) were inside the do-done loop, it would be opened and closed six times. If the 'open' is listed after the loop, as the referenced "./statfile" file is, the file is opened once when the while loops starts and closed once when the while loop closes. This is what is meant by redirected output for the whole of the loop.

while true Example

The Script "forever" is a tough cookie!

```
#!/usr/bin/ksh
# An endless loop with a trap for INT QUIT TSTP
trap 'print "hasta la vista - baby!"' 2 3 18
while true
do
        print  "I'll be back."
        sleep 10
done
```

`$  forever`	
`I'll be back.` `I'll be back.` `I'll be back.`	every ten seconds the script speaks!
`<Ctrl-c>`	an attempt to stop it...
`hasta la vista - baby!` `I'll be back.` `I'll be back.`	invokes the trap, and it carries on.

Figure 4-6. while true Example AL321.0

Notes:

The *true* and *false* are shell built-ins that are available for use as expressions.

This script traps normal keyboard kill sequences, so that you must *kill* it from another terminal.

for Loop Syntax

```
for identifier in word1 word2 ...
do
   commands using $identifier
   more commands
done
```

```
for identifier    # equivalent to: for identifier in "$@"
do
   commands using $identifier which takes values from the
      positional parameters
done    # optional < filename
```


Figure 4-7. for Loop Syntax AL321.0

Notes:

Perhaps a better description of the *for* loop is a specific value iteration command. It iterates over a parameter list (the set of values).

The *for* command sets the *identifier* variable to each of the values from the *word* or positional parameter list in turn, and executes the command block. Execution ends when the *word* or positional parameter list is exhausted. The return value is that of the last block command executed, or true if none were.

The word list in the first form of the *for* command can contain metacharacters for file name expansion. It can also contain command substitution, which we will learn later. The words in the word list are separated out by IFS - the input field separator. The IFS variable can be changed to a different delimiter if the words are separated by something other than a space.

You can apply redirection to the whole of the loop.

for - in Loop Example

Here we have a quick tidy-up to delete files:

```
$  for  varfile  in  *.tmp
>  do
>   rm -f $varfile
>  done
$  _
```

Why use the option `-f` ?

What else could be `test`ed?

Figure 4-8. for - in Loop Example AL321.0

Notes:

The word list in the *for* command has been formed by metacharacter expansion into the file names from the current directory that end in *.tmp*.

for Loop Example

The sample Script "getprice.ksh" will look up the price list:

```
#!/usr/bin/ksh
# getprice.ksh - select price from "pricelist" file
# for each item entered on the command line
# Usage: getprice item1 item2 ...
#
for item
do
        grep -i "$item" /home/cashier/pricelist
done


$ getprice.ksh  "Shock Absorbers"  "Air Filter"
Front Shock Absorbers      49.99
Rear Shock Absorbers       59.99
Air Filter                 10.99
$ _
```


Figure 4-9. for Loop Example AL321.0

Notes:

By omitting the *in word1 word2 ...* part of the *for* command syntax, the command takes its list from the positional parameters — as if you had specified `in "$@"`.

Arithmetic for Loop

The arithmetic for loop is available in ksh93 and bash.

```
for (( initialize; test; increment ))
do
  commands
done
```

Example:

```
for (( num=0; num <5; num++ ))
do
  mv file${num} file${num}.bkup
done
```


Figure 4-10. Arithmetic for loop AL321.0

Notes:

The above example renames file0 to file0.bkup, file1 to file1.bkup, and so forth.

This syntax is not available in bsh.

The incrementing is done after the iteration, and every iteration after that. The initialization and test are done before the first iteration.

The case Statement

```
case word in
( pattern1 | pattern2 | ... )
        action      ;;
(*)     default     ;;
esac
```

```
case $identifier in
(pattern1)                command1
                          more_commands ;;
(pattern2 | pattern3)     commands    ;;
(*)                       commands    ;;
esac
```


Figure 4-11. The case Statement AL321.0

Notes:

The *case* statement compares the *word* with each *pattern* in turn. If a match is found, the corresponding action is performed. The double semi-colon syntax marks the end of an action. Null actions are allowed. Multiple patterns can be associated with an action — each separated by a pipe character. Patterns can contain metacharacters. Spaces around a pattern are ignored.

There must be at least one pattern block and it is a good idea to include a final "catch-all" pattern the metacharacter "*". Once a match is found, or after all patterns have been checked, the program continues after the *case* statement.

The Korn and Bash shells allow an optional open bracket "(" at the start of each pattern group, so that you can use the command grouping ( ) syntax around a *case* construct. The Bourne shell does not allow this.

case Code Example

A guessing game of sorts:

```
#!/usr/bin/ksh
# Usage:  match string
# To see how lucky you are feeling today

case "$1" in
   Ace )         print "You are really close."    ;;
   King )        print "Missed it by that much."  ;;
   Queen )       print "Finally!"                 ;;
   Jack )        print "Maybe next time."         ;;
   Ten|10 )      print "Getting closer."          ;;
   * )           print "Guess again."             ;;

esac
```


Figure 4-12. case Code Example AL321.0

Notes:

You can use combinations of variable references and fixed text to form a *word* to be matched if you like.

Note where you specify the catch-all pattern. Note the use of the "|" with the ten "or" 10.

In the above example, we are trying to match the value of a variable to the pattern choices. We can also try to match the output of a command to the pattern choices using command substitution. We will learn command substitution later, this example is listed here for reference later.

```
case $(command) in
   pattern|pattern) action;;
   pattern) action;;
   *) action;;
esac
```

.. where any shell or system command can be put inside the $( ).

case Code Output

A casino dealer in the making?

```
$ match Three
Guess again.

$ match Jack
Maybe next time.

$ match Ace
You are really close.

$ match King
Missed it by that much.

$ match Queen
Finally!
```


Figure 4-13. case Code Output AL321.0

Notes:

Mini Quiz

1. True or False. There can be any number of **`elif`** statements in an **`if - then - else`** construct.

2. How does one redirect for the whole of an **`until`** or **`while`** loop?

3. True or False. The statement: "**`for identifier`**" takes its input from positional parameters.

Figure 4-14. Mini Quiz AL321.0

Notes:

The Shell select Syntax

```
select identifier in word1 word2 ...
do
  commands using $identifier usually containing a case
    statement
done
```

```
select identifier   # equivalent to: select identifier in "$@"
do
  commands using $identifier from positional parameters
    usually containing a case statement
done
```


Figure 4-15. The Shell select Syntax AL321.0

Notes:

The shell *select* command displays the *word* or *positional parameter* list as items in a numbered menu, output is to standard error. The environment variables *LINES* and *COLUMNS* control output size.

The *PS3* prompt is displayed as a prompt for you to enter the number of your choice. The variable *REPLY* is set to the character string that you enter. The variable *identifier* is set to the *word* or *positional parameter* value corresponding to your selection. If you choose an unlisted item, or enter any other unidentified text, *identifier* is set to null.

The command block is executed for each selection. A null selection re-displays the menu and *PS3* prompt without executing the command block.

The select command only terminates if it encounters an end-of-file (<Ctrl-d>) input, *exit*, *break* or *return*. The program continues after the *done* statement. The return value is that of the last block command, or true if no commands were executed.

The select command does not exist in the Bourne shell. The select syntax has serious bugs before bash version 1.14.3.

select Code Example

To help identify animals we have a "barn.ksh" Shell Script:

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3="Pick an animal: "
select  animal  in  cow  pig  dog  quit
do
        case $animal in
        (cow)   print "Moo"
                ;;
        (pig)   print "Oink"
                ;;
        (dog)   print "Woof"
                ;;
        (quit)  exit
                ;;
        ('')     print "Not in the barn"
                ;;
        esac
done
```


Figure 4-16. select Code Example AL321.0

Notes:

The environment variable *LINES* defaults to 24, while *COLUMNS* is 80 by default. This is fine for the screen we are using, so they were left at their default values. The *PS3* prompt default is “#? “.

The *case* catch-all is executed when the *select* command doesn't recognize your selection, and the animal variable is set to null.

select Output Example

Running "barn.ksh" we can choose an animal to examine ...

```
$ barn.ksh
1) cow
2) pig
3) dog
4) quit
Pick an animal: 1
Moo
Pick an animal: 2
Oink
Pick an animal: 3
Woof
Pick an animal: 8
Not in the barn
Pick an animal: 4
$
```

Do you think setting PS3 to "Pick an animal" was a good choice?

Figure 4-17. select Output Example AL321.0

Notes:

The menu would be redisplayed if we just press return without making a selection. As we make more and more selections, the menu is of course disappearing as the screen scrolls upward.

More on Select

- In the previous example, the selected choice (for example **`cow`**) was stored in **`$animal`**, however, the input from the user was stored in **`$REPLY`**
- Using the **`$REPLY`** variable makes the select syntax a bit more flexible as seen on the next page
- In ksh93, the **`TMOUT`** variable can be set to a number of seconds. The select loop will timeout if no input is received within the **`TMOUT`** seconds set.

Figure 4-18. More on Select AL321.0

Notes:

Select Example Using $REPLY

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3 = "Pick an animal:"
Select animal in cow pig dog quit
do
  case $REPLY in
  cow|COW)          print "Moo"        ;;
  pig|PIG)          print "Oink"       ;;
  dog|DOG)          print "Woof"       ;;
  quit|QUIT)        exit               ;;
  *)                print "Not in the barn"      ;;
  esac
done
```


Figure 4-19. Select Example Using $REPLY AL321.0

Notes:

By doing a case on $REPLY instead of $animal, the case will try to match up with whatever input the user typed in, whether it was a number or animal name. This allows for slightly more flexibility.

exit the Loop

In the Korn shell script /usr/sbin/snap

```
...
if [ "$badargs" = n ]
then
  if [ "$found" = y ]
  then
    if [ -r "$destdir/$component/$component.snap" ]
    then
    more $destdir/$component/$component.snap
    else
    echo "^Gsnap:  $destdir/$component/$component.snap not found"
    exit 25
    fi
  fi
else
    usage
    exit 26
fi  ...
```


Figure 4-20. exit The Loop AL321.0

Notes:

The ***exit*** causes the script to end. A status number can be attached to the *exit* to inform a calling script of its success, failure, or otherwise.

break the Loop

The **break** command jumps out of **do . . . done** loops:

- Exits from the smallest enclosing loop
- Jumps out a specified **number** of layers/loops

```
break  number
```

```
select  choice  in  Backup  Restore  Quit
do
  case $choice in
  (Backup)   find . -print|backup -iqf /dev/rfd0
  ;;
  (Restore) restore -xqf /dev/rfd0
  ;;
  (Quit)     break
  ;;
  ('')       print "What ?" 2>&1
  ;;
  esac
done
```


Figure 4-21. break the Loop AL321.0

Notes:

Following a *break* the program continues after the *done* statement just as if the command was complete.

This is applicable to *until*, *while*, *for*, and *select* constructs.

continue the Loop

The **continue** command begins the next iteration of a **do . . . done** loop:

- Starts at the top of the smallest enclosing loop
- Begins again a specified *number* of layers/loops out

```
continue  number

$ for File in *
> do
> if [[ -d $File ]]
> then
>       continue
> fi
> file $File
> done
$ _
```


Figure 4-22. continue the Loop AL321.0

Notes:

Following a *continue* the command block is aborted, the next value is selected, and the next iteration of the command block is begun — just as if it had completed the command block in full. So in the above example, when a directory file is found in the current directory, it is ignored: all other files are classified using the *file* command.

continue is applicable to *until*, *while*, *for* and *select* constructs.

In the example above, the commands are entered against the dollar prompt, rather than in a script.

If the number provided to the *continue* command is greater than the current block nesting depth, the shell prints a warning and execution continues at the outermost block.

null Logic

Sometimes you require a command, but you don't actually want to do anything – a **NULL** command

```
:           # a COLON character
```

For example:

```
sys_call parameter1 parameter2
if  [[  $?  -eq  0  ]]
then
     # Debug slot
     :
else
    print $0: Error: command failed
    exit $ERRNO
fi
```

{ without the null command ":" this would be illegal syntax

Figure 4-23. null Logic AL321.0

Notes:

Constructs like *if*, *until*, *while*, *for* and *select* require at least one command block. When you're debugging a program, null command slots can be handy — you can easily put in another print command without needing to change the logic of the enclosing construct.

You can have arguments to the null command, which will be expanded, and thus may affect the current environment. The return value is zero (true), so you can use the null command in place of the *true* keyword.

Program Logic Constructs Example

Here's a Script to delete empty files:

```
#!/usr/bin/ksh
# Usage: delfile file1 file2 ...
while [[ $# -gt 0 ]]
do
        if [[ -f "$1" ]]
        then
                if [[ ! -s "$1" ]]
                then
                        rm "$1" && print "$1" deleted
                else
                        print "$1" not deleted 2>&1
                fi
        elif [[ -d "$1" ]]
        then
                print "$1" is a directory
        else
                print "$1" is a special file
        fi
        shift
done
```


Figure 4-24. Program Logic Constructs Example AL321.0

Notes:

Here's delfile in action...

```
$ delfile /dev/null /tmp/jesse file1 file2 $PWD
/dev/null is a special file
/tmp/jesse deleted
file1 deleted
file2 not deleted
/home/jesse is a directory
$ _
```

A file can be deleted without write permission to it; write permission on its directory is all that is required. An attempt to delete a file will fail if its directory has no write access. The above example attempts to delete empty files and will report successful deletions.

No allowance is made for the non-existence of the named file; a special file is assumed.

Checkpoint (1 of 2)

1. What is wrong with this fragment of shell script?

```
if [ "$x" -eq 5 ]
then
      echo $x
elif [ "$x" -eq 3 ]
else
      echo "x is only 3"
      exit
fi
```

2. What is the fundamental difference between a `while` and an `until` construct?
3. How could we write an endless loop?
4. What syntax would we use to perform a loop a finite number of times, resetting an identifier (variable) each time the loop goes through?

Figure 4-25. Checkpoint (1 of 2) AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

Checkpoint (2 of 2)

5. Which construct is best suited to allow conditional processing, based on pattern matching?
6. What would the following lines produce?

```
select word in To be or not to be
do
      :
done
```

7. Which construct is best used within the previous `do-done`? block?
8. How can we terminate one iteration of a loop and commence the next?
9. How can we abruptly terminate all iterations of a loop but continue further processing in a shell script?

Figure 4-26. Checkpoint (2 of 2) AL321.0

Notes:

6.

7.

8.

9.

Unit Summary

- The `if - then - else` construct
- Conditional loops with `until` and `while`
- Specific value iteration with `for`
- Multiple choice pattern matching with `case`
- The `select` command for menus
- Leaving loops – `exit` and `break`
- Beginning again – `continue`
- Doing nothing — the `null` command – `:`

Figure 4-27. Unit Summary AL321.0

Notes:

Unit 5. Shell Commands

What this unit is about

Creating an interactive script is a common activity for Korn shell programming. This unit focuses on the print and read interactive commands as well as the set command.

What you should be able to do

After completing this unit, you should be able to:

- Use the print, echo, and read commands
- Understand and use getopts
- Control the programming environment using the fc and set commands
- Use additional shell commands

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use the `print` and `echo` command
- Use special printing characters
- Use the `read` command
- Understand option and argument processing with `getopts`
- Use history manipulations with `fc`
- Use the `set` command
- Use shell options with `set`
- Use shell invocation
- Use built-in commands
- Use shell commands provided by AIX

Figure 5-1. Unit Objectives AL321.0

Notes:

The print Command (ksh 88 and ksh93)

The **print** command is the Korn shell output mechanism:

`print argument ...`	prints arguments to standard output separated by spaces
`print - argument ...`	to print arguments that look like options
`print -r argument ...`	RAW mode – do not interpret print's special characters (listed on next page)
`print -R argument ...`	equivalent to "**-**" and "**-r**"
`print -uN argument ...`	output sent to file descriptor **N**
`print -s argument ...`	output to the shell history file only

Figure 5-2. The Print Command (ksh 88 and ksh 93) AL321.0

Notes:

The Bourne shell and Bash shell provide the *echo* command as an equivalent for *print*. The Korn shell provides *echo* as a built-in command for backward compatibility, however, it has no options,

Arguments are optional; if you omit them, a blank line is printed, except with *-n*.

Redirecting output with the *-u* option can be more efficient than using individual redirection.

Options may be mixed in the usual way, except: no option can follow *-r*, only *-n* can follow *-R*.

The -n option provides for no trailing newline after output -- see “\c” on next page.

There is another *print* option: `print -p argument ...` to output to a co-process. We shall not be looking further at this.

Special print Characters

Backslash character sequences have special meaning (except in raw mode)

`\a`	Alarm - ring the terminal bell
`\b`	Backspace
`\c`	Print without trailing newline (same as `print -n`)
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\0xxx`	Character with octal code `xxx` (up to three octal digits)

Figure 5-3. Special print Characters AL321.0

Notes:

The backslash character is used to escape the special meaning of the following character. The shell removes it when an unquoted command line is processed, so that you need two successive backslashes to pass a single backslash to the *print* command. The *print* command interprets the shell processed line following the conventions listed above.

If you surround *print* arguments with quotes (single or double), the shell does not strip away backslashes.

All of the above special characters work with the Bourne shell *echo* command. However, *\a* was not provided with *echo* prior to AIX Version 4.

When you use *\c*, it must be the last option specified, that is *\r\c* not *\c\r*.

The echo Command (Bash)

The **echo** command is the Bash shell output mechanism:

- The **echo** special characters in bash are the same as the **print** special characters in ksh (**\a**, **\b**, **\c**, and so forth)
- To use the **echo** special characters, you must use the **-e** option
- On some systems, **-e** is the default. In this case **-E** turns off the interpretation of special characters (similar to **-r** in **print**)

echo also has **-n** option to omit trailing newline after input

Figure 5-4. The echo Command (bash) AL321.0

Notes:

The Bash shell does not have the *print* command builtin. Instead, it has the *echo* command builtin. The differences between *echo* and *print* are listed above.

For octal codes of characters, use \xxx where xxx is the octal code.

echo does not support the -s or -u option.

print Examples

When you use the **print** command, here's what you get.

```
$ print "Line 1\n\tLine2"
Line1
      Line 2
$ print 'One quarter = \0274'
One quarter = ¼
$ print 'Backslash = \0134'
Backslash = \
$ print -r 'hi\\\\there 1'          with -r and quotes
hi\\\\there 1
$ print -r hi\\\\there 2            with -r and no quotes
hi\\there 2
$ print 'hi\\\\there 3'             with no -r and quotes
hi\\there 3
$ print hi\\\\there 4               with no -r and no quotes
hi\there 4
$ _
```


Figure 5-5. print Examples AL321.0

Notes:

In the 'hi\\\\there 1' example, raw mode and single quotes prevent backslash interpretations by both the shell and the *print* command. For the second example, there are no quotes used so the shell processes the line and removes two backslashes. *print* processes the resulting line but as raw mode was specified the output is two backslashes.

For the third ('hi\\\\there 3') example, the shell passes the input without processing to the print command. The command interprets the four backslashes passed to it from the shell and prints two, since two backslashes input result in a single backslash output from *print*. Finally, the fourth example has both the shell and then the *print* command interpreting the entered line; the shell removes two backslashes and, without raw mode, two backslashes result in a single backslash output.

These examples are trying to point out that with metacharacters, there are often several “entities” that want to expand the metacharacters. In these examples, both the shell and the print command have a backslash as a metacharacter. Be sure to use quoting correctly so the correct entity expands the metacharacter.

In bash, use \274 and \134.

The printf Command - An Advanced Print

- The `printf` command allows for more powerful formatting.
- The `printf` commands comes built-in with ksh93. However, AIX also has a version of `printf` (/usr/bin/printf) that can be used from bash and ksh88.
- Syntax:
  - `printf format-string [arguments ...]`
- Examples:

Examples	Results:
`printf "#%10s#\n" title`	# Title#
`printf "#%-10s#\n" title`	#Title #
`printf "%.5f" 123456.789`	123456.78900

Figure 5-6. The printf Command - An Advanced Print AL321.0

Notes:

Examples of format specifiers:

%s string
%d decimal integer
%f, %e floating point format ([-] add, precision, [-d]d. precision [+ - dd])
%o unsigned octal value

printf can be used to display a simple string like print, however *printf* does not automatically supply a newline. You must use \n.

The read Command

To get input while a shell script is running, use **`read`**:

```
read variable ...
```

The **`read`** command reads a line from its standard input

- Assigns input words to the variables
- Set remaining variables to null if too few words
- Set last variable to the remainder of the words if too few variables

For the Korn and bash shells, if no variables are specified, the **`REPLY`** variable is set to the whole input line

Figure 5-7. The read Command AL321.0

Notes:

Standard input for the *read* command is normally the keyboard.

Words are delimited by a character from the *IFS* environment variable: space, tab or newline.

Apart from not using the *REPLY* variable, the Bourne shell *read* command works in the same way.

read Examples

We can use the **read** from the shell prompt as well:

```
$ read var1 var2
123 456 789
$ print "var1 = $var1 \tvar2 = $var2"
var1 = 123          var2 = 456 789
$ read var1 var2
abc
$ print "var1 = $var1 \tvar2 = $var2"
var1 = abc var2 =
$ read
hi there
$ print $REPLY
hi there
$ _
```


Figure 5-8. read Examples AL321.0

Notes:

Remember that you cannot change the value of a *readonly* variable.

The AIX Operating System provides the "*line*" command as an equivalent to the shell commands:

```
( read ; print -R "$REPLY" )
```

If you require input to be taken from a terminal one character at a time, without the need to press return at each input, the *dd* command can be applied:

```
dd if=/dev/tty bs=1 count=$charcount > inputread
```

Here */dev/tty* is a link to the current terminal you are using, and *$charcount* has the number of characters you wish to take as input. In Unit 7 we will learn how to store the output of a command in a variable.

read Command Options

The Korn shell **read** command has some options:

`read -r variable ...`	raw mode – \ is not taken as a line continuation character
`read -uN variable ...`	read from file descriptor N

You can specify a prompt for the command to display on standard error, add a "?prompt" to the first variable

```
read  variable?prompt  variable ...
```

For example, to request a user for a text string:

```
read string?'Please enter a text string'
```


Figure 5-9. read Command Options AL321.0

Notes:

Options above may be mixed in the usual way.

Before AIX Version 4, *read -r* did not require a variable; the *REPLY* variable would be used by default.

It may be more efficient to use the *-u* option, rather than normal command input redirection.

In the read variable? prompt variable example, beware, your prompt is sent to standard error. This is to prevent losing your prompt in a pipeline, however, it can cause your prompt to go somewhere else (for example /dev/null) if the user redirected standard error.

There is another option where you can read from a co-process instead of standard input which we do not discuss further:

```
read -p variable ...
```

read Options for ksh93

`read -A variable`	reads words into an indexed array named *variable*, starting at index 0
`read -d delimiter`	use *"delimiter"* instead of newline
`read -n number`	read, at most, *"number"* bytes
`read -t seconds`	wait *"seconds"* for input, else exit

Figure 5-10. read Options for ksh93 AL321.0

Notes:

We will look at arrays later. This example is listed for future reference:

```
$ read -A name
Lee Lynn Llewellyn
```

This will unset the *name* array first, then set *name[0]* to Lee, *name[1]* to Lynn, *name[2]* to Llewellyn.

read Options for bash

`read -a variable`	reads words into an indexed array named *variable*, starting at index 0
`read -p prompt variable`	similar to read *var?prompt* in ksh
`read -s`	silient mode (no echo)

Figure 5-11. read Options for bash AL321.0

Notes:

We will look at arrays later. This example is listed for future reference:

```
$read -a name
Lee Lynn Llewellyn
```

This will unset the *name* array first, then set *name[0]* to Lee, *name[1]* to Lynn, *name[2]* to Llewellyn.

The -p option can be used in the following manner:

```
read -p "Enter your name" var1 var2
```

Again, the prompt will be sent to standard error (see notes for Korn shell “read var?prompt” for more information).

read Options Examples (1 of 2)

```
#!/usr/bin/ksh
# usage: readrun
# prompt the user for their name
read first?"Enter your name: " last
print "firstname = $first\tlastname=$last"
```

What would the result be for the following?

```
# readrun
Enter your name: Lee
  firstname = __________     lastname = ___________

  Enter your name: Lee Lynn Llewellyn
  firstname = __________     lastname = ___________
```


Figure 5-12. read Options Examples (1 of 2) AL321.0

Notes:

The *read* command cannot control how many words a user types in. But, as a programmer, we can immediately check to see if they typed in enough information. Consider the following:

read first?"Enter your name:" last junk

- First we test if anything got stored in *$junk* with the -*n* or -*z test* option. If there is something in *$junk*, they typed in too much.
- Next we test to see if anything got stored in *$first* and *$last* with the -*n* or -*z test* option. If they are empty, they did not type in enough.

Also notice that you cannot use *print* command special characters in the *read* command prompt string. Instead you would have to split it into two lines as shown:

print -n "Enter your name:\a"
read first last junk

Also, notice in the *read* command prompt string, the cursor stays on the same line as the prompt string.

read Options Examples (2 of 2)

```
#!/usr/bin/ksh
# Usage: readpwd
# Read & print parts of /etc/passwd.
IFS=:
while read name pwd uid guid gecos home shell
do
         print "$name" "$uid" "$guid" "$shell"
done  <  /etc/passwd
```

Here's what happens:

```
$ readpwd
root 0 0 /bin/bash
bin 1 1 /sbin/nologin
daemon 2 2 /sbin/nologin
adm 3 4 /sbin/nologin
...
$ _
```


Figure 5-13. read Options Examples (2 of 2) AL321.0

Notes:

This example shows how input terminates. End of file (*EOF*) for terminal input is normally <*Ctrl-d*>. When the *read* command gets *EOF*, it returns false. The example also shows that you can change the IFS to whatever you need in order for the input to be correct.

Processing Options

Parameters on a script command line are of two types:

- Arguments – used in script
- Options – used to tell the script what to process

General parameter/argument processing is difficult

Consider

```
$ myscript -a -f optionfile argfile
$ myscript -foptionfile -va argfile
```

Shell provides `getopts` as a solution

Figure 5-14. Processing Options AL321.0

Notes:

There is a general convention that options are prefaced by a "-"(sometimes a "+"). Arguments are the remainder of the parameters supplied to the program or script.

Processing arguments passed to programs and scripts is not too difficult provided you have to parse only a small number of cases. The examples indicate two of the possible combinations of permitted options for myscript. Creating code for the two examples given is relatively easy but what happens if a new option is added?

The getopts Command

- The **getopts** command processes options and associated arguments from a parameter list

```
getopts optionstring variable parameter...
```

- Each invocation of **getopts** processes the next option in the **parameter** list (parameter list usually comes from the command line, but can come from within a script)
  – Usually called within a loop
- The **optionstring** lists expected option identifiers
  – If an option identifier requires an associated argument, add a colon (**:**)
- A leading colon in the list suppresses "invalid option" messages by **getopts**

Figure 5-15. The getopts Command AL321.0

Notes:

Usually no *parameters* are specified on the *getopts* command line, so that the positional parameters are processed. A "--" option can be used to mark the end of your option list.

The *getopts* command uses your chosen variable *variable* and *OPTARG* and *OPTIND* to store the results of each processing operation on the parameters. *variable* contains the current option being processed or a "?" if it is not recognized as a valid option. *OPTARG* contains the string for an associated argument where a ":" has been added to an option identifier in *optionstring*.

The index variable *OPTIND* is not normally examined until the end of processing. Whenever a shell script is invoked, the value of *OPTIND* is set to 1. When *getopts* recognizes the end of the options or reads a "--" option, processing of the parameters stops. At this point *OPTIND* indexes the first non-option parameter. By convention (see previous notes) this is the first proper argument.

Option parameters begin with a "+" or "-" and may contain several option identifiers, that is, *-abc*. By convention, identifiers with a minus are used to set options: a plus means unset that option.

getopts Syntax Example

How are options processed when passed to a script?
Assumptions:

- The possible options are **a**, **b** and **c**
- Option **b** is to have an associated argument
- Suppress normal OpSys error messages

Inside the script **getopts** will be used early on:

```
while  getopts  ':ab:c'  flag
do
        identify the values set by getopts
done
```

A correct command line to the script might be

```
$ prog.ksh  +c  -ab  barg  --  arg1  arg2
```

What about?

```
$ prog.ksh  -c -b -a -- arg1 arg2
```


Figure 5-16. getopts Syntax Example AL321.0

Notes:

The second example is tricky. At first glance it looks OK but there is a problem; what is it?

getopts is often used within a while loop. As we know, a *while* loop ends when the return code of the command (in this case -getopts) returns a non-zero exit code. It is important to note that *getopts* returns a non-zero exit code when there are no more options on the command line to process.

In the above example, a, b, and c are the valid options. The colon behind the b indicates that option b must have an associated argument. The colon at the beginning of the list of valid options means *getopts* will suppress the system error message when the user gives an invalid option; this allows the programmer more control of error messaging.

In the above example, *flag* is the name of the variable that will hold each option as it is stripped off of the command line. Upon encountering an error, the *flag* variable will be set to a “?”. If an option takes an argument, the argument is stored in the variable OPTARG. The OPTIND variable is used to keep track of the next command line argument to be processed.

getopts Example

```
#!/usr/bin/ksh
#  Example of getopts
USAGE="usage:  example.getopts.ksh [+-c] [+-v] [-a argument]"

while getopts :a:cv varflag
do
case $varflag in
     a)     argument=$OPTARG ;;
     c)     compile=on ;;
    +c)     compile=off ;;
     v)     verbose=on ;;
    +v)     verbose=off ;;
     :)     print "You forgot an argument for the switch called a.";
            exit ;;
    \?)     print "$OPTARG is not a valid switch" ; print "$USAGE" ;
            exit ;;
    esac
done
print -c "compile is $compile; verbose is $verbose;
print "argument is $argument "
#END
```


Figure 5-17. getopts Example AL321.0

Notes:

The problem on the previous example was that it is not clear whether the "-a" is the associated argument to "-b" or not.

Notice how using a leading ':' in the *getopts optionstring* means doing your own error processing. You do not have to exit with an invalid option but it's usually the best or safest course of action.

In this example, each option will be taken from the command line and stored in a variable called varflag. We then do a “case” on varflag. Notice the “+” is stored with the option but the “-” is not.

How do we get to the actual or proper arguments? Recall that *OPTIND* contains an index to the parameters processed. In particular, after all options have been processed it is “pointing” to the first non-option argument. The usual practice is then to use *shift* to shift over the option parameters by using the index.

`shift (( OPTIND - 1 ))`	This works for Korn shell.
``shift `expr $OPTIND - 1` ``	Use this for Bourne shell..

getopts Notes

- `getopts` does not support options that start with a "**+**" in bash
- `getopts` supports putting a "**#**" after an option letter (in the valid option list) instead of a "**:**" to specify the option's argument must be a number in ksh93
  – Example:
  `:ab#c` **`b`** takes an argument, which must be a number

Figure 5-18. getopts Notes AL321.0

Notes:

The fc Command

The Shell fc command interactively edits and then re-executes portions of your command history file:

`fc start end`	edits and executes a command range -- `start` defaults to the last command -- `end` defaults to the value of start
`-e editor`	to specify an editor other than `$FCEDIT` -- WARNING! The shell default is `/bin/ed`

To re-execute a single command with automatic editing:

```
fc -e - old=new command
```

`old=new`	to swap string `old` with string `new`
`command`	to specify a `command` -- default command is the last command

Figure 5-19. The fc Command AL321.0

Notes:

The *HISTSIZE* environment variable sets the maximum *start finish* range size you can specify — 128 commands by default.

The *fc* command returns the value of the last command executed.

The *r* command is equivalent to *fc -e* -. Beginning with AIX Version 4, *fc -s* is also equivalent to *fc -e* -.

The *fc* command is less often used now but some of the inline command editing may be of interest.

fc Examples - Edit and Execute, List

Ranges may be strings, absolute or relative numbers...

`$ fc`	edit the last command with the `$FCEDIT` editor, and then re-execute
`$ fc cc`	edit the previous command beginning with `cc`
`$ fc -e vi 10 20`	use `vi` to edit history lines `10` to `20`

Automatic editing can specify a command in a similar way

`$ fc -e -`	re-execute last command as it was
`$ fc -e - 2=3 10`	swap `3` for `2` in command number `10`

The ksh fc command lists portions of your command history file:

`$ fc -l start end`	list the specified command range -- the default is the last 16 commands

For example...

`$ fc -l pg grep`	lists commands from the last `pg` to a `grep`
`$ fc -l 15 20`	lists commands `15` to `20`
`$ fc -l -5 -1`	lists the last `5` commands

Figure 5-20. fc Examples - Edit and Execute, List AL321.0

Notes:

The *$FCEDiT* variable defaults to */bin/ed*. When you execute an *fc* "edit" command and you see just a number, you are probably in *ed*. The safest way to exit *ed* is to press ***<Ctrl-d>*** at the "?" prompt.

when you bring the command list into "your favorite editor", you can then make any changes you want, but be aware -- what ever commands you leave in the editor -- whether you "save and exit" or save w/no exit" -- the commands will be sent to the shell. If you truly do NOT want to execute any of the commands, empty the screen, then exit the editor.

The *fc* command always returns true when commands are listed. It is equivalent to the *history* command. Indeed, you will see in Unit 7 that it is an *alias*.

The set Command

We have seen three functions performed by the **set** command:

`set`	lists set variables with their values
`set value ...`	resets the positional parameters
`set -o vi`	enables line recall and editing

This last form sets a shell option. There are several more options to **set**:

- Shell options and settings are listed by `set -o`
- Turn option on using `set -o option` or `set -L`
  (where `L` is an option identifier)
- Turn option off using `set +o option` or `set +L`

Figure 5-21. The set Command AL321.0

Notes:

The Bourne shell also has some of the same options as the Korn shell. The Bourne shell *set* command does not have a "*-o*" option syntax; it uses the single letter option identifiers. Most of the option identifiers explained in the following pages are provided by the Bourne shell, those that are not are noted in the text below.

Korn Shell Options with Set (1 of 2)

Option:	*L*	*Description:*
`allexport`	`a`	automatically export each variable set
`bgnice`		run all background jobs at a lower priority – this is on by default for interactive shells
`ignoreeof`		stops an interactive shell exiting on **<Ctrl-d>** – you must use the `exit` command
`errexit`	`e`	exits if any command returns a non-zero return code
`noclobber`	`C`	stops the shell overwriting existing files with > redirection (`>|` works instead)
`noexec`	`n`	for a non-interactive shell to check syntax without executing commands

Figure 5-22. Korn Shell Options with Set (1 of 2) AL321.0

Notes:

The -*C* option was introduced with AIX Version 4 — for earlier systems *noclobber* has no option letter equivalent. The *notify* or -*b* option is also new with AIX Version 4; other systems have no equivalent.

The Korn shell also provides a privileged or -*p* option. However, this is not supported by AIX, as AIX does not allow *SUID* (set user id) shell scripts. The *privileged* option is very similar to the *protected* option that was only available with the 6/3/86 version of the Korn shell (the same option by a different name for that version only).

On systems that do operate *SUID* shell scripts, the *privileged* or -*p* option is on if the effective user or group ids differ from the real ones. It disables the processing of *$HOME/.profile* and *$ENV* — using */etc/suid_profile* instead. Turning the option off resets the effective ids to the real ones.

Korn Shell Options with Set (2 of 2)

Option	*L*	*Description*
`noglob`	`f`	to disable metacharacter expansion
`notify`	`b`	to notify asynchronously of background job completions
`nounset`	`u`	displays an error message when an unset variable is used
	`s`	to sort positional parameters -- ksh only
`trackall`	`h`	set-up a tracked alias for each new command -– on for non-interactive shells
`verbose`	`v`	to display input on standard error as it is read
`vi`		turns on history line recall and **vi** editing
`xtrace`	`x`	the debug option – the shell displays PS4 with each processed command line

Figure 5-23. Korn Shell Options with Set (2 of 2) AL321.0

Notes:

Unit 7 deals with command aliases and the use of the *trackall* or *h* option. The Bourne shell provides command hashing instead of aliases, which is where the *h* originates.

There is a *keyword* or *k* option, that allows "keyword parameters" to be used. These are variable assignments placed in front of a shell Script invocation, that are passed to the script:, e.g. `$ variable=value ... shell_prog argument ...`

The use of "keyword parameters" is **strongly discouraged;** it is provided only for Bourne shell compatibility, and may be withdrawn from future versions of the Korn shell.

One important use for the *set* command is to assign values to shell arrays, using the *+A* and *-A* options. Arrays are covered in Unit 7, so we will leave this for later.

The *interactive* option is listed by a *set* command. However, this option is a shell invocation option, and cannot be altered with the *set +o option* or *set -o option* syntax.

Additional ksh93 Shell Options

`set -o pipefail`

Usually the exit status is of the last command in a pipeline.

`set -o pipefail` changes this behavior.

The exit status of a pipeline is changed to that of the last command to fail

`set -o viraw`

Allows for **`set -o vi`** plus allows **<Tab>** for file name completion

Figure 5-24. Additional ksh93 Shell Options AL321.0

Notes:

Although *set -o pipefail* does not tell you which command failed, it at least tells you something went wrong in your pipeline.

The *viraw* does not work in all versions of ksh93.

Bash Shell Options with Set

- The bash shell options are the same as the Korn shell unless noted:
  - The **set -h** (**set -o hashall**) disables hashing of commands
  - There is no **set -o bgnice** or **set -o trackall**
  - Bash users traditionally use **set -o emacs**

- Use "**set -o**" to list all of bash's options

Figure 5-25. Bash Shell Options with Set AL321.0

Notes:

The Bash shell also provides the “shopt” command to set shell options.

The Bash shell also provides *set -o posix* to change the behavior of the shell to match POSIX 1003.2

Set Quiz

1. What command would you use to re-set the positional parameters to "one" "two" "three"?

2. What lists the shell options with settings?

3. Which **`set`** option ensures that each variable assignment will be inherited by a subshell?

4. What would stop **`<Ctrl-d>`** from logging me out?

5. How can I use **`set`** to protect my files from being overwritten by output redirection?

Figure 5-26. Set Quiz AL321.0

Notes:

Shell Built-in Commands

We have seen the following built-in shell commands:

.	:	bg	break
cd	continue	echo	eval
exec	exit	export	fc
fg	getopts	jobs	kill
print	pwd	read	readonly
set	shift	test	[ ]
trap	typeset	unset	wait

In the later units we will see:

alias	command	let or (( ))	return
times	ulimit	unalias	whence

All built-in commands can run in the current environment

Special built-in commands may terminate the shell if an error occurs

Figure 5-27. Shell Built-in Commands AL321.0

Notes:

Variable assignments made with the underlined special built-in commands remain effective after the commands complete; that would not be the case for regular built-in commands. Command redirections are processed after parameter assignments with special built-in commands only. The "." and ":" special built-in commands won't terminate the current shell when in error; other special builtin commands will. Italicized commands above are not available in the Bourne shell. The *command* command was introduced with Korn shell for AIX Version 4.

Bourne shell has built-in commands for its special features too (these are beyond the scope of this course): *hash*, *login*, *setxvers*, *type*. The *wait* command is a special built-in for the Bourne shell.

The Korn and Bourne shells also provide the following commands (explained in AU14):

`umask`	to set and display default file creation permissions
`newgrp`	to change the effective group id, so that created files are associated with that group.

AIX Shell Commands

Some built-in Korn shell commands are also provided as AIX commands, accessible from all shells:

`alias`	`bg`	`cd`	`command`
`echo`	`fc`	`fg`	`getopt`
`jobs`	`kill`	`newgrp`	`read`
`umask`	`unalias`	`wait`	

AIX commands are also provided for the logical words:

`false` `true`

Most of these commands are shell scripts in /usr/bin – they are provided for POSIX compliance

Figure 5-28. AIX Shell Commands AL321.0

Notes:

By default, the Korn shell will use its own built-in commands instead of AIX built-ins of the same name. To specify the AIX built-ins, you could use a full pathname, for example, */usr/bin/jobs.*

Before AIX Version 4, the following commands were **not** normally implemented by the operating system: *alias*, *bg*, *cd*, *command*, *fc*, *fg*, *getopts*, *jobs*, *read*, *umask*, *unalias* and *wait*. It should however, be an easy matter to write missing mini-shell-scripts for a system.

As we shall see in Unit 7, *true* and *false* are not shell built-in commands as such.

The operating system also provides a *getopt* command (note spelling) that performs a similar function to the Korn shell *getopts* built-in command. Because it is provided by the operating system, it is accessible in all shells.

Checkpoint

1. Without using redirection, how could we print information to file descriptor 2?
2. What is wrong with the following command?
   `read speed?"mph" distance?"miles"`
3. What `getopts` statement would allow you to process options `p`, and `a`, with option `t` expecting an associated value?
4. In the bash shell, `print` is not built-in. What is the built-in command in Bash that performs similarly to Korn's `print`?
5. Which `set` option disables metacharacter pathname expansion?
6. Which `set` options would be most useful in helping to debug a shell script?

Figure 5-29. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

Unit Summary

- The Korn shell `print` command
- The Bash shell `echo` command
- Special printing characters
- The `read` command
- Option and argument processing with `getopts`
- History manipulations with `fc`
- The `set` command
- Shell options with `set`
- Shell invocation
- Built-in commands
- Shell commands provided by AIX

Figure 5-30. Unit Summary AL321.0

Notes:

Unit 6. Arithmetic

What this unit is about

This unit presents the three ways of doing arithmetic operations in shell, expr, let, and bc.

What you should be able to do

After completing this unit, you should be able to calculate using expr, let or (( )) and bc.

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use the `expr` utility
- Understand `expr` arithmetic and logical operators
- Use shell `let` or `((  ))`
- Use number bases
- Use `let` logical operators
- Use integer variables
- Use implicit `let`
- Understand the `bc` utility

Figure 6-1. Unit Objectives AL321.0

Notes:

expr Arithmetic

AIX provides the **expr** utility to perform ***integer*** arithmetic

expr argument1 operator argument2 …

expr features

- Runs as an external executable
- Writes results to standard output
- Exit code is 0 for non-zero evaluations
- Exit code is 1 for zero or null evaluations
- Exit code is $\geq$ 2 if an expression is invalid
- Mostly used for control flow in shell scripts – loop counters

Figure 6-2. expr Arithmetic AL321.0

Notes:

As the *expr* utility is provided by the operating system, it is available from all shells.

In AIX Version 4 error conditions result in an exit code greater than 2, while AIX Version 3 gives 2.

Internally numbers are treated as 32-bit two's complement integers, but are held and output as character strings.

Remember that there are two results; that on standard output and the command exit status. *Expr* also performs pattern matching and string manipulations. We will not be covering these aspects. See the man page if you are interested.

expr Arithmetic Operators

To group expressions use:

( )	fixes evaluation order - otherwise normal rules of precedence apply

The integer operators result in mathematical evaluations:

=	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Figure 6-3. expr Arithmetic Operators AL321.0

Notes:

Expr only does **integer** arithmetic.

You must use a backslash or quotes to protect special characters from the shell, for example, *.

Spaces are required between operators and expressions — except for the unary minus with a literal value, for example, `-3`.

Operators are shown here in order of precedence: highest to lowest.

For the logical operators, if both expressions are integers, numerical evaluation is performed. If character strings are present, ASCII character order is used. Notice the odd standard output values — opposite to the true=0, false=non-zero command exit codes.

expr Examples

Here is some simple integer arithmetic...

```
$ var1=6; var2=3
$ expr  $var1  /  $var2
2
$ expr  $var1  -  $var2
3
$ expr  \(  $var1  +  $var2  \)  \*  5
45
$ _
```

What is the result of the following?

```
$ expr  10  %  3

$ expr  10  /  3
```


Figure 6-4. expr Examples AL321.0

Notes:

Notice that ***everything*** is an argument to *expr*. Make sure you have whitespace around parameters -- except where you need to quote.

The let Command

```
let argument ..

-or-

((  argument  ))
```

- The **`let`** built-in shell command performs long integer arithmetic approximately 10 times faster than **`expr`**
- Evaluates each argument as an arithmetic expression
- No quotes for special characters, or arguments with **`spaces`** or **`tabs`** in them, within **`(( ... ))`**
- Variables need no **`$`**
- The exit code is 0 (true) for non-zero, and 1 (false) for zero evaluations
- In ksh93, **`let`** will use decimal numbers, if you give the arguments in decimal notation
- In bash and ksh88, integer only

Figure 6-5. The let Command AL321.0

Notes:

As multiple arguments are space or tab separated for the ordinary *let* form, you must quote such characters if they appear in an expression.

The *((...))* form of the command may have only one argument.

((...)) is not available in versions of bash prior to 2.0.

let Arithmetic Operators

For simple arithmetic:

`( )`	overrides normal precedence rules
`*`	multiplication
`/`	division
`%`	remainder
`+`	addition
`-`	subtraction (or unary minus)
`=`	assignment

`var op= exp` means `var = var op exp`

Up to nine levels of nested processing will be evaluated:

```
$ z=2  ;  y="z + 1"
$ ((  x=3*y  ))
$ print  $x
9
$ _
```


Figure 6-6. let Arithmetic Operators AL321.0

Notes:

A null variable equates to zero. shell variables do not need the $. When using the ((...)) form, there is no standard output. To keep the result you must save it in a variable.

Operators are listed in order of precedence. The unary minus is evaluated after *( )* and both are evaluated before the other simple operators above. The assignment operator has the lowest precedence of all.

let Arithmetic Examples

Some simple arithmetic...

```
$ a=1 b=2
$ ((  z = 2#10 + -b ))
```

unary minus needs a space before it, not after

```
$ let  c=a+b  d=b\*b
```

no spaces, but \ needed for * multiple arguments

```
$ (( e  =  9 / b ))
```

integer division

```
$ (( e  +=  a ))
```

assignment: addition

```
$ print $z $a $b $c $d $e
```

What do you think we get?

What is the difference between these?

`$(( ... ))` and `(( ... ))`?

Figure 6-7. let Arithmetic Examples AL321.0

Notes:

The *print $((...))* combination will print the answer to standard output. the *((...))* will require you to *echo $z* for you to see the value.

Nesting is possible and (maybe) easier to read and write than *expr*.

```
w=$(( $x + ($y * $z) )) ; print $w
```

let Logical Operator

Logical expressions evaluate to 1 if true, 0 if false
(the exit code is 0 for non-zero, 1 for zero – as expected):

!	logical negation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical "and" = 1 if both LHS and RHS are true (RHS not evaluated if LHS is false)
||	logical "or" = 1 if either LHS or RHS are true (if LHS is true, RHS not used)

Figure 6-8. let Logical Operators AL321.0

Notes:

let 0 (zero) returns 1 (false) — which is equivalent to the Korn shell *false*.

Operators are listed in order of precedence. The logical negation operator has the highest order of precedence after *( )* and the unary minus. Other operators above have a lower order of precedence than the simple arithmetic operators. Notice that these operators have correct logic semantics.

let Logical Examples

```
$ (( p = 9 ))

$ (( p = p * 6 ))
$ print $p
54

$ (( p > 0 && p <= 10 ))
$ print $?
1

$ q=100
$ (( p < q || p == 5 ))
$ print $?
0

$ if (( p < q && p == 54 ))
> then
> print TRUE
> fi
TRUE

$ _
```


Figure 6-9. let Logical Examples AL321.0

Notes:

Follow the flow of the variables.

In the first two examples, the variable is assigned to a value. Numeric expressions are tested in the other examples, using "both true" and "either - or" operators. Finally, an *if* statement precedes the *let* command used for conditional testing.

base#number Syntax

With `let` you are not limited to just decimal (base ten) integers:

- `let` constants are of the form `base#number`
- `base` is an integer in the range 2 to 36 (10 default)
- `number` may include upper or lowercase letters for bases greater than 10

`2#100` in binary	=	4 (in base 10)
`8#33` in octal	=	27
`16#b` in hexadecimal	=	11
`16#2A` in base16	=	42

Figure 6-10. base#number Syntax AL321.0

Notes:

Ways to do your octal or hexadecimal arithmetic perhaps?

Shell integer Variables

Shell variables are stored as character strings unless defined with the **integer** command

```
integer variable=value ...
```

-or-

```
typeset -iN variable=value ...
```

- Sets the **integer** attribute for each variable
- **typeset** can define a base **N**, variables then *print* in the specified base (2 to 36)
- Assignment to an **integer** variable causes expression evaluation – an implicit **let** command
- **let** does not have to convert **integer** variables from character strings to numerical values

Figure 6-11. Shell integer Variables AL321.0

Notes:

We shall see more of the *typeset* command in the next Units. Both *typeset* and *integer* are shell commands.

integer Examples

Some examples of integer and typeset -i ...

`$ integer x`	x can hold only integers
`$ x=string`	
`ksh: string: 0403-009 The specified number is not valid for this command.`	
`$ x=5+10`	implicit let command
`$ print $x`	
`15`	
`$ (( x = 5 + 100 ))`	
`$ print $x`	
`105`	
`$ typeset -i8 nums0 nums1 nums2`	
`$ nums0=8#5`	define an octal integer variable
`$ nums1=8#10`	
`$ (( nums2=8#3*nums0 ))`	assign value
`$ print ${nums2}`	
`8#17`	
`$ x=${nums2}`	
`$ print $x`	print gives answer in base 10
`15`	
`$ _`	

Figure 6-12. integer Examples AL321.0

Notes:

An ordinary *integer* variable assumes the base of its first value assignment — base 10 for *x* in above example.

The above example:

nums0=8#5	equates to nums0=5
nums1=8#10	equates to nums1=8
nums2=8#3*num0	equates to nums2=3 * 5 == 8#17
x=nums2	equates to 15

Implicit let Command

`integer` variable assignments are an implicit `let` command
Other implicit `let` commands are:

- Values for the `shift` command

```
shift OPTIND-1
```

- Resource limits with `ulimit`

```
ulimit -t TMOUT+60
```


Figure 6-13. Implicit let Command AL321.0

Notes:

The *ulimit* command is a shell built-in command: *ulimit -a* displays current settings. Other options are:

`-c N`	core dump size limit (512 byte blocks),
`-f N`	file size limit for all child processes (512 byte blocks),
`-d N`	data area size limit (kilobytes),
`-s N`	stack area size limit (kilobytes),
`-m M`	physical memory limit (kilobytes),
`-t N`	time limit in seconds.

You have already seen the implicit let usage with *OPTIND*. There is one other use in connection with arrays which we cover in the next unit.

bc - Mathematics

The AIX system provides the `bc` utility

```
bc   [file]
```

- Performs floating point arithmetic
- Acts as a filter command or interactively
- Reads arithmetic expression strings from standard input or from a specified file
- Semicolons or new lines separate expressions
- Sets the `scale` variable inside `bc` to define the required number of decimal places
- Prints results to standard output

Figure 6-14. bc - Mathematics AL321.0

Notes:

The *bc* command works in decimal, octal or hexadecimal. Set the variables *ibase* and *obase* to specify the input and output number bases respectively.

Caution: Base conversion will not work for hexadecimal to decimal, or octal to either of the other bases.

Another caution: *bc* is not good for financial figures.

bc Operators

For simple arithmetic and logical evaluations, use:

`(, ), +, -, *, /, %, =`	as for `let` arithmetic operators
`==, !=, <, <=, >, >=`	as for `let` logical operators
`x^y`	raise `x` to the power `y`
`sqrt(x)`	square root
`x++ ++x`	post and pre increment `x`
`x-- --x`	post and pre decrement `x`
`x op= y ≡ x = x op y`	for +=, -=, *=, /=, %=, ^=

A library provides complex mathematical functions:

`s(x)`	sine of `x`
`c(x)`	cosine of `x`
`e(x)`	natural exponential of `x`
`l(x)`	natural log of `x`
`a(x)`	arctangent of `x`
`j(n,x)`	Bessel function

Precision functions:

`length(n)`	number of significant digits for example, 123.456 has n=6
`scale(n)`	number of digits after decimal point for example, 123.456 has n=3

Figure 6-15. bc Operators AL321.0

Notes:

To use the complex mathematical functions, you may need to specify the *-l* option to *bc* on the command line.

It is also possible to define complex functions of your own, in a C-language like syntax.

Logical flow control is also provided in *bc* — again in C-language structures.

Comments may be included in complicated files using the /* *comment* */ C notation.

Again **Caution:**

> The multiply routine may yield incorrect results if a number has more than LONG_MAX / 90 total digits. For 32 bit longs, this number is 23,860,929 digits.

Checkpoint

1. Multiply together variables **`a`** and **`b`**, using **`expr`**.
2. Use **`expr`** to multiply variable **`a`** by the sum of **`b`** and **`c`**.
3. Set variable **`hex`** to contain the hexadecimal value **7c**.
4. Write a **`let`** statement to test whether variable **`a`** is smaller than variable **`b`**.
5. Define a variable **`num`** as numeric only.
6. Increment a numeric variable **`numvar`**, by three.
7. How would you calculate 6/7 to 6 decimal places?
8. How would you calculate the square root of 8541976320?

Figure 6-16. Checkpoint AL321.0

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.
7.
8.

Unit Summary

- The `expr` utility
- `expr` arithmetic and logical operators
- Shell `let` or `(( ))`
- Number bases
- `let` logical operators
- Integer variables
- Implicit `let`
- The `bc` utility

Figure 6-17. Unit Summary AL321.0

Notes:

Unit 7. Shell Types, Commands, and Functions

What this unit is about

This unit describes shell arrays, command substitutions, functions and variables, and aliases.

What you should be able to do

After completing this unit, you should be able to:

- Use array variables
- Use command substitution
- Define and call functions
- Use typeset variables
- Process aliases
- Understand shell command line processing

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use shell arrays
- Define and call functions
- Use **`typeset`** command
- Use **`autoload`** functions
- Process command aliases
- Use preset aliases
- Use tracked aliases
- Use the **`whence`** command
- Understand command line processing
- Understand command line re-evaluation with **`eval`**

Figure 7-1. Unit Objectives AL321.0

Notes:

Defining Arrays

The Korn and Bash shells supports one-dimensional arrays:

- Arrays need not be "declared"
- Access an element of an array by a subscript to a variable name
- Any variable with a valid subscript becomes an array
- A subscript is an expression enclosed within `[ ]`
- Subscripts should lie in the range 0 to 4095 -- (ksh only)
- Variable attributes (for example, **`readonly`**) apply to all elements of the array

Caution: An entire array cannot be exported, only the 0th element

Figure 7-2. Defining Arrays AL321.0

Notes:

All variables are arrays in Korn and Bash shell but because the default is element zero then

```
VAR1 == VAR1[ 0 ]
```

Assigning Array Elements

Just like ordinary variables, values can be assigned, and later referred to:

- Assign contents to an array element using
  `array[N]=argument`
- To **unset** an array and assign new values sequentially, use
  `set -A array argument ...`
- To simply replace existing array values with new ones, use
  `set +A array argument ...`

Figure 7-3. Assigning Array Elements AL321.0

Notes:

Korn shell variable names and contents are not limited in length; this applies to array elements also.

You can unset an array by: *unset array* — specifying the array name is enough.

The set -A syntax does not work in the Bash shell.

In ksh93 and bash, arrays can also be set up in the following manner:

array=(arg1 arg2 arg3)

In addition, the Bash shell supports the following syntax:

array=([2]=value [0]=diffvalue [3]=anothervalue)

The read -a (bash) or read -A (ksh93) allows the read command to set up arrays. (Refer to Unit 5)

Associative Arrays in ksh93

- ksh93 allows associative arrays
- Associative arrays are indexed by string values
- Indicate an associate array with **`typeset -A`**
  - Examples:

```
$ typeset -A tax
$ tax[NJ]=6
$ tax[NM]=5
$ tax[NY]=4
```


Figure 7-4. Associative Arrays in ksh93 AL321.0

Notes:

Associative arrays are allowed in ksh93.

Referencing Array Elements

The **$** notation is used to refer to the value in a variable:

- When referencing an array element use **${ }** notation
  `print ${array[N]}`
- To refer to all the elements of an array use an ***** or **@** subscript (to give a space separated list)
  `${array[*]}` or `${array[@]}`
- If you omit a subscript, it means the zeroth element
  `${array[0]}` == `$array`
- To show how many elements exist within an array
  `${#array[@]}`

Figure 7-5. Referencing Array Elements AL321.0

Notes:

Just as for positional parameters, where:

```
        "$@" = "$1" "$2" ...
and     "$*" = "$1 $2 ..."
```

with array elements:

```
        "${array[@]}" = "${array[0]}" "${array[1]}" ...
and     "${array[*]}" = "${array[0]} ${array[1]} ..."
```

Array Examples

`$ list[0]="Line 0"`	fill the array list.
`$ list[1]="Line 1"`	
`$ list[3]="Line 3"`	
`$ print $list`	print the zeroth element.
`Line 0`	
`$ print ${list[*]}`	print all elements.
`Line 0 Line 1 Line 3`	
`$ print ${list[0]}`	print elements individually.
`Line 0`	
`$ print ${list[1]}`	
`Line 1`	
`$ print ${list[2]}`	element `[2]` is null.
`$ print ${list[3]}`	
`Line 3`	
`$ print $list[1]`	without `{ }` notation, we
`Line 0[1]`	get "`$list`" + "`[1]`".
`$ _`	

Figure 7-6. Array Examples AL321.0

Notes:

Another Array Example

Here we have the beginnings of a card game.

```
#!/usr/bin/ksh
# Usage: pickacard.ksh
# To choose a random card from a new deck
integer number=0
for suit in CLUBS DIAMONDS HEARTS SPADES
  do
  for n in ACE 2 3 4 5 6 7 8 9 10 JACK QUEEN KING
    do
    card[number]="$n of $suit"
    number=number+1
    done
  done
print  ${card[RANDOM%52]}

$ pickacard.ksh
QUEEN of DIAMONDS
$ _
```


Figure 7-7. Another Array Example AL321.0

Notes:

The lines picked out in italic or bold italic have implicit *lets* which were covered in the Unit 6. With an implicit *let* you don't need the dollar to reference shell variables.

Defining Functions

- Commands can be group together and named.
- The set of commands form the **function** body.
- **function** definitions look like:

Bourne, Korn, and Bash	Korn and Bash
`identifier()`	`function identifier`
`{`	`{`
`    commands`	`    commands`
`}`	`}`

- Functions:
  – Provide a means of breaking down programs into discrete units
  – Stored in memory for fast access
  – Executed, like new commands, in the current environment

Figure 7-8. Defining Functions AL321.0

Notes:

Functions are helpful in scripts for several reasons. It allows you to reuse code, allows the script to be readable, and is stored in memory for faster access.

A function must be defined before it is used, that is, put the definitions at the top of a Shell script.

In the Korn shell, functions may have the same name as that of a script variable: in the Bourne shell, this is not possible.

Don't use reserved words in a function name: *!*, *{*, *}*, *case*, *do*, *done*, *elif*, *else*, *esac*, *fi*, *for*, *function*, *if*, *in*, *select*, *then*, *time*, *until*, *while*, *[[*, *]]*. You cannot create a function with the same name as a special shell builtin command. If you give a function the same name as a regular builtin command, and use that command within the function definition, recursion occurs.

The Korn shell *command* command (introduced with AIX Version 4) suppresses function lookup — this allows you to avoid recursion within a function.

Functions and Variables

Functions have different variables to the main script:

- Arguments
  - Taken as positional parameters to the function
  - Calling script **`$1-${n}`** parameters are reset on leaving the called function
- Variables
  - Declared with the **`typeset`** or **`integer`** commands (inside a Korn shell function) are "local" variables to the function
  - All other variables are "global" in the Script
  - The "scope" of a "local" variable includes all functions called from the current function

Figure 7-9. Functions and Variables AL321.0

Notes:

Inside a function *$** and *$@* refer to the arguments to the function.

Local variables do not exist in the Bourne shell. More on the *typeset* command later in this unit.

Normally all variables in a shell script are global, that is, accessible anywhere in the script.

In ksh88, $0 will be the function name while inside the function and $0 will reflect the scriptname when it leaves the function, **IF** the function was set up with the “function identifier” syntax.

If set up with “identifier()” syntax, $0 will reflect the scriptname while both inside and outside the function.

function Example

A useful function...

Handy for usage errors in Shell Scripts
Invoke function usage with arguments: script
followed by arglist. Note exit status!

```
function usage
{
  prog="$1"; shift
  print -u2 "$prog: usage: $prog $@"
  exit  1
}
```


Figure 7-10. function Example AL321.0

Notes:

Ending Functions

A function completes after executing the last command:

- The exit code is normally that of the last command
- **return** can be used to specify an exit code **N**, or just end the function at that point

  return N
- **exit** will terminate the current function and current shell

  exit N
- Errors within a Korn shell function cause it to return control and the error exit code to the calling Script

Functions may be deleted from memory using...

unset -f functionname

Figure 7-11. Ending Functions AL321.0

Notes:

In the Bourne shell function errors abort the script, like an *exit* command.

Functions and Traps

The behavior of **`trap`** with functions is determined by the shell type:

Bourne:	a **`trap`** is "global" – the same in and out of a **`function`**
Korn:88	a **`trap`** is "local" to a **`function`** and is reset on completion
	a main program **`trap`** is shared with **`function`**s, but can be overriden inside **`function`**
	a **`signal`** that is not caught or ignored, may cause the script to terminate
	a **`signal`** that is ignored by the shell, is also ignored by **`function`**s called from it

Figure 7-12. Functions and Traps AL321.0

Notes:

Before AIX Version 4, only main program *ERR* and *EXIT* traps were not shared with functions. Where a signal was neither caught nor ignored, the condition would be passed back to the calling program.

A signal that is ignored by the main shell cannot be trapped by any subshell; it is always ignored.

Functions in ksh93

- Function's characteristics change in ksh93 depending on which syntax was used to set up the **function**

```
identifier ( )
{ ...
}
```

- All variables are global

- **$0** always scriptname

- A main program trap is shared with **functions**

- A **trap** inside a **function** overrides a main program **trap**, and is passed out

```
function identifier
{ ...
}
```

- Variables are made local with **"typeset"**

- **$0** reflects **function** name while inside the **function**

- A main program **trap** is shared with **functions**

- A **trap** inside a **function** overrides a main program **trap**, but only while inside the **function**

Figure 7-13. Functions in ksh93 AL321.0

Notes:

The identifier() form is for compatibility with the Bourne shell and for POSIX compliance. The functions identifier form is a more powerful Korn shell form.

Functions in bash

- `$0` will always be the scriptname, whether inside or outside `function`
- Prefers "`declare`" or "`local`" over `typeset`
- A main program `trap` is shared with `function`
- A `trap` within a `function` overrides the main program `trap` while inside the `function`, and is passed out to the main program

Figure 7-14. Functions in bash AL321.0

Notes:

Typeset is available for compatibility.

The typeset/declare Commands

The Korn shell **typeset** and Bash shell **declare** commands define or list variables and their attributes:

```
typeset ±LN variable1=value1 variable2=value2 ...
```

Omitting variables lists variables with specified attributes

- sets attributes, or lists names and values

+ unsets attributes, or lists just names

Where **L** is any of ...

r	the **readonly** attribute – no modification of variables' value
i	sets the **integer** attribute – use with **N** to set number *base*
x	the **export** attribute – the variable will be exported

The preferred method in bash is the "**declare**" command

Figure 7-15. The typeset/declare Commands AL321.0

Notes:

Attributes are set, or unset, after assigning optional values to specified variables.

"`-H`" sets the pathname mapping attribute; on non-UNIX systems pathnames are converted into host system names.

We saw the "*-i*" option used in the last unit.

bash provides the *typeset* command for compatibility but preferred usage is declare.

typeset Examples

Declare arrays to specify size and/or attributes:

```
$ typeset -xi8 a2[1]             exported & octal integer
$ a2=52
$ a2[1]=25
$ ksh
$ print $a2 ${a2[1]}
8#64                             only element 0 was exported
$ _
```

Inside a Korn Shell function, **typeset** creates a "local" variable...

```
# Function to convert numbers into binary
function  binary_convert
{
      typeset -i2 binary=$1
      print "$1 = $binary"
}
```


Figure 7-16. typeset Examples AL321.0

Notes:

If you create a “local” variable with the same name as a “global” one, the two variables are distinct.

To list variables with the readonly attribute...

```
$ typeset +r
LOGNAME
$
```

typeset with Functions

Other uses of **typeset** are:

- Display functions
- Set function attributes
- Unset function attributes

```
typeset ±fL function1 function2 ...
```

- To list functions with specified attributes, omit function list
- **-f** sets attributes, or displays function names and definitions
- **+f** unsets attributes, or displays only function names

Where **L** is any of...

x the **export** attribute – the function will be available to implicit shells invoked from the current one

t the shell **xtrace** option for a function

Figure 7-17. typeset with Functions AL321.0

Notes:

You must be using your history file for the listing options to work: the shell *nolog* option must be off when function definitions are read.

Functions that are to be defined across explicit invocations of a shell should be defined in the *$ENV* file, with the *export* attribute so that they are available to subsequent shells (implicit or explicit).

The *-t* option is not available in bash.

The *-x* option does not work in ksh93.

typeset with Functions Examples

```
$ typeset -f                        shows functions in full
function list
{
      while [[ "$1" != "X" ]]
      do
                  print $1
                  shift  1
      done
}
$ typeset -fx list                  export the list function
$ typeset +f  or typeset -F         (bash) lists function names
list
$ unset -f list
$ typeset -f                        list doesn't exist anymore
$ _
```


Figure 7-18. typeset with Functions Examples AL321.0

Notes:

In the next unit we will see even more uses for *typeset*.

autoload Functions

A shell function that is defined only when it is first called, is an **autoload** function:

- Using **autoload** functions improves performance
- The shell searches directories listed in the **FPATH** variable for a file with the name of the called function
- Call the **autoload** from within your .profile (or .bash_profile)
- The contents of that file then defines the function
- Existing function definitions are not unset

Another way is to:

- Place functions into a separate directory
- Set **$FPATH** equal to the full pathname of that directory
- Make sure the function name and file name is the same

Figure 7-19. autoload Functions AL321.0

Notes:

By putting several related function definitions in a file, and using the operating system *ln* command to create multiple names for the file, you can *autoload* libraries of functions. The multiple names are those of the functions in the file of function definitions.

There is another way. Follow these steps:

Place functions into a separate directory.

Set $FPATH equal to the full pathname of that directory.

Make sure the function name and file name is the same.

Autoload is not necessary now.

Aliases

The Korn shell **alias** facility provides:

- A way of creating new commands
- A means of renaming existing commands

Creation:	**alias name=definition**
Deletion:	**unalias name**

An **alias** definition may contain any valid shell script or metacharacters

Figure 7-20. Aliases AL321.0

Notes:

Like *functions*, aliases must be defined before they are used, so put definitions at the top of shell Scripts.

You may redefine shell built-in commands using aliases, but don't use aliases for reserved words.

Reserved words are: *!* *{*, *}*, *case*, *do*, *done*, *elif*, *else*, *esac*, *fi*, *for*, *function*, *if*, *in*, *select*, *then*, *time*, *until*, *while*, *[[*, *]]*.

Since AIX Version 4, all aliases can be removed with a single command: *unalias -a*.

Processing Aliases

Command lines are split into words by the shell:

- Check the first word of each command line for a defined **`alias`**
- A backslash in front of a command name prevents **`alias`** expansion if the **`alias`** exists
- If the definition ends in a **`space`** or **`tab`**, the next command word will also be processed for **`alias`** expansion
- Resolve **`alias`** names within a **`function`** when function definitions are read, not at execution

Figure 7-21. Processing Aliases AL321.0

Notes:

Definitions must be quoted to include spaces or tabs.

Preset Aliases

- Korn shell uses the following exported aliases
  – May be unaliased or redefined

```
alias functions='typeset -f'
alias hash='alias -t'
alias history='fc -l'
alias integer='typeset -i'
alias nohup='nohup '            with trailing space
alias r='fc -e -'
alias stop='kill -STOP'
alias suspend='kill -STOP $$'
alias type='whence -v'
```


Figure 7-22. Preset Aliases AL321.0

Notes:

It is not a good practice to alter the above aliases; it will confuse other programmers if nothing else.

In ksh93, these are the preset aliases:

command=’command ’
fc=hist
float=’typeset -E’
nameref=’typeset -n ’
redirect=’command exec ’
times=’{ { times; } 2>&1; }’

We shall see what *hash* and *whence* do in a moment.

The alias Command

The **alias** command has some options:

alias -L name=definition

Where **L** is any mix of...

x	to set, or display exported aliases
t	to set, or list tracked aliases

If ***definition*** is quoted...

"definition"	doubles are interpreted when entered
'definition'	singles are interpreted when executed

Figure 7-23. The alias Command AL321.0

Notes:

A backslash can be used inside a "*definition*" to prevent recursion for a command. Single quotes around the whole *definition* have the same effect.

Tracked aliases are covered in a moment.

An exported alias is passed to shells invoked from the current one. However, to export an alias across different explicit shells, you must define and export it from the *$ENV* file. Explicit means wherever you can see "*ksh*" in the invocation — for example, `ksh, ksh -c "commands", ksh prog`. Also, running a script that has the special "*#!/usr/bin/ksh*" comment as its first line will invoke a new explicit shell.

Notice what happens when you use single or double quotes. In most cases you will want single quotes so that any interpretation occurs when the alias expands later.

There are no *-x* or *-t* options in bash.

The *-x* option is only available in ksh88, and then only to implicit Korn shells.

alias Examples

`$ x=10`	
`$ alias px="print $x" rx='print $x'`	
`$ x=100`	
`$ px`	prints `$x` as it was
`10`	
`$ rx`	prints the current `$x`
`100`	
`$ alias -x ls='ls -a'`	`ls` is set and exported
`$ rm /tmp/*`	you want to remove all /tmp
`rm: remove '/tmp/atestfile'? _`	
`<Ctrl-c>`	you realize the list is too long
`$ \rm /tmp/*`	you *escape* the alias for rm
`...`	you cross your fingers
`$ ls /tmp`	you hope you did it correctly

Figure 7-24. alias Examples AL321.0

Notes:

Tracked Aliases

A **`tracked alias`** reduces the search time for a future use of a command

`set -o trackall` **or** **`set -h`**

Turns on Shell **trackall** option

First use of a command creates **tracked alias**

Force creation with

`alias -t name`

List all **tracked alias**es

`alias -t`

NOTE: The value of a **tracked alias** becomes undefined when the **`PATH`** variable is reset

Figure 7-25. Tracked Aliases AL321.0

Notes:

Once created, a tracked alias will obscure a new command of the same name if it is placed in the command search *PATH*, in a directory that is before that of the original command.

Some tracked aliases are predefined for the Korn shell. What these are varies from system to system.

The Bourne shell provides command hashing instead of tracked aliases, which is where the *-h* originates.

Hashing in bash

A **`hash`** reduces the search time for a future use of a command.

All commands are remembered in a **`hash`** table by bash. Disable this facility by:

`set -d` or **`set -o nohash`**

The built-in **`hash`** lists the table

Add an explicit entry by

`hash command` (must be in PATH)

To delete the **`hash`** table:

`hash -r`

Figure 7-26. Hashing in bash AL321.0

Notes:

Once created, a hash will obscure a new command of the same name if it is placed in the command search *PATH,* in a directory that is before that of the original command. However, only commands that are searched for in PATH are remembered.

The whence Command

whence reports how a command will be carried out by the shell

```
whence -pv command
```

- **-v** for a verbose report
- **-p** to force a PATH search even if the command is an **alias** or **function** (AIX only option)

```
$ whence vi
/usr/bin/vi
$ whence -v vi                          executable program
vi is a tracked alias for /usr/bin/vi
$ whence -v print
print is a shell builtin
$ whence type                           so type is an alias
whence -v
$ type for
for is a reserved word
$ _
```

* when in bash, use **type** instead of **whence** (**type** is built-in in bash)

Figure 7-27. The whence Command AL321.0

Notes:

The whence command reports: aliases, exported aliases, keywords (shell reserved words), built ins, functions, undefined functions (autoload functions), tracked aliases and programs.

Since AIX Version 4, the *command* command is provided as both a Korn shell built in and as an AIX command accessible from all shells. *command -v* and *command -V* perform similar functions to *whence* and *whence -v*. When used as an AIX command, *command* operates in a subshell, and thus will not report functions or aliases unless they were defined and exported by the *$ENV* file. *command -p* is similar to *whence -p*, but the former uses a default *PATH* for its search, and thus will only find the standard AIX commands.

When in bash use *type* instead of *whence*. *type [-a|-p -t]*.

-a print all places name is found
-p returns pathname if name is a file only
-t output actual type only

The eval Command

The shell processes each command line read before invoking the relevant commands.

- If you want to reread and process a command line, use **`eval`**:
- **`eval`** processes its arguments as normal
- The arguments are formed into a space separated string
- The shell then executes that string as a command line
- The return value is that of the executed command line

Figure 7-28. The eval Command AL321.0

Notes:

The *eval* command works in the Bourne, Bash, and Korn shells in the same way.

eval is a very powerful feature. It has been known for programmers to emulate their favorite command interpreters with a script based on using argument processing and *eval*.

eval Examples

Here are some eval command lines...

```
$ eval print '*sh'
getopts.example.ksh      eval.ksh try.sh

$ message10=Hello

$ variable=message10
$ eval print '$'$variable
Hello

$ cmd='ps -ef | grep marty'
$ eval $cmd
...
$ _
```

- print the message named by $variable
- run a string command to list marty's processes

Figure 7-29. eval Examples AL321.0

Notes:

From a shell script, you can use *eval* with the positional parameters.

```
#!/usr/bin/ksh
# Usage: put [options] filename
# Test that the last argument is a filename.
if eval  [[ ! -f \${$#}  ]]
then
    print -u2  "File not found:"
    exit  1

fi
```


Command Line Processing

Each command line is processed in the following way by the shell:

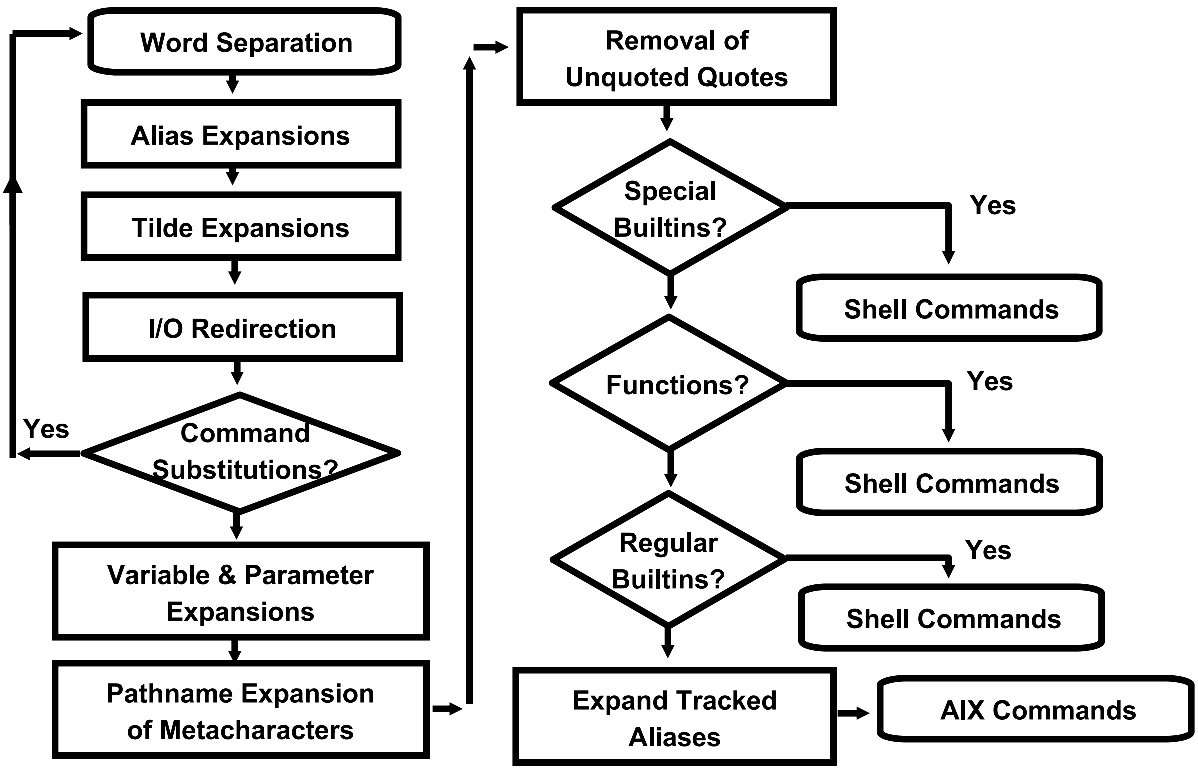


Figure 7-30. Command Line Processing AL321.0

Notes:

In the next unit we will see what tilde expansion does. Other shells process lines in a different way.

Before AIX Version 4, shell regular built-in commands were handled along with special built-in commands. Special built-in commands are: “*.*”, “*:*”, *break*, *continue*, *eval*, *exec*, *exit*, *export*, *newgrp*, *readonly*, *return*, *shift*, *times*, *trap* and *typeset*.

Checkpoint

1. How is an array defined?
2. How do we refer to array elements?
3. How could we set a variable **users**, to contain the number of users logged onto the system?
4. How would we write a function to check the readability of a file?
5. How do we print out the first and last positional parameter?
6. How do we define local variables within a function?
7. How can we list which functions are defined?
8. Which command would allow you to load a library of functions?
9. How could we create an alias to show how many minutes have elapsed since the current shell began?

Figure 7-31. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

8.

9.

Unit Summary

- Shell arrays – defining and referencing
- Functions
- `typeset` command
- `autoload` functions
- Command aliases
- Preset aliases
- Tracked aliases
- The `whence` command
- Command line processing
- Command line re-evaluation with `eval`

Figure 7-32. Unit Summary AL321.0

Notes:

Unit 8. More on Shell Variables

What this unit is about

This unit describes more uses for variables; replacement, changing substrings, length operator, and typeset options.

What you should be able to do

After completing this unit, you should be able to:

- Evaluate substrings
- Provide default or alternate values for variables
- Format strings using typeset options

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use variable replacements
- Evaluate variable substrings
- Evaluate variable lengths
- Understand further `typeset` options
- Use compound variables
- Use indirect variables
- Use tilde expansions

Figure 8-1. Unit Objectives AL321.0

Notes:

Variable Replacements

Value of variables can be replaced with alternate values

`${variable:-WORD}`	value is WORD if variable is unset (use as a temporary value)
`${variable:=WORD}`	value is WORD if variable is unset and assigns word to variable if it is unset (use as a permanent value)
`${variable:+WORD}`	value is null if variable is unset, else value is WORD (use as alternate value)
`${variable:?WORD}`	if variable is unset, WORD is displayed on standard error and the shell script or function terminates with a non-zero exit code (exit 1)

Figure 8-2. Variable Replacements AL321.0

Notes:

These *${ }* forms work in Bourne, Bash, and Korn shells. There are no spaces between curly braces, variable, special characters or word.

The `:-` provides a temporary replacement, where `:=` is permanent.

If you omit *word* from the `${variable:?word}` form the Korn shell displays the message *ksh: variable: 0403-040 Parameter null or not set.* by default, otherwise *ksh: variable: word* results.

The behavior of the `${variable:?word}` syntax in functions varies across AIX versions. In Version 3, a function terminates and returns control to the calling program. Since version 4, the shell Script terminates completely.

The Korn shell allows extended parameter lists, which enable the generated line to exceed the traditional Bourne shell line length limit of 5120 characters. *Variable* can be a number — a positional parameter.

The use of the ":" allows you to decide whether a NULL variable is itself valid or not. A NULL variable has the value of the null string (usually written "" or '').

Variable Replacement Examples

Some simple examples...

- Print date and time using command substitution, or use what was set earlier (do not allow null date):

```
print ${date:-$(date)}
```

- To assign the value of TERM_DEF to TERM if it is unset or null:

```
TERM_DEF=ibm3162
...
print "TERM set as ${TERM:=$TERM_DEF}"
```

- Using the alternate value "1" if variable has a value:

```
var_flag=${var:+1}
```

- To exit the script if var1 is unset or null

```
${flag:?"flag is unset"}
```


Figure 8-3. Variable Replacement Examples AL321.0

Notes:

Remember that the use of a : (colon) means the value of variable may be null. So the second example only allows a string with characters in the variable `date` (but maybe not a valid date string!). In the extra example below, you allow positional parameter 3 to have a null string value.

Extra example:

To exit the script if positional parameter was not given (it can not be null)

```
${3?'No third paramter!'}
```


Shell Substrings

In the shell the `${ }` syntax also works with patterns:

`${variable#pattern}`	removes smallest matching left pattern from variable
`${variable##pattern}`	removes the largest matching left pattern
`${variable%pattern}`	removes the smallest right matching pattern
`${variable%%pattern}`	removes the largest matching right pattern

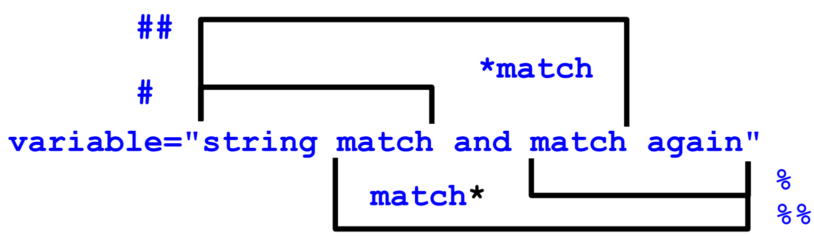


Figure 8-4. Shell Substrings AL321.0

Notes:

Patterns can be composed using shell metacharacters.

Shell Substring Examples

A bit of chopping...

```
$ variable="Now is the time"
$ print ${variable#N*i}            shortest left
s the time
$ print ${variable##N*i}           longest left
me
$ print ${variable%time}           shortest right
Now is the
$ print ${variable%%t*e}           longest right
Now is
$ _
```

Here's a function to strip out the file name from its path and print it...

```
function base
{
        print  ${1##*/}            # match what?
}
```


Figure 8-5. Shell Substring Examples AL321.0

Notes:

The function *base* says take the first parameter to the function and then applies a leftmost match from the start of the string value. The */ pattern matches up to the last / in the string or none. The result is to remove any such match leaving the last component of the pathname.

For those that are curious and have come across old scripts, the utility *expr* that was seen earlier can do similar work but it is slower and has a trickier syntax.

Shell Substring Quiz

Now it's your turn...

1. How can I strip the ".c" extension from a C program file name held in variable "name", and print it?

2. Write a function "path" to print the pathname part of a file name.
   -- /usr/local/bin/program

Figure 8-6. Shell Substring Quiz AL321.0

Notes:

Variable Lengths

A special variant of the `${}` syntax can be used to find the length of a variable:

- To find the number of characters in a variable...

  `${#variable}`

- The number of positional parameters is...

  `${#*}` or `${#@}`

- For the number of elements set in an array (not the highest element subscript)...

  `${#array[*]}` or `${#array[@]}`

Figure 8-7. Variable Lengths AL321.0

Notes:

You can regard the # character here as a (sort of) *length operator* when it appears inside a variable reference.

typeset Options Review

typeset command is used to:

– Set attributes for variables or functions
– Create local variables in functions

`typeset ±LN variable=value...`

where **L** is...	**i**	**integer**, **N** is a fixed base	
		r	to set **readonly**
		x	to **export** the variable

`typeset ±fL function...`

where **L** is...	**x**	to **export** the function	
		u	for an **autoload** function
		t	to set **xtrace** in the function

- To set attributes, display names and values
+ To unset attributes or display just names

Figure 8-8. typeset Options Review AL321.0

Notes:

In the last unit we saw the *typeset* command used to set attributes of variables and functions and create local variables in function definitions. There are several more options that allow variables to be formatted upon expansion by the Korn shell. The *typeset* command is a Korn shell built-in.

Bash users should use *declare* instead of *typeset*, which is obsolete.

Further typeset Options

Options below allow variables to be formatted upon expansion by the Korn shell:

```
typeset ±LN variable=value...
```

where **L** is...

- **u** convert ***value*** to uppercase when expanded
- **l** convert ***value*** to lowercase
- **L** left-justify, pad with trailing blanks to width **N** – if value is too big, truncate from the right
- **R** right-justify, adding leading blanks to width **N**– if wider than **N**, truncate from the left
- **LZ** left-justify to width **N** and strip leading zeros
- **RZ** right-justify to width **N**, adding lead zeros if the first character is a digit

*The bash shell does not support these options

Figure 8-9. Further typeset Options AL321.0

Notes:

For some systems there are multibyte versions of the Korn shell (using national language support). There the width refers to the number of columns rather than the number of characters. The default width is the width of first assignment.

Option *Z* is identical to *RZ*.

typeset Examples

Here are the different types in action...

```
$ typeset -u var=upper
$ print $var
UPPER
$ typeset -l var=LOWER                    # lower case "ell"
$ print $var
lower
$ typeset -L6 text=SIDE
$ print "${text}="
SIDE  =
$ typeset -R6 text
$ print "=$text"
=  SIDE
$ typeset -LZ4 num=000.1234567
$ print ${num}
.123
$ typeset -RZ5 num=123
$ print $num
00123
```


Figure 8-10. typeset Examples AL321.0

Notes:

Extra examples:

```
$ typeset -L6 text=SIDEWAYS
$ print "${text}="
SIDEWA=
$ typeset -R6 text=SIDEWAYS
$ print "=$text"
=DEWAYS
```

Compound Variables in ksh93

ksh93 has an additional feature called *compound variables*, for example:

```
$ time="10:47:24 EST"
$ time.hour=10; time.minute=47
$ time.seconds=24; time.zone=EST
$ print $time
10:47:24 EST
$ print ${time.hour}
10
```

- or -

```
$ time=(hour=10 minute=47 seconds=24 zone=EST)
$ print $time
(hour=10 minute=47 seconds=24 zone=EST)
$ print ${time.zone}
EST
```


Figure 8-11. Compound Variables in ksh93 AL321.0

Notes:

With compound variables there is a requirement that the parent variable exist (in our example $time) before individual elements can be set.

The { } are mandatory when printing out an element.

You can also reference variables indirectly using the *nameref*:

```
$ var1="Terry Terrell"
$ nameref doctor=var1
$ print  $doctor
Terry Terrell
$ print $var1
Terry Terrell
```


Variable Pattern Substitution in bash and ksh93

The bash and ksh93 shells allow for ***on the fly*** variable pattern substitution.

Syntax:

`${variable/pattern/newpattern}`

If **`variable`** contains **`pattern`** the ***first*** match the of **`pattern`** is replaced with **`newpattern`**

`${variable//pattern/newpattern}`

Same as above syntax, except ***every*** match of **`pattern`** is replaced

Also:

`${variable:offset:length}`

Show the substring beginning at **`offset`** for **`length`** number of characters

Figure 8-12. Variable Pattern Substitution in bash and ksh93 AL321.0

Notes:

Example 1:

```
$ print ${PATH/\//\\}
   \usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:$HOME/bin
```

Example 2:

```
$ print ${PATH//\//\\}
   \usr\kerberos\bin:\usr\local\bin:\bin:\usr\bin:$HOME\bin
```

Example 3:

```
$ print ${PATH:47:13}
/home/pat/bin
```

Tilde Expansions

Following **alias** expansion the Korn shell checks for a leading unquoted **~** character to see if it is:

~	tilde by itself is replaced by **$HOME**
~user_name	is expanded into the **$HOME** value for the ***user_name*** given
~other_text	will be left alone

Examples...

cd ~	**=**	**cd $HOME**
lastdir=~-	**=**	**lastdir=$OLDPWD**
johns=~john	**=**	**johns=/home/john**

Figure 8-13. Tilde Expansions AL321.0

Notes:

The use of tilde is not often seen now, though you may see ~*userid*.

Checkpoint

1. What happens when the variable **TMOUT** is set and you enter the following? **TMOUT=${TMOUT:-60}**
2. What would your prompt say if you were in your bin directory and you entered this: **PS1='${PWD#$HOME/} $'**.
3. How could you find out the number of characters in the variable **HOME**?

Figure 8-14. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

Unit Summary

- Variable replacements
  – For unassigned/null strings
- Variable substrings
  – Simple pattern matches
- Variable lengths
  – The # operator
- Further typeset options
  – Justification and padding
- Tilde expansions
  – Shortcuts
- Compound variables
  – ksh93
- Indirect variables
  – ksh93

Figure 8-15. Unit Summary AL321.0

Notes:

Unit 9. Regular Expressions and Text Selection Utilities

What this unit is about

This unit describes regular expressions, and some UNIX text selection utilities.

What you should be able to do

After completing this unit, you should be able to:

- Understand and use regular expressions
- Use grep, cut, and other text selection and manipulation tools

How you will check your progress

Accountability:

- Checkpoint questions
- Hands on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use `regular expressions`
- Use the `grep` command
- Use the `tr` command
- Use the `cut` command
- Use the `paste` command

Figure 9-1. Unit Objectives AL321.0

Notes:

Sample Data File

To manipulate data, we need to know its format.

The data file we will use in this unit has the following structure:

```
Lastname,<Space>Firstname<Tab>nnn-mmmm

$ cat  phone.list
Terrell, Terry              617-7989
Franklin, Francis           704-3876
Patterson, Pat              614-6122
Robinson, Robin             411-3745
Christopher, Chris          305-5981
Martin, Marty               814-5587
Llewellyn, Lynn             316-6221
Jansen, Jan                 903-3333
Llewellyn, Lee              817-8823
$ _
```


Figure 9-2. Sample Data File AL321.0

Notes:

The *phone.list* file will be used in examples on following pages. There is a single space character after the comma following the *Lastname*.

Can you tell what separates the firstname from the phone number? Is it a tab? Is it spaces? Use “cat -vet phone.list” to find out.

Regular Expressions

Powerful feature available in many programs

Used to **select** text in:

– vi, ex, emacs, grep/egrep, sed, awk, perl

What are **`RegExes`**?

– An expression representing a pattern of characters

– Contain a sequence of characters/metacharacters

Figure 9-3. Regular Expressions AL321.0

Notes:

You will find the regular expression feature is part of many programs such as in editors and in pattern matching utilities (we see later in this unit). The principles and uses of regular expressions (often abbreviated to RE) appear in many places in AIX and UNIX systems. Once you have grasped the essential techniques you will find that they can be used over and over again.

An RE is just that — an expression that represents a pattern of text. Such an expression can contain simple sequences of characters or more complex sequences that use special characters (*metacharacters*) to describe more complex patterns of text.

Regular Expression Metacharacters

Pattern	Matches
alphanumeric character	The character itself (not really a metacharacter)
. (period)	Any single character
[AZ]	One of A or Z
[^AZ]	Any character not A or Z
[A-Z]	Any character in range A to Z
[-AZ]	One of -, A or Z
[0-9]	Any digit 0 to 9

Figure 9-4. Regular Expression Metacharacters AL321.0

Notes:

The shell interprets metacharacters differently from AIX operating system commands.

You may also use the *[:class:]* named classes from POSIX and the shell. For example, for any digit you can use [[:alpha:]], [[:blank:]], [[:cntrl:]], [[:digit:]], [[:lower:]], [[:upper:]], [[:punct:]], [[:space:]], and [[:alnum:]]

You may also use [A-Za-z] to mean choose ONE character: either an uppercase or lowercase letter.

Extending the Pattern

Two ways:

- Anchors
- Multipliers

Anchors are

^	Matches beginning of line
$	Matches end of line

Multipliers apply to patterns. They are:

*****	zero or more occurrences of previous pattern
?	zero or one occurrence of previous pattern
+	one or more occurrences of previous pattern
{m,n}	at least **m** and no more than **n** occurrences of previous pattern ("quoted braces")

Figure 9-5. Extending the Pattern AL321.0

Notes:

Two other metacharacters used within regular expressions specify position in the line of the character(s). The caret “^”specifies the beginning of the line; “^*t*” says any line starting with a *t*. The *$* specifies the end of the line; *7$* says match any lines that end with a *7*.

You can also get wildcard effects by extending the pattern with *multipliers*. The most common are the use of * and quoted braces. The next page deals with braces.

You find the other multipliers in programs that have an extended RE syntax such as egrep, awk and perl.

The * in the shell expands differently than the * for grep. Here is an example of grep’s *:

```
grep 'bugs*' file1
```

This would match: bug, bugs, bugsss, bugssss, and so forth. The * means 0 or more “s” in this example.

When used in a regular expression, the "*" says match zero or more of the previous character. A dot (.) means any single character so to match one or more occurrences of any character use ".*" as the regular expression.

Simple Regular Expression Example

What would the following match?

```
grep '^[M-Z]' phone.list
grep '^[^M-Z]' phone.list
grep '^L.*3$' phone.list
grep '^P.tt' phone.list
grep 'n\*' phone.list
```


Figure 9-6. Simple Regular Expression Examples AL321.0

Notes:

Notice the last example. This is looking for lines that have 0 or more n's on the line. This matches every line in the phone.list. Notice how the "*" is expanded differently by grep then by the shell.

Quoted Braces

To specify the number of consecutive occurrences

Syntax 1: `regular_expression\{min, max\}`

To look for two, three or four occurrences of any combination of the characters 3, 4 and 5 consecutively

```
grep '[345]\{2,4\}' phone.list
```

Syntax 2: `regular_expression\{exact\}`

To look for any lines which have two consecutive "`r`" characters

```
grep 'r\{2\}' phone.list
```

Syntax 3: `regular_expression\{min,\}`

To look for any lines with at least two consecutive "`r`" characters preceded by an "e"

```
grep 'er\{2,\}' phone.list
```


Figure 9-7. Quoted Braces AL321.0

Notes:

We shall see more on the *grep* command later in this unit. Quoted braces offer a more specific wild-card than the asterisk.

`\{min,max\}`

This will search for lines which contain between the minimum and maximum number of the previous RE in a sequence.

`\{min\}`

Here an exact number of repeats are specified, as the maximum number is omitted.

`\{min,\}`

Here the minimum number is set, there is no maximum number, it is equivalent to looking for at least "*min*" repeats.

The single regular expression preceding quoted braces can be regular characters or a pattern of metacharacters. Further characters or patterns will be matched in the usual way:

Quoted Parentheses

To capture the result of a pattern:

Syntax: `\(regular expression\)`

- Stores the character(s) that match the regular expression (within parentheses) in a register.
- Nine registers are available; characters which match the first quoted parentheses are stored in register one, those that match the second quoted parentheses in register two, and so forth.
- To reference a register use a backslash followed by a register number:

  `\1` to `\9`

For example, to list any lines in "phone.list" where there are two identical characters together...

```
grep '\(.\)\1' phone.list
```


Figure 9-8. Quoted Parentheses AL321.0

Notes:

Quoted parentheses store characters from the input line to use as patterns to match against other characters from the input line. If you want to know whether the first two characters on the line are the same, but you don't know what the first character is, quoted parentheses allow the first character to be read into a buffer (or *register*) and then the second character to be compared with the buffer's contents.

"`\(.\)`"matches any single character and puts it into register "`\1`". So the pattern "`\(.\)\1`" identifies a two-character sequence where both characters are the same.

Regular Expressions – Quiz

Using the "phone.list" file, what RE gives:

1. People with six-letter surnames?

2. People with first names of at least four characters?

3. All entries where the number before the dash is the same as that after the dash for example 3-3456?

4. People whose surnames begin with A, B or C?

Figure 9-9. Regular Expressions — Quiz AL321.0

Notes:

Regular expressions may be quoted so that the shell does not interpret the metacharacters.

grep Command

- Search files or standard input for lines containing a match for a specific pattern

  `grep [options] pattern [ file1 file2 . . . ]`

- Valid options:

`-c`	print only a ***count*** of matching lines
`-i`	***ignore*** the case of letters when making comparisons
`-l`	***list*** only the names of the files with matching lines
`-n`	***number*** the matching lines
`-s`	works ***silent***ly, does not display error messages
`-v`	print lines that do NOT match
`-w`	do a whole ***word*** search

Figure 9-10. grep command AL321.0

Notes:

The **grep** command (**g**/**re**/**p**) searches for the specified pattern from STDIN and displays to STDOUT. The search can be for simple strings or regular expressions.

There are other greps in the family:

fgrep only fixed string allowed

egrep allows multiple (either | or) patterns (can also use grep -E)

Historically, early greps did not allow quoted "\" parentheses or braces. Only egrep understood the extended syntax.

The -q option is also helpful in grep. It works quietly. It will not display any matching lines, but does retain a 0 return code if it finds a matching line.

grep Examples

```
$ grep -i "tech support" phone.list

$ grep bob /etc/passwd

$ ps -ef | grep chris

$ ls -l | grep '^d'

$ grep -n '.*' /etc/passwd > \

> passwd.file.numbered.lines

$ egrep 'gene|jean' /etc/passwd
```


Figure 9-11. grep Examples AL321.0

Notes:

1. In a file called **phone.list** in the current directory, search for the string 'tech support' and display to STDOUT. The *-i* will allow grep to find the string whether the letters are uppercase or lowercase. This command will **not** find technical support or support line.
2. This will search the /**etc**/**passwd** file and find bob and display that line.
3. Find any processes that were started by the user named chris — but will also find any command with the same string, that is, `mail chris < letter`.
4. Display only directories in the current directory.
5. Creates a new file that includes all the /etc/passwd information and numbers the lines.
6. Find a line that includes either gene or jean and display to STDOUT.

tr For Translations

The **tr** command translates one set of characters into another:

```
tr LISTIN LISTOUT < in_file > out_file

  – or -

tr -d LISTIN < in_file > out_file
```

- Characters in **LISTIN** are replaced by the corresponding ones in **LISTOUT**
- If **LISTOUT** contains fewer characters than **LISTIN** ignores extra ones from **LISTIN**
- If **LISTOUT** contains more characters than **LISTIN** ignores extra ones from **LISTOUT**
- With **-d**, characters in **LISTIN** are deleted
- Only works with **STDIN** and **STDOUT**
- The **-s** option squeezes multiple characters in a row into one character

Figure 9-12. tr For Translations AL321.0

Notes:

There are two versions of the *tr* command supplied by AIX: the AIX version */usr/bin/tr* (explained above), and a BSD version */usr/ucb/tr* which uses slightly different syntax. The AIX flavor */usr/bin/tr*, will be the one obtained by a default *PATH*. The BSD version pads a short *LISTOUT* to the same length as *LISTIN* using the last character of *LISTOUT*.

Note that *tr* does not allow filename arguments.

tr does not require the brackets in [a-z], and does recognize most \<char> sequences.

tr Examples

Some simple translations...

```
$ print $HOME | tr "/" "-"
-home-team01
$ print "{ { [ ... ] } }" | tr "{}" "()"
( ( [ ... ] ) )
$ print "Lower to upper" | tr "[a-z]" "[A-Z]"
LOWER TO UPPER
$ print "TOP DOWN" | tr '[:upper:]' '[:lower:]'
top down
$ print "vowels and consonants" | tr -d 'aeiou'
vwls nd cnsnnts
$ tr -d '\015' < dos_txt_file > aix_txt_file
$ print 'Lynn Llewellynn' | tr -s "ln"
Lyn Lewelyn
$ _
```


Figure 9-13. tr Examples AL321.0

Notes:

Translate does a character by character translation. For example, print “dad and mom” | tr ‘dad’ ‘mom’ does not say translate dad to mom, it says to translate d to m, a to o, and d to m. The result to the screen would be “mom onm mom”. The -s option, in the above example, squeezes multiple “r”s in a row into 1 r.

The cut Command

`cut` extracts fields or columns from text input

```
cut -dS -s -flist [ file ]

          or

cut -cLIST [ file ]
```

`-dS`	where `S` is the character to take as a delimiter (`<Tab>` ***is default)***
`-s`	with `-d s` suppresses lines that do not contain delimiters
`-fLIST`	specifies a `LIST` of fields to cut out and keep
`-cLIST`	is a `LIST` of columns to cut (character positions)
LIST	- specifies field or column numbers - may contain comma separated values `(m,n)` or a range `(m-n)`

Figure 9-14. The cut Command AL321.0

Notes:

The *cut* command is provided by the AIX operating system. Standard input can be used in place of a named file. The default delimiter is *TAB*.

cut Examples

Field numbering starts at 1

```
$ cut  -d:  -f1,3,4  /etc/passwd  |  head  -3
root:0:0
daemon:1:1
bin:2:2
...
robin:0:0
$ _
$ df | cut  -c-12,35-41 | sort
/dev/hd1             4%
/dev/hd10opt        55%
/dev/hd2            95%
/dev/hd3             6%
/dev/hd4            39%
...
$ _
```

What could this mean?

Figure 9-15. cut Examples AL321.0

Notes:

A "-" by itself at the start of a range means from the first column or field; at the end of a range it means to the end of the line.

What If There Is No Common Delimiter?

- Using `tr -s` and `cut -d`, have the output from the `df` command only show `%used` and `mount point`
- Using only `cut -c`, have the output from the `df` command only show `%used` and `mount point`
- We will do this again later using `awk`

Figure 9-16. What If There Is No Common Delimiter? AL321.0

Notes:

Hint: First, do a regular *df* to become familiar with the output. Notice there is not a common delimiter. Use *tr -s* to squeeze spaces down to one space, and then use *cut* and declare your delimiter between fields to be one space.

The paste Command

As name suggests, **paste** sticks or merges things together

Commonly used to create or format a data stream

Default output is
line from file1 **<Tab>** line from file2

Separators may be changed on command line

Options:
- **-d [dlist]** the delimiter between files (may be a list)
- **-s** make the output a single line in each file

Figure 9-17. The paste Command AL321.0

Notes:

The *paste* command is complementary to the *cut* command. It assembles files into a single multicolumn file — each column formed from a named file. The *dlist* characters are inserted as delimiting characters — either one character that is used to separate all columns, or a list that will be used sequentially — one character for each column join. You may use the *print* special characters to represent a newline, <Tab>, and so forth:

```
paste -dS file1 file2 ... > joined_file
```

Print a three column listing of .ksh files:

```
ls *.ksh | paste - - -
```

Format a listing in three columns using <Tab> <Tab> <Newline> as delimiters

```
ls *.ksh | paste -d"\t\t\n" -s -
```

Paste cars1 and cars4 together `paste cars1 cars4`

In order to paste 1 file on top of another file, use the “cat” command.

```
cat file1 file2 > joined-file
```


Checkpoint

1. What **Reg Ex** can you use to select surnames?
2. What **regular expression** can you use to select text with repeated characters in the surname?
3. What command can you use to select lines in phone.list with four character first names?
4. How could you count the number of processes whose PIDs are in the range 1000-9999?
5. How would you convert spaces to a tab in phone.list?
6. What would this next command accomplish?

   `cut -d: -f1,3,4 /etc/passwd`

7. Using the **paste** command, output the /etc/passwd file so that each line of information is separated by a tab and so that the fifth, sixth and seventh fields are on a separate line from the others. (Hint: make each field a line.)

Figure 9-18. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

Unit Summary

- Understanding **regular expressions**
- Using the **grep** command to select text
- Using the **tr** command to translate characters
- Using the **cut** command to select text fields
- Using the **paste** command to merge data streams

Figure 9-19. Unit Summary AL321.0

Notes:

Answers to quizzes in Unit:
"Simple regular expressions"

1. Matches all lines that start with a capital M through capital Z.
2. Matches all lines that Don't start with a capital M through capital Z.
3. Matches lines that start with a capital L, followed by 0 or more characters, and ends with a 3. (Lee Llewellyn)
4. Matches lines that start with a capital T, followed by one character, followed by 2 r's. (Terry Terrell)
5. Matches lines that have 0 or more n's in a row. (all lines in phone.list)

"Regular Expressions quiz"

1. `grep '^[A-z]\{6\}, ' phone.list`
2. `grep', [A-z]\{4,\}' phone.list`

3. `grep '\([0-9]\)-\1' phone.list`
4. `grep '^[ABC]' phone.list`

“What if there is no common delimiter?”

1. df | tr -s “ “ | cut -d “ “ -f4,7
2. df | cut -c35-40,56-

Unit 10.The sed Utility

What this unit is about

This unit describes how the sed utility manipulates data.

What you should be able to do

After completing this unit, you should be able to:

- Use sed to edit file contents
- Understand sed advanced features

How you will check your progress

Accountability:

- Checkpoint questions
- Hands-on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use the stream edit utility – `sed`
  - Line selection
  - Substitution
  - Delete
  - Print
  - Append, insert, and change
  - Multiple editing
  - And more

Figure 10-1. Unit Objectives AL321.0

Notes:

sed

There are several ways of running **sed**:

```
sed 'edit-instructions' filename
command | sed 'edit-instructions'
sed -f command.file filename
```

Note: The input file is not changed or overwritten by **sed**!

Figure 10-2. sed AL321.0

Notes:

The *sed* command can be invoked in a number of ways. The *sed* command takes its input from standard input unless a *filename* is specified on the command line; it writes its output to standard output. Thus *sed* is a filter and can be used within a pipe.

The output of *sed* can be redirected to a file; ***a word of warning***, never try to redirect the output of *sed* back to the original input file as this is not supported by the shell and due to the order in which the shell processes the command line, you will end up losing the original contents of the input file.

The edit instructions can be provided on the command line, or in an ASCII file if *sed* is invoked with the *-f* option.

Line Selection

The **sed** instructions operate on <u>all lines of the input</u>, unless you specify a ***SELECTION*** of lines:

```
sed 'SELECTION edit-instructions'
```

SELECTION can be

- A single line number
  - `1` = line `1` of the input
  - `$` = the `last line` of the input
- A range of line numbers
  - `5,$` = from line `5` to the `end` of the input
- A regular expression to select lines matching a pattern
  - `/string/` = selects all lines containing "`string`"
- A range using regular expressions
  - `/^on/,/off$/` = from the first line beginning with "`on`" to the first ending in "`off`"

Figure 10-3. Line Selection AL321.0

Notes:

Regular expressions used for line selection must be delimited by the '/' character.

The Substitute Instruction

This instruction changes data:

Syntax: `s/old string/new string/`

Some examples

1. To replace the first occurrence of "Smith" on each line with "Smythe"
   `sed 's/Smith/Smythe/' phone.list`

2. To replace the same as above using a different delimiter
   `sed 's!Smith!Smythe!g' phone.list`

3. To replace every match in a line, add the "global"

```
print xxx | sd 's/x/y/'     # responds with yxx
print xxx | sd 's/x/y/g'    # responds with yyy
print xxx | sd 's/x/y/2'    # responds with xyx
```


Figure 10-4. The Substitute Instruction AL321.0

Notes:

The data file phone.list is same as that in Unit 9.

A "\" can be used to escape any special meanings of characters in your strings or addresses, that is, "\." is a dot, and "\&" a literal ampersand.

In addition to the *"g"*, you can specify that the nth occurrence is to be replaced by putting a number *"n"* in place of the *"g"*.

To precede each phone number with "Tel:"

```
sed '/[0-9]\{3\}-[0-9]\{4\}/s//Tel: &/g' phone.list
```

The "&" is used to redisplay what was previously matched in the *SELECTION*.

Substitutions - Quiz

1. Convert the "phone.list" into just a name list, that is, get rid of the phone numbers

   Desired output:

   ```
   Terrell, Terry
   Franklin, Francis
   Patterson, Pat
   ..., ...
   ```

   ```
   sed 's/__________________//' phone.list
   ```

2. Convert the "phone.list" file to a first-name and number list

   Desired output:

   ```
   Terry                617-7989
   Francis              704-3876
   Pat                  614-6122
   ...                  ...
   ```

   ```
   sed 's/__________________//' phone.list
   ```


Figure 10-5. Substitutions - Quiz AL321.0

Notes:

sed with Quoted Parentheses

- Repeating the first character

```
$ print "1234" | sed 's/^\(.\)/\1\1/' 11234
$ _
```

any single character to register 1 (points to `.`)

register 1 is repeated (points to `\1\1`)

- Stripping out all but the first and last characters

```
$ print "1234"|sed 's/^\(.\).*\(.\)$/\1\2/' 14
$ _
```

character to register 1

character to register 2

registers 1 and 2

Now it's your turn...

Working on the "phone.list" file, abbreviate everyone's first name to an initial and a period (use register 1 to store each initial)

```
sed 's/______________/__________/' phone.list
```


Figure 10-6. sed with Quoted Parentheses AL321.0

Notes:

Delete and Print

This command removes text:

Syntax: `SELECTIONd`

- To delete all lines in the output stream:

```
$ sed  d  phone.list
```

- Delete from line 5 to the end of the file:

```
$ sed  '5,$d'  phone.list
```

- By default `sed` writes out every line it selects
  - Makes print instruction "`p`" by itself redundant:

```
$ cat in.file
line 1
line 2
$ sed p in.file
line 1
line 1
line 2
line 2
$ _
```


Figure 10-7. Delete and Print AL321.0

Notes:

- Delete the last line of output

```
$ sed '$d' phone.list'
```

- To remove any blank lines

```
$ sed '/^$/d' phone.list
```

Print is of more use with the **-n** option — to suppress normal printing of input lines, and only print a ***SELECTION***

```
$ sed -n in.file              #select all lines
line 1
line 2
$ sed -n '/2/p' in.file       #select lines with a "2"
line 2
$ _
```


Append, Insert, and Change

These instructions add or modify text

Syntax: `SELECTIONx\`
`text`

Where **x** is

i inserts **text** before a single selected line

a appends **text** after a matched line

c changes a range of matched lines into **text**
SELECTION can be a single line or a range
but only one copy of **text** is printed in its place

Figure 10-8. Append, Insert, and Change AL321.0

Notes:

SELECTIONS can be:

- line number
- regular expression
- range of lines

Example ...

```
$ sed '1a\
> Add after line 1 of the input' in.file

Line 1
Add after line 1 of the input
Line 2
$ _
```

Command Files

- A **sed** command file consists of one or more **sed** instructions on separate lines
- Command files are useful in many situations:
  - Storing multiple instructions
  - Storing a long complex command
  - For commands which may need to be modified and reused
- Use the **-f** option to use a command file
- Example...

```
$ cat sedscript.sed
  s/ GA/, Georgia/
  s/ FL/, Florida/
  s/ IL/, Illinois/
  s/ TX/, Texas/
  s/ MD/, Maryland/
  s/ DC/, District of Columbia/
$ sed -f sedscript.sed addrs.file > new.addrs.file
$ _
```


Figure 10-9. Command Files AL321.0

Notes:

It is sometimes useful to add an extension to a script to denote the type of its contents. You have seen the use of *.ksh* for script files; here we adhere to the same convention and use *.sed* for our sed scripts.

A Practical Example

Converting a "BookMaster" script to a "wysiwyg" file

```
:ul.
:li.An unordered list starts with ":ul.".
:li.Each list item is tagged with ":li." - it
appears as an indented bullet point.
:li.The end of the list is marked by ":eul."
:eul.
```

Strategy:

1 Remove lines which contain just ":ul." or ":eul."

2 For lines that start with ":li.", substitute the ":li." with a dash followed by five spaces

```
$ cat  bkm.wysi.sed
/^:e*ul\.$/d
s/^:li\./-     /
$ sed  -f bkm.wysi.sed  bookmaster.file > wysi.file
$ cat wysi.file
-   An unordered list starts with ":ul.".
-   Each list item is tagged with ":li," - it
      appears as an indented bullet point.
-   The end of the list is marked by ":eul."
```


Figure 10-10. A Practical Example AL321.0

Notes:

Multiple Editing Instructions

- Multiple instructions can be applied to each line
- Each instruction must be on a separate line

Example:

```
$ sed      '/[1-4]-/s/$/ (Bldng 1) /
>           /[5-9]-/s/$/ (Bldng 2) /'  phone.list

Terrell, Terry          617-7989     (Bldng 2)
Franklin, Francis       704-3876     (Bldng 1)
```

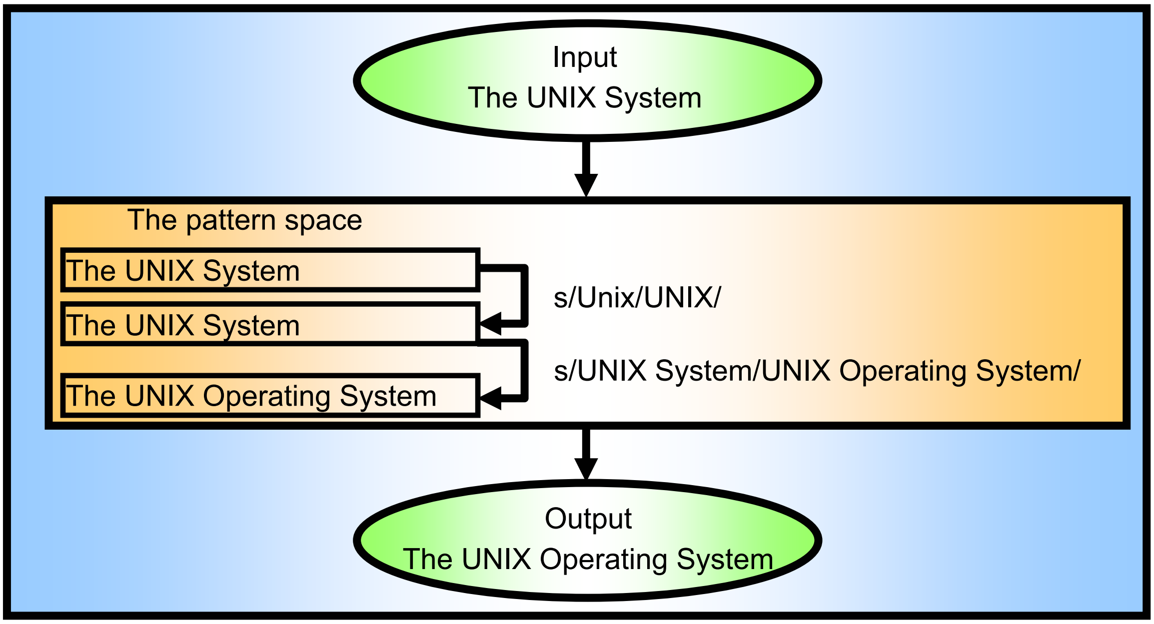

Figure 10-11. Multiple Editing Instructions AL321.0

Notes:

Why did Terry get Bldng2 and Francis get Bldng1?

Hint: Look at the number in front of the dash in the phone number.

Internal Operation



- `sed` applies all editing instructions to a line before it moves on to the next line.
- It holds each input line in a "pattern space" or temporary buffer while editing instructions are applied in sequence.

Figure 10-12. Internal Operation AL321.0

Notes:

The command is:

```
$ print “The Unix System” | sed ‘s/Unix/UNIX/ \
> s/UNIX System/UNIX Operating System/’
The UNIX Operating system
```

Grouping Instructions

Braces "{" "}" are used for two purposes:

- One **SELECTION** inside another (*nest*)
- To apply multiple instructions to the same **SELECTION** range (*group*)

Example...

```
/\.ul/,/\.eul/{
              /^$/d
              /:li\./-    /
              }
```

SELECTION range

instructions for the SELECTION range

- The instruction "**/^$/d**" (delete blank lines) will be applied to a range of lines between one that contains an "**.ul**" and up to the first containing an "**.eul**", as will the "**/:li\./-   /**"
- The special meaning of the dot preceding "**ul**" and "**eul**" is escaped by the use of a backslash

Figure 10-13. Grouping Instructions AL321.0

Notes:

Here is the contents of myfile:

```
philadephia 1
dc 2
start 3
dc 4
philly 5
dc 6
nyc 7
dc 8
end 9
dc 10
```

Suppose we ran the following command:

```
sed '/start/,/end/{ /philly/,/nyc/s/dc/district/g }' myfile
```

Which "dc" or "dc"s would change to "district"? (Answer is on summary page)

Checkpoint

1. Write a command line script that displays a `ps -ef` with your username as the owner of init.

2. How can I make phone.list appear double spaced?

3. Cat out the sulog (located in `/var/adm/sulog`) and change all + to the word "successful" and all - to the word "unsuccessful" using sed.

4. Using `sed`, insert `#!/usr/bin/ksh` as the first line of a program called program1 and store in program2.

Figure 10-14. Checkpoint AL321.0

Notes:

Write down your answers here:

1.
2.
3.
4.

Unit Summary

- Use of **sed** to automate repetitive editing tasks
  - Line selection
  - Substitution
  - Delete
  - Print
  - Append, insert, and change
  - Multiple editing
  - And more

Figure 10-15. Unit Summary AL321.0

Notes:

Answers to quizzes in unit:

"Substitutions quiz"

```
1. sed 's/[0-9]\{3\}-[0-9]\{4\}//'phonelist
2. sed 's/.*, //'phone.list
```

"sed with quoted parenthesis"

```
1. sed 's/, \(.\).*    /, \1.    /'phone.list
```

"Grouping instructions"

```
1. The 'dc's on line 6 would change to 'district'
```

Unit 11. The awk Program

What this unit is about

This unit describes how to use and program in *awk*.

What you should be able to do

You should be able to:

- Use awk to generate formatted output from input files
- Create and use a simple awk script
- Be aware of the more advanced and powerful features of awk programming that are available

How you will check your progress

Accountability:

- Checkpoint questions
- Hands-on exercises

Unit 11 Objectives -- The awk Program

After completing this unit, you should be able to use the awk utility by looking at:

- Regular expressions in **`awk`**
- Basic **`awk`** programming
- **`BEGIN`** and **`END`** processing
- Flow control – **`if`**, **`while`** and **`for`**
- Leaving loops – **`continue`**, **`next`** and **`exit`**
- **`awk`** arrays
- Better printing
- **`awk`** functions

Figure 11-1. Unit Objectives AL321.0

Notes:

What Is Awk?

- **awk** is a programming language used to manipulate text
- **awk** sees data as words (**fields**) in a line (**record**)
- An **awk** command consists of a **pattern** and an **action** comprising one or more statements

```
awk  '/pattern/ { action }'  file  ...
```

- **awk** tests every **record** in the specified ***file(s)*** for a **pattern** match. If a match is found, the specified **action** is performed
- **awk** can act as a filter in a pipeline or take input from the keyboard (standard input) if no ***file(s)*** are specified

Figure 11-2. What Is Awk? AL321.0

Notes:

awk is sometimes called a report generator tool.

awk program text may be thought of as a data driven program.

There are at least three major implementations of awk in the field:

- Original (Bell Labs) awk and its updated nawk
- GNU awk - the Free Software Foundation Implementation
- Vendor specific versions, usually based in POSIX

It is best to consult your documentation to discover which is in use. They have very slight differences.

Sample Data – awk

```
Lastname,<Space>Firstname<Tab>nnn-mmmm
```

```
$ cat phone.list
Terrell, Terry            617-7989
Franklin, Francis         704-3876
Patterson, Pat            614-6122
Robinson, Robin           411-3745
Christopher, Chris        305-5981
Martin, Marty             814-5587
Llewellyn, Lynn           316-6221
Jansen, Jan               903-3333
Llewellyn, Lee            817-8823
$ _
```

The same file is used in the RE and sed units

Figure 11-3. Sample Data - awk AL321.0

Notes:

The *phone.list* file will be used again. There is a single space character after the comma and a *tab* after the Firstname.

awk Regular Expressions

- Like sed, regular expressions are "`/`" delimited – "`/x/`"
- All of the previous regular expression metacharacters can be used with **`awk`**

`awk` has the following extensions

`/x+/`	for one or more occurrences of **`x`**
`/x?/`	zero or one occurrence of **`x`**
`/x|y/`	matches either "**`x`**" or "**`y`**"
`(string)`	groups a string – for use with **`+`** or **`?`**

Example:

`/t[i|o]?n[iey]+/`

matches: tiny, tony, toni, toney, tone, tny (and others...)

Figure 11-4. awk Regular Expressions AL321.0

Notes:

In this example, the “|” symbol is optional.

The programming language Perl has similar extensions.

awk Command Syntax

- Basic syntax

```
pattern   { actions }
pattern
          {
           actions
          }
```

- Multiple statements in an action
  – Use a line break or a semi-colon

```
$ awk '/Ll/ { print $1 ; print $3 }' phone.list
```

- Comments start with a # until the end of a line

```
$ awk '/Ll/ { print $1 # prints field 1
> print $3 }' phone.list
```


Figure 11-5. awk Command Syntax AL321.0

Notes:

The three basic syntax awk program lines work as follows:

- If *pattern* is present, then do the actions.
- If *pattern* is present but no actions are specified, this defaults to printing the complete current line (record) to stdout.
- If *pattern* is not present, then **all** lines (records) match and each line is processed by the specified actions.

Multiple actions may be specified.

The print Statement

One useful **action** is to `print` the data!

```
awk '/pattern/ { print }' ifile > ofile
```

`awk` tests each `record` of the input for the specified *pattern*

- When a match is found the `print` statement sends the entire `record` to standard output

Figure 11-6. The print Statement AL321.0

Notes:

This is the default action.

awk Fields and Records

- Referencing fields in a record

`$0`	=	the entire record
`$1`	=	the first field in the record
`$2`	=	the second field in the record
...		

- To print Jansen's phone number from phone.list:

```
$ awk '/Jansen/ { print $3 }' phone.list
903-3333
```

- To place that phone number into a variable:

```
$ JanNum=$(awk '/Jansen/ { print $3 }'  phone.list)
```


Figure 11-7. awk Fields and Records AL321.0

Notes:

awk sees all input as a *record* which is made up of *fields*. By default, a record is delimited by a newline ("\n"). An awk field is delimited by whitespace by default. You will see later that these defaults may be changed.

Note that the RE metacharacters "^" and "$" refer to the beginning or end of a **field** respectively.

print Examples

- Special character sequences are available for use in print strings or regular expressions

`\n`	newline
`\t`	tab
`\r`	carriage return

```
$ awk '/^Ll/ { print "Name:\t", $1
>       print "Number:\t", $3, "\n" }' phone.list
Name:         Llewellyn,
Number:       316-6221

Name:         Llewellyn,
Number:       817-8823

$ _
```


Figure 11-8. print Examples AL321.0

Notes:

print can take an expression following an I/O redirection to specify a pathname. The print command always ends with an end of record character. Again, this is usually newline. There is another output command, *printf* that you will see later (it allows better formatting).

Comparison Operators and Examples

To compare regular expressions or strings with values:

`==`	equal to	`!=`	not equal to
`<`	less than	`<=`	less than or equal to
`>`	greater than	`>=`	greater than or equal to
`~`	matched by RE	`!~`	not matched by RE
`||`	logical "or"	`&&`	logical "and"

Examples:

`$1 ~ /x/`	field one matches regular expression **x**
`$1 != "No"`	field one doesn't match string "**No**"

You can use comparison operators in the pattern to select records

```
$ awk '$1 == "Terrell," { print $2, "Smythe" }' phone.list
Terry Smythe
$ _
```


Figure 11-9. Comparison Operators and Examples AL321.0

Notes:

This example finds records with the first field (Lastname) starting with T or the phone number starting with 4 or 6.

```
$ awk '$1 ~ /^T/ || $3 ~/^[46] / {
print }' phone.list

Terrell, Terry      617-7989
Patterson, Pat      614-6122
Robinson, Robin     411-3745
$ _
```

Arithmetic Operators

You can use the following operators to perform arithmetic:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	remainder
`^`	exponential (x^y, raise x to the power y)
`++x  x++`	pre and post increment
`--x   x--`	pre and post decrement
`=`	assignment (x = 4)

x **op=** y   x = x **op** y

for: `+=, -=, *=, /=, %=`

Example

count = count + 2

count += 2

Figure 11-10. Arithmetic Operators   AL321.0

Notes:

count = count +2

Sets *count* to 2 the first time, because *count* will be automatically initialized to zero.

num *= 8

Sets *num* to 8 times its value. The first time this will make *num* zero.

val ^= 2

Raises *val* to the power of 2.

User Variables and Expressions

You can define your own variables:

- Names must:
  - Start with a letter or underscore
  - Be followed by letters, underscores, or digits

- **awk** does not require variables to be defined before use

Variables are initialized as empty (numerically zero)

- The empty string is null ("")
- Referenced by name only
- Can be passed through from the command line

Figure 11-11. User Variables and Expressions AL321.0

Notes:

It is possible to pass parameters into an *awk script*.

```
awk -v var=val -f commands_file data_file
- or -
awk -f commands_file variable1=val1 var2=2 FS=\: data_file
```

You can use these methods to assign values to built-in variables or to define your own variables.

BEGIN and END Processing

You have seen the **pattern** and **action** with **awk** syntax
You can also have actions at the beginning and end of input
You use the special patterns **BEGIN** and **END**

```
awk 'BEGIN { begin_action }
     pattern { action }
     pattern { action }
     END { end_action }'   file...
```

Where:

BEGIN means execute the *begin_action* before any input read

END means execute *end_action* once all input has been read

Figure 11-12. BEGIN and END Processing AL321.0

Notes:

These special patterns can be very handy for explicit variable initialization or explicit EOF processing.

BEGIN without END Example

You can use `BEGIN` to print a header to the output...

Here we have a `BEGIN` with no `END`

```
$ awk  'BEGIN  { print "Words in phone.list"}
>               { wcount = wcount + NF
>                print wcount }' phone.list
Words in phone.list
    3
    6
    9
    ...
    24
    27
$ _
```

The statements within the second set of braces were performed on every line of "phone.list" as no pattern was specified

Figure 11-13. BEGIN without END Example AL321.0

Notes:

To determine the value of *NF* (total number of fields in the current record), an input line has to be read.

END without BEGIN Example

You can use **END** to print a trailer or summary after the output:

```
$ awk  '{ wcount = wcount + NF }
> END { print "Words in phone.list: ",
          wcount }' phone.list
Words in phone.list: 27
$ _
```

- The statement within the first set of braces refers to the main **action**
- The main **action** is performed on every line of the file "phone.list", so the final value of **wcount** holds the total number of fields (or words) in the file
- At the end of the input **END** actions are processed
- This prints the heading with the total word count

Figure 11-14. END without BEGIN Example AL321.0

Notes:

The built-in variable ***NF*** refers to “Number of Fields”

Built-In Variables

awk provides a number of useful built-in variables:

FILENAME	the name of the current ***file***
NF	total number of ***fields*** in the current record
NR	number of ***records*** encountered
FS	***input field separator*** (default is space or tab)
RS	***input record separator*** (default is newline)
OFS	***output field separator*** (default is space)
ORS	***output record separator*** (default is newline)

Figure 11-15. Built-In Variables AL321.0

Notes:

If *NR* is placed inside an *END* action, it is the number of the last record processed.

FS can be set using a regular expression to define several possible field separators. A single space is taken as any number of spaces and tabs. "`[ ]`" would be taken as a single space, "`\t`" a tab and "`\t+`" as several (one or more) tabs.

If *RS* is set to the null string "", *awk* will assume multiline records, that is, a single record may be more than a single line.

Built-In Variables Examples (1 of 2)

```
$ cat employee.list
Name, company, city, phone
Drew A. Chart, IBM, Wash. D.C., 202-555-3788
Wanda C. Results, IBM, Denver, 303-555-8068
Hyde N. Sikh, IBM, Atlanta, 404-555-3523
$ _


$ awk 'BEGIN { FS = "," ; OFS = ":" }
>         { print $1, $4 }' employee.list
Name: phone
Drew A. Chart: 202-555-3788
Wanda C. Results: 303-555-8068
Hyde N. Sikh: 404-555-3523
$ _
```


Figure 11-16. Built-In Variables Examples (1 of 2) AL321.0

Notes:

Built-In Variable Examples (2 of 2)

```
$ cat authors
Drew A. Chart
Wash. D.C.
202-555-3788

Wanda C. Results
Denver, CO
303-555-8068

Hyde N. Sikh
Atlanta, GA
404-555-3523

$ awk 'BEGIN { FS="\n" ; RS="\n\n" ; OFS="\n" ;
ORS="\n\n"}
> { print $1, $3
> } ' authors
```

FIELD 1
FIELD 2
FIELD 3
RECORD SEPARATOR

Figure 11-17. Built-In Variables Examples (2 of 2) AL321.0

Notes:

And the answer is:

```
Drew A Chart
202-555-3788

Wanda C. Results
303-555-8068

Hyde N. Sikh
404-555-3523
```

if - else if - else Statement

Syntax:

```
awk '{
     if (first logical test) {
         action if test true
     }
     else if (second logical test)  {
         action if first test false and
         second test true
     }
     else  {
         action if both tests false
     }
}' file
```

Example:

```
$ awk '{
     { if ( $2 == "Terry" )
           print $2 ", " $1 "--" $3
     }
}' phone.list
```


Figure 11-18. if - else if - else Statement AL321.0

Notes:

You can see that *awk* is a proper programming language. It has variables, input/output facilities and program logic constructs.

The *else if* and *else* parts of the *if* statement are optional. Comparison operators (">", "<", "==", and so forth.) must be used in the logical tests of the *if* statement to test for a value. Don't use the assignment operator "=", which assigns a value to a variable, if you are testing for equality use "==".

```
$ awk '{
     { if ( $2 == "Terry" )
           print $2 ", " $1 "--" $3
     }
}' phone.list
```

which gives

```
Terry, Terrell,--617-7989
```

The while Loop

Syntax:

```
awk   ' {
      while    (condition)          {
                   action
      }
      } ' file
```

Example:

```
awk   ' {i = 1
      while (i <= 4)
      {  print $i ; ++i }
      } ' file
```


Figure 11-19. The while Loop AL321.0

Notes:

The for Loop

Syntax:

```
awk  '{
         for (initialize; test; increment)
         {    action
         }
       }' file
```

Examples...

- To read and print each field of the current input line

```
for (i=1; i<=NF; i++){
     print $i
    }
```

- To print from the last field to the first of the current line

```
for (i=NF; i>=1; i--){
     print $i
    }
```


Figure 11-20. The for Loop AL321.0

Notes:

The *for* syntax can be rewritten as a *while* loop:

```
awk '{
            initialize; while (test) {
                        action; increment
                  }
    }'  file
```

The break, continue and next Statements

The `continue` statement stops the current innermost loop iteration and starts the next one:

```
awk  '{
       y = 42
       for (x=1; x<=NF; x++) {
          if (y!=$x)
          {
           continue # break
          }
          print x, $x
          }
       }'  file
```

The `next` statement causes the next `record` to be read in, and the program to start from the first `pattern { action }` block again:

```
awk 'BEGIN { action }
   pattern {
            action
            action
            next
            action
          }
   END { action }' file
```


Figure 11-21. The break, continue and next Statements AL321.0

Notes:

In *awk* there is also a *break* statement. This functions similar to a break in shell and leaves the processing of the current loop.

Unconditional control statements:

break	Break out of "while" or "for" loop.
continue	Perform next iteration of "while" or "for" loop.
next	Get and scan next line of input.
exit	Finish reading input and perform END statements.

The exit Statement

The **exit** statement jumps to any **END** processing – or out of the program if already in the **END** section. An exit code can be passed back to the shell:

```
$ awk  '{
>          y = 42
>          for (x=1; x<=NF; ++x) {
>               if (y==$x) {
>                    print x, $x
>                    exit
>               }
>          }
>        }
>        END   { exit 3 }' file
$ print $?
3
$ _
```


Figure 11-22. The exit Statement AL321.0

Notes:

Arrays

- **`awk`** allows **`array`** variables
- An **`array`** is a variable with an index
- An index is an expression in brackets
  - For example, **`array[10]`**
- **`awk`** arrays are **associative**
  - Index can be a string or number
  - No implicit order
  - To access all elements, use the **in** operator

    `for ( var in array_name )`
- Be aware that all **`array`** indices are internally strings

Figure 11-23. Arrays AL321.0

Notes:

To define an array element, you just use it. As with any *awk* variable no definition or initialization is needed. You can iterate through an array by numeric index as in:

```
for ( i=1; i < 6; i++ )
        arr[ i ] = i
```

If you have a record with two text fields as fields 1 and 2, such as a database with a word followed by a definition phrase, you can use the associative array concepts as in:

```
arr[ $1 ] = $2
```

If you want to delete an array, it is not sufficient to null the value. Use the *delete* command:

```
delete arr[ i ]
```

printf for Formatted Printing

- One use of **`awk`** is as a report generator
- Better printing formats required
  - Use **`printf`**
  - **`printf`** **syntax:** **`printf ( fmt [, args] )`**
- Parentheses are optional
- **`fmt`** is usually a string constant with format specifications
- Specifiers are like the C language **`printf`**
- Format specification: **`%<char>`**
  - **`%s`** string
  - **`%d`** decimal integer
  - **`%f,%e`** floating point (fixed or exponent notation)
  - **`%o`** unsigned octal
  - **`%%`** literal percent

Figure 11-24. printf for Formatted Printing AL321.0

Notes:

printf allows better formatting of output than *print*. For those who are familiar with the language C or C++, the format specifiers are very similar. For *awk*, remember that print will terminate each occurrence with the ORS but printf does not — hence the "\n" usually found at the end of format string.

Do not forget to make sure that you supply enough arguments to satisfy the number of format specifiers. It is a common error to make at first.

printf Formats

- Format specification strings can use modifiers
  - `%-width.precision`
  - If width used, contents are right justified
  - Use `-` (minus/hyphen) after `%` to left justify
  - Precision controls
    - Number of digits to right of decimal point for numeric values
    - Maximum number of characters to print for string values
- To print Hello within #'s right justified in 10 character field
  - `printf ("#%10s#\n", "Hello")`
- To print a number left justified with minimum three characters
  - `printf ("%-3d\n", $1)`

Figure 11-25. printf Formats AL321.0

Notes:

You get more control of the output the more you specify but maybe at the cost of more complexity.

Functions in Awk

- There are four types of functions
- Three types are built-in to **awk**
  - General
  - Arithmetic
  - String
- The fourth type is a user defined function
  - General functions include
    - Close
    - System
    - Getline

Figure 11-26. Functions in Awk AL321.0

Notes:

The general functions allow the explicit *close()* of a file so that it can be reopened or used later in the awk script. It also has the benefit of avoiding running out of file descriptors etc. *system()* takes a string argument which is the external command to use. *getline* reads the input stream for the next record.

Built-In Arithmetic Functions

Functions available include:

`atan2(y,x)`	arctangent of y/x in range $-\pi$ to $+\pi$
`cos(x)`	cosine of x (x in radians)
`sin(x)`	sine of x
`exp(x)`	*e* to the power x
`log(x)`	natural log of x
`sqrt(x)`	square root of x
`int(x)`	truncated value of x
`rand()`	pseudo-random number r, $0 \leq r \geq 1$

Figure 11-27. Built-In Arithmetic Functions AL321.0

Notes:

The list of arithmetic functions includes all the usual facilities. One not shown but available is *srand* that will set the random number seed. See the online documentation for details.

Built-In String Functions

Functions available include:

`length(s)`	length of string **`s`** or of $0 if **`s`** not supplied
`index(s,t)`	position of substring **`t`** in **`s`** or zero if not present
`match(s,r)`	position in **`s`** of where RE **`r`** begins or zero
`sub(r,s,t)`, **`gsub(r,s,t)`**	substitute **`s`** for **`r`** in **`t`**, returns 1 for OK uses $0 if **`t`** not supplied (gsub does all matches)
`split(s,a,sep)`	parses **`s`** into array **`a`** elements using field separator **`sep`** (use **`RS`** if not supplied)
Set by match() **`RSTART`**	start of the match (same as the return value)
`RLENGTH`	length of the matching sub-string

Figure 11-28. Built-In String Functions AL321.0

Notes:

The *sub* and gsub syntax can also be written out as follows:

```
sub(regular.exp, replacement, target)
```

Built-In String Functions Examples

```
1 awk '{print len($1)}' myfile
2 awk '{print index($1, "a")}' myfile
3 awk '{print match($1, "i.a")}' myfile
4 awk '{match($1, "i.a"); print RSTART, RLENGTH]'
  myfile
5 awk '{print gsub(/a/,"b",$1), $0}' myfile
6 awk '{gsub(/a/,"b",$1)' print $0}' myfile
7 awk '{split ($0,var,":"); print var[1], var[2],
  var[6]]' /etc/passwd
```


Figure 11-29. Built-In String Functions Examples AL321.0

Notes:

Examples:
1. prints out the length of the first field of each line of myfile (only use print if you want the length to print to the screen)
2. prints out the position # where it found an "a" in the first field of each line in myfile
3. prints out the position # of the match of the pattern "i.a" (where . represents any single character) in the first field of each line
4. match does not print anything, it just finds the match of pattern "i.a" in $1 RSTART and RLENGTH print out start of matched pattern, and length of pattern
5. prints the number of substitutions and prints the entire record (line) ($0) with substitutions in place
6. does the substitution of "b" for "a" in $1 and then print $0 (the line) with the substitutions in place, but does not print the number of substitutions it did.
7. splits the record (line) ($0) in an array named var - each element is deliminated by a : (try this one yourself!)

Checkpoint

1. With **awk**, what happens if I don't supply a **pattern**?

2. With **awk**, what happens if I don't supply the **action**?

3. **awk** causes the **-f** option to read instructions from a default line.

4. **awk** must have both the **BEGIN** and **END** statements. T or F

5. Using **awk**, have the output from the **df** command only show the **% used** and **mount point**.

Figure 11-30. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

Unit Summary

- Regular expressions in **awk**
- Basic **awk** programming
- **BEGIN** and **END** processing
- Flow control – **if**, **while** and **for**
- Leaving loops – **continue**, **next** and **exit**
- **awk** arrays
- Better printing
- **awk** functions

Figure 11-31. Unit Summary AL321.0

Notes:

Unit 12.Good Practices and Review

What this unit is about

This unit discusses general design, overall layout, ease of maintenance, and general performance of shell scripts. It also provides a brief course summary.

What you should be able to do

After completing the unit, you should be able to:

- Understand why “plan and design” comes before “write and test”
- Use comments to your advantage
- Debug your code
- Understand some performance issues

How you will check your progress

Accountability:

- Checkpoint questions

Unit Objectives

After completing this unit, you should be able to:

- Write any serious script you need to
- Plan the activity
- Produce **good code**
- In this unit:
  - Planning and design
  - Documentation
  - Debugging
  - Performance issues
  - Guidelines for scripting
  - Course summary

Figure 12-1. Unit Objectives AL321.0

Notes:

Planning and Design

- As well as your favorite design methodology (Flow Charts, Data-Flow, SSADM, and so forth) consider:
  – Functionality – clearly defined specification
  – Modular design – use of functions, separate programs
  – Environment – variables, directories
  – File naming convention – for temporary files, results
  – Testing – individual units, integration tests, boundary conditions
  – Debugging code – do not forget the next maintainer

Figure 12-2. Planning and Design AL321.0

Notes:

Without a specification, how do you know when you have finished? The specification should include a description of the required outputs and return codes, files that are to be used or created, and any environment variables that are to be used.

Modular coding often means that you can reuse bits in other programs — sharing common functions. It is also a lot easier to read, understand and maintain.

It might seem trivial, but a file naming convention will help you later on when you try to interface different programs. This may be something that the specification has set-out for you to follow.

If you don't plan to test your code from the start, you will find it much more time-consuming later on. Testing should be with sample data, or whatever is typical of the final environment, and with extreme cases — boundary testing. If you have a program that deals with numbers, test the smallest and the largest values that you can have, plus and minus one.

By including debugging code, activated by setting some flag variable for example, you can make it much easier to track down the source of a bug later on.

Use of Comments

- A good programmer uses comments in a program to:
  - Explain the purpose and function of the code at key points
  - Describe the use of variables
  - Explain complicated syntax
  - Give yourself the credit (or the blame) for your work
  - Mark corrections or additions

- Remember to update the comments with the code

Figure 12-3. Use of Comments AL321.0

Notes:

Key points for your script might be function definitions and the start of the script. With variables perhaps you should describe the expected values. If you have a complicated or clever piece of script or syntax and you do not describe it in comments, then you may well forget what, why, and how you did it.

When giving yourself the credit do not forget the versions and dates, even if you are using one of the source code control tools. When you do the change, mark it at the top of the script (in your version history perhaps) and where the code changed.

Commenting Out

- Lines can be commented out using the # comment character:
  - `# command arg1 arg2`
  - – No Shell interpretation is performed to the right of #
  - – Legal anywhere, except as the only statement in a flow-control construction (`if`, `while`, `until`)

- The "null" command can be used where commenting out would not work:
  - `:  command arg1 arg2`
  - – Arguments are ignored, but processed as usual
  - – Always returns 0 (true)

Figure 12-4. Commenting Out AL321.0

Notes:

Watch out for the second syntax using the null (:) command. When you supply variables or arguments they are evaluated and can cause unwanted side effects.

Script Layout

- Some things must be done in a certain order other things can be arranged for **good code**:
  - Shell control line (first in script)
    `#!/usr/bin/ksh` or `#!/bin/bash`
  - Header comments
  - Validation of options
  - Testing of arguments
  - Initialization of variables
  - Function definitions
  - Main code

Figure 12-5. Script Layout AL321.0

Notes:

Debugging Code

Shell options can help with syntax checking:

- To check the syntax of a script without running it

  `set -o noexec` or `set -n`

- For the shell to print its input as it reads it

  `set -o verbose` or `set -v`

- An execution trace displays each command before it is run and after command line processing

  `set -o xtrace` or `set -x`

- For functions, use

  `typeset -ft function ...`

Figure 12-6. Debugging Code AL321.0

Notes:

The *PS4* variable is expanded and displayed with each *xtrace* line — set it to *$LINENO* to get Script line numbers.

Notice that you can debug a single function by appropriate use of *typeset*.

DEBUG Traps

After each simple command the shell issues the fake signals

- **DEBUG**
- **ERR**
- **EXIT**

The order is **DEBUG**, **ERR**, then any other traps, and lastly **EXIT**

To display the environment after each command set this trap

```
trap  "set"  DEBUG
```

When a command has a non-zero exit status, the shell sends the **ERR** signal

For example, to see what signals are causing error exits set this trap

```
trap  "kill -l $?"  ERR
```


Figure 12-7. DEBUG Traps AL321.0

Notes:

DEBUG is technically a fake signal, that is, it is not raised by the operating system but the Korn shell itself.

Main program traps are inherited by functions, and in the Korn shell, function traps are local to functions.

The *kill* command syntax used above was introduced with AIX Version 4. You might use "*print $?*" with earlier versions of AIX to see the return code for each error exit.

The bash shell supports DEBUG and EXIT.

Maintaining Code

- Documentation: Design and comments
- Clarity
  - Code
  - Documentation
- Modularity
  - Main script
  - Use "good" functions or separate programs

Figure 12-8. Maintaining Code AL321.0

Notes:

Maintenance of code is at least as important as its creation. These are some issues that you may like to consider to ensure that your script can be maintained by others.

Good Functions

- To write functions that are reliable and easy to maintain:
  – Avoid altering global variables inside a function
  – Define and export functions only when necessary
  – Do not change the working directory inside a function
  – Tidy up local temporary files

Figure 12-9. Good Functions AL321.0

Notes:

Remember that functions run in the same environment as the caller, so *$$* is the same for the function and its calling shell.

Setting traps inside a function will not work with early versions of the Korn shell, so think about portability before using traps in a function.

The answer to the question is: because any changes to the current directory remain in force once the function completes or returns.

Performance Issues for Shell Scripts

- If performance is an issue
  - Do not guess
  - Measure
- Performance of a script means two areas:
  - That of the shell
  - That of the script
- Remember that you should work in this order
  - Make it work
  - Make it robust
  - Make it more efficient/faster

Figure 12-10. Performance Issues for Shell Scripts AL321.0

Notes:

If you suspect performance is an issue, then get some measurements.

When tuning a script, it is more usual to make it robust before worrying about whether it needs to be faster.

Timing Commands

- To report the elapsed, user and system time for a command or pipeline, use **time** in the Korn or Bash shells:
  - A reserved word (not a command)
  - Output is to standard error
  - Input or output redirection applies to the commands under test only
  - Return value is that of the commands under test

```
$ time find / -name 'unix*' -print|sort
/unix

/usr/lib/unixtomh                    output from find
real        0m25.51s                 wall clock time
user        0m1.56s
sys         0m11.01s
$ _
```


Figure 12-11. Timing Commands AL321.0

Notes:

The operating system also has a *time* command (*/bin/time*). It only reports in tenths of a second, and cannot handle pipelines. There is also a *timex* operating system command that uses the *sar*, *vmstat*, or *iostat* utilities to monitor a single command.

It would be a good idea to run the same command, with *time*, and take averages. Simply running the test once may not give you a true reading of “how long” the command(s) take.

Times for Shells

- The **times** command displays how much time your current shell and all its subshells have consumed:

```
$ times
0m0.99s  0m15.37s
0m8.61s  0m33.21s
```

  – User and system timings given in hundredths of a second
  – First line for the current shell
  – Second line for the subshells

Figure 12-12. Times for Shells AL321.0

Notes:

The *times* command returns 0 (true) always.

Shell Performance

- To increase the startup speed of a new shell:
  - Keep your history file (**.sh_history**) small
  - Minimize the size of any **$ENV** file
  - Use **autoload** with your functions (ksh)
  - Use **FPATH** with your functions
  - Use **set -o nolog** to prevent function definitions being logged in your history (ksh)
  - Use **tracked aliases** or **hash**es
  - Try to use an **alias** in place of a simple function
  - Set **MAILCHECK** greater than the 600 second default
  - In bash, use brace expansion, for example:
    mkdir ../release_{src,doc}

Figure 12-13. Shell Performance AL321.0

Notes:

Keeping the history small reduces the shell startup speed because it is read when the script starts. The file pointed to by the *ENV* variable is read for each Korn shell invocation.

Setting *MAILCHECK* to 0 causes the shell to check for new mail at every new prompt!

Bash has an expansion type facility called brace expansion. As you can see, it can generate any string. The example above would create two directories, release_src and release_doc in the parent directory. For brace expansion to be performed, there must be at least a matched pair of braces containing at least one comma.

Shell Script Performance

Tips for faster performance shell scripts:

- Shell built-in commands run faster than UNIX built-ins
- Avoid command substitution where you can use `${ }` parameter expansions, let or pattern matching
- Note `$(< file)` is faster than `$(cat file)`
- Use multiple arguments rather than separate commands – for example, `typeset -i a=3 b=4`
- Use `set -f` or `set -o noglob` if not using pathname metacharacters
- Use `{ }` grouping that is faster than `( )`
- Apply I/O re-directions to the whole of a loop syntax
- Set the `integer` attribute for suitable variables and don't use `$` for them with arithmetic expressions

Figure 12-14. Shell Script Performance AL321.0

Notes:

Make sure that your *PATH* is correctly set to prevent long search times for AIX commands. A tracked alias (see Unit 7) may also be helpful to reduce command search time. There is a table of Korn shell built-in commands in Unit 7 also.

General programming techniques can also bring about performance benefits. Move loop invariants to before the loop if you have a fixed command inside a loop you are repeating it many times without reason. Vary loop increments or the order of nesting; quite a bit of optimization relies on this kind of trick, for example, the obvious way to perform matrix multiplication is not the fastest!

Good Rules to Follow

1. Documentation
2. Make backups
3. Try three times
4. Do not overlook the obvious
5. Try it, it might work
6. Never say never, always avoid always
7. There is usually another way to do it

Figure 12-15. Good Rules To Follow AL321.0

Notes:

1) Documentation:	Comment, comment, comment.
2) Make backups:	Every good user has a good backup... right?
3) Try three times:	Then get help, whether it be another person, a reference *manual*, or another set of eyes. Don't frustrate yourself too much, you'll go crazy!
4) Don't overlook the obvious:	The easiest soluti<on to implement is the easiest to overlook.
5) Try it, it might work:	Just be sure of Rule Number 2.
6) Never say never, always avoid always:	Either one will come back to haunt you.
7) There's usually another way to do it:	Every situation can, and will, be different. Use what works well for you.

Checkpoint

1. What allows you to document your program for future reference?

2. Why is it a good idea to plan and design before you code?

3. Which statement is faster and why?
   `$(< data.file)` or `$(cat data.file)`

4. What set options can help in debugging a script?

Figure 12-16. Checkpoint AL321.0

Notes:

Write down your answers here:

1.

2.

3.

4.

Unit Summary

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary

Figure 12-17. Summary AL321.0

Notes:

Course Summary

- Basic concepts
- Shell variables and parameters
- Exit status, return codes and traps
- Programming constructs – flow control
- Shell commands and features
- Arithmetic in shells
- Shell types and functions
- Regular expressions and text selection
- Productivity using *sed* and *awk*
- Summary – good practice, debugging, performance

Figure 12-18. Course Summary AL321.0

Notes:

HAPPY SCRIPTING!

Appendix A. Utilities for Personal Productivity - Optional

What this unit is about

This unit looks briefly at three utilities to help improve productivity - tar, at and crontab.

What you should be able to do

After completing the unit, you should be able to:

- Make use of tar archive
- Be able to schedule scripts for execution at a later date

How you will check your progress

Accountability:

- Checkpoint questions
- Hands-on exercises

Unit Objectives

After completing this unit, you should be able to:

- Use the archive utility: tar
- Manipulate when your work gets done: at and crontab

Figure A-1. Unit Objectives AL321.0

Notes:

The tar Utility

This is an archive/backup command
Historically used tape but now any device

- default to /dev/rmt0

Syntax: `tar options pathname(s)`

Figure A-2. The tar Utility AL321.0

Notes:

The *tar* utility is very useful for temporary archives and backups. It was originally written to output to a tape device but is now used for virtually any storage device. For AIX the normal default is */dev/rmt0* but as you will see this can be changed by a command line option.

tar Options

- Options are of two types
  – Required
  – Optional

- Should be specified using a leading hyphen

- Required options are one of
  – c - create an archive
  – x - extract file(s) from archive
  – t - list (tell) what is in archive

- Other (optional) options are
  – f - used to specify other than default device
  – v - verbose (usually with t or x)
  – m - restore/keep modification times

Figure A-3. tar Options AL321.0

Notes:

tar options are in two groups — required and optional. The original utility did not conform to the normal syntax for parameters and options. Some old scripts using tar may be seen without a leading hyphen (-) before the options. Normal modern practice is to use the correct option syntax.

tar options are many and use of the AIX documentation and/or the man pages may be helpful. As the syntax suggests, there must be a *required* option present. The most common "optional" options are *-f* and *-v*. For example, to read an archive from the default device:

```
$ tar -tv
-rw-r--r-- phil/office     527 2000-02-01 17:13:09 getopts.ksh
-rwxr-xr-x phil/office      50 2000-07-06 13:25:26 group1.ksh
-rwxr-xr-x phil/office      55 2000-07-06 13:25:26 group2.ksh
-rwxr-xr-x phil/office     195 2000-07-06 13:25:26 if-then-elif.ksh
-rwxr-xr-x phil/office     123 2000-07-06 13:25:26 if-then-else.ksh
$ _
```


Notice that using *v* gives the equivalent of a long listing of a directory.

Typically the *-f* option is used to specify a tar file, often called a *tarfile*. For example:

```
$ tar -cf au23.tar examples
$ _
```

creates a tarball of the directory examples.

tar examples:

To back up your home directory relatively:

```
cd $HOME
tar -cvf /dev/fd0 .
```

To back up your home directory with a full path:

```
tar -cvf /dev/fd0 $HOME
```

To restore from the floppy

```
tar -xvf /dev/fd0
```

To get a listing of the files backed up on floppy

```
tar -tvf /dev/fd0
```

tar Pathnames

tar takes a pathname as one of its parameters

Full pathnames mean that restores (extracts) will be to original directory

Relative pathnames mean that restores may be to any part of filesystem

tar may be used to do recursive copies of data from one directory to another

```
$ cd fromdir; tar cf - . | (cd todir;\ >tar xf -)
```


Figure A-4. tar Pathnames AL321.0

Notes:

Since a pathname is involved it can be either a full or relative path. With tar, a full pathname will mean that files/directories extracted will be to the original path.

For that reason, relative pathnames are usually preferred for backups or archives. Choose carefully if you think that full paths are necessary.

Working in Absentia

- You can submit jobs for execution later
- AIX provides two useful utilities
  - at
  - crontab
- Access to these facilities is controlled by the system administrator

Figure A-5. Working in Absentia AL321.0

Notes:

Suppose you want to process some material but can wait (for example, overnight). The AIX utilities *at* and *crontab* (with the *cron* daemon) will help you.

It is possible that a tightly controlled system will not allow you to use these facilities until expressly enabled by the system administrator.

The at command

at submits a set of commands (a job) for later execution

Syntax: `at [-r|-l] time`

Commands are read from stdin

`time` can be specified as absolute or relative

- The time may include a date

Options include

`-l` list your at jobs

`-r` remove your *at* job(s)

at uses mail to send the stdin and stderr output (unless redirected)

System administrator determines who may use *at*

Figure A-6. The at Command AL321.0

Notes:

The set of commands (or script) submitted by *at* becomes an *at job*. This is not the same as a job in the Korn Shell.

The *time* syntax can be absolute as in 2200 or relative to some other time. The time specification can also include a date if required. The important point is that the “job” only executes once.

Note that the script (the set of commands) are copied to a spool area. This means that even if the script is subsequently edited, the changes are not made to the submitted script.

at Usage and Examples

Here are some examples (commands excluded)

at 2100
at 10pm
at 4am
at 9am tomorrow
at 10:30 Jul 3
at now + 2 hours
at now + 2 days
at now + 1 year

Figure A-7. at Usage and Examples AL321.0

Notes:

There are many different formats that you can use to specify the time. The use of *now* and *tomorrow* are useful.

The crontab Command

This command is like *at* but for regular "jobs"

Syntax: `crontab [-e | -l | -r] [job-file]`

The commands executed are in job-file (or from stdin)

The options allow you to edit, list or remove your crontab file

System administrator determines who may use *cron*

cron will mail the output of the command to crontab owner

Figure A-8. The crontab Command AL321.0

Notes:

crontab allows you to specify both date/time and frequency of a particular “job”. The crontab file has a particular format (you will see this next). To create an entry in your crontab, use

```
$ crontab job-file
```

The system daemon *cron* examines crontab files in the spool area every minute and loads any changes. Using crontab to edit your crontab entries is the best way to ensure that cron is informed of any updates.

Like the at command, a system administrator controls which users have access to crontab facilities.

crontab File Format

cron needs crontab files in a particular format

Each line has time(s)/date(s) and the command to run

Format of each line is a set of fields

- minute (0-59)
- hour (0-23)
- day (1-31)
- month (1-12)
- day of week (0-6, 0 = Sunday)

Each of the first five fields may be

- a number
- a comma separated number list (1,3,4,13)
- a range (4-9)
- an asterisk (*)

Sixth field contains the command(s) executed (a % means a newline)

Figure A-9. crontab File Format AL321.0

Notes:

Each of the six crontab fields are separated by whitespace, usually a space or tab character.

Unit Summary

Having completed this unit, you should be able to:

- Archiving using tar
- Batching commands for later execution
  - The tar command for backing up
  - Using at
  - Regular or repeated processes using cron tab

Figure A-10. Unit Summary AL321.0

Notes:

Each of the six cron tab fields are separated by whitespace, usually a space or tab character.

Appendix B. vi Reference

Overview of Operations

Initially, when you enter a command you are in input mode. To edit, the user enters control mode by typing *ESC* and moves the cursor to the point needing correction and then inserts or deletes characters or words as needed.

Most control commands accept an optional repeat Count prior to the command.

When in *vi mode* on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater and it contains any control characters or less than one second has elapsed since the prompt was printed.

The *ESC* character terminates canonical processing for the remainder of the command and the user can then modify the command line.

This scheme has the advantages of canonical processing with the type-ahead echoing of *raw mode*.

If the option *viraw* is also set, the terminal will always have canonical processing disabled.

This mode is implicit for systems that do not support two alternate end of line delimiters, and might be helpful for certain terminals.

vi Input Edit Commands (by default the editor is in input mode)

ERASE	(User-defined erase character as defined by the stty command, usually Ctrl-h or #) Deletes previous character.
Ctrl-w	Deletes the previous blank separated word.
Ctrl-v	Escapes the next character.
Ctrl-v	Editing characters, the user's ERASE or KILL characters can be entered in a command line or in a search string if preceded by a Ctrl-v
Ctrl-V	The Ctrl-V removes the next character's editing features (if any).
\	Escapes the next ERASE or KILL character.

Motion Edit Commands

l	Moves the cursor forward (right) one character.
w	Moves the cursor forward one alphanumeric word.
W	Moves the cursor to the beginning of the next word that follows a blank.
e	Moves the cursor to end of the current word.
E	Moves the cursor to end of the current blank delimited word.
h	Moves the cursor backward (left) one character.
b	Moves the cursor backward one word.
B	Moves the cursor to the previous blank separated word.
		Moves the cursor to the column specified by the Count parameter.
fc	Finds the next character c in the current line.

Fc	Finds the previous character c in the current line.
tc	Equivalent to f followed by h.
Tc	Equivalent to F followed by l.
;	Repeats Count times, the last single character find command.
0	Moves the cursor to start of line.
$	Moves the cursor to end of line.
^	Moves the cursor to start of line.

Text Modification Edit Commands

A	Appends text to the end of the line.
C	Deletes the current character through to the end of line and enters input mode.
d	Deletes the current character through to the end of line.
i	Enters the input mode and inserts text before the current character.
I	Inserts text before the beginning of the line.
P	Places the previous text modification before the cursor.
p	Places the previous text modification after the cursor.
R	Enters the input mode and types over the characters on the screen.
rc	Replaces the number of characters specified by the Count parameter, starting at the current cursor position, with the character(s) specified by c
x	Deletes the current character.
X	Deletes the preceding character.
.	Repeats the previous text modification command.
~	Inverts the case of the number of characters specified by the Count parameter, starting at the current cursor positions, and advances the cursor.

Search Edit Commands (these commands access your command history)

k	Fetches the previous command.
j	Moves forward through command list.
G	Fetches the command whose number is specified by the Count parameter that should precede it.
/String	Searches backward through history for a previous command containing the specified String. String is terminated by a RETURN or new-line character. If the specified string is preceded by a caret (^), the matched line must begin with String. If String is null, the previous string will be used.
?String	Same as / except that the search is in the forward direction.
n	Searches for the next match of the last pattern to / or ? commands.
N	Searches for the next match of the last pattern to / or ?, but in the opposite direction. Searches history for the String entered by the previous / command.

Other Edit Commands

y	Yanks the current character through the character to which Motion would move the cursor and puts them into the delete buffer. The text and cursor are unchanged.
Y	Yanks from the current position to the end of the line. Equivalent to y$.
u	Undo the last text modifying command.
U	Undo all the text modifying commands performed on the line.
e	Count in the input buffer. If Count is omitted, then the current line is used.

Features of “vi” with “set -o vi” only

\	Filename completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a / is appended if the file is a directory and a space is appended if the file is not a directory.
*	Appends an asterisk to the current word and attempts filename generation. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.
=	Lists the file names that match the current word as if an asterisk were appended to it.
_	(Underscore) Causes the Count word of the previous command to be appended and input mode entered. The last word is used if Count is omitted.
@Letter	Searches the alias list for an alias named Letter. If an alias of this name is defined, its value is placed into the input queue for processing.
#	Sends the line after inserting a # in front of the line. Useful for causing the current line to be inserted in the history without being executed.
Ctrl-c	Terminates the set -o vi edit
Ctrl-j	(New line) Executes the current line, regardless of the mode.
Ctrl-l	Line feeds and prints the current line. Has effect only in control mode.
Ctrl-m	(Return) Executes the current line, regardless of the mode.

Appendix C. Checkpoint Solutions

Unit 1 - Basic shell Concepts

1. What type of file is **/dev/tty3**?

Correct Answer:

/dev/tty3 is a special device file, representing a terminal.

2. How could we find out a file type?

Correct Answer:

Use the *file* command to identify a file type.

3. How can we get .kshrc to run in an explicit Korn shell?

Correct Answer:

export ENV=”$HOME/.kshrc”.

4. How can we specify the first character in a file name to be uppercase?

Correct Answer:

[[:upper:]]* or [A-Z]*.

5. How can we ignore error messages from a command?

Correct Answer:

command ... 2>/dev/null.

6. How do you make the normal output of a command appear as error output?

Correct Answer:

command ... 1>&2.

7. How can we group commands, in order to re-direct the standard output from all of them?

Correct Answer:

Use braces, or curly brackets, to surround the group and then do the redirection on the closing brace.

8. What will **kill 1** do?

Correct Answer:

Nothing. kill %1 will kill your job no.1, but kill 1 will attempt to kill process id 1, which is init, the parent of all other process. Even root cannot kill init.

9. If you have submitted a job to run in foreground, how could you move it to background?

Correct Answer:

First suspend the job with <Ctrl>-z, and then use the bg command to move it to the background.

10. How would you set up a command line recall facility?

Correct Answer:

set -o vi.

Unit 2 - Variables

1. How could we use positional parameter 3 in a shell script?

Correct Answer:

$3 or (better) ${3}.

2. Which variable contains the number of positional parameters?

Correct Answer:

$# or ${#}.

3. How can we change the value of a variable set in a different process?

Correct Answer:

This can't be done. A subprocess can only change a copy of an exported variable supplied by its parent process.

4. What is the variable ***IFS***?

Correct Answer:

Internal Field Separator used to read statements, and many other commands. It normally contains a space character, followed by a tab character, followed by a newline character.

5. How can we reset ***PS1*** to show the current directory?

Correct Answer:

export PS1='${PWD} $ '.

6. By setting a variable, how can we have a command recall facility?

Correct Answer:

set EDITOR or VISUAL to vi, emacs, or gmacs, and export it.

Unit 3 - Return Codes and Traps

1. How can you tell whether a command you have just entered was successful?

Correct Answer:

echo $? or print $?

2. How can you test if file *datafile* is non-empty?

Correct Answer:

test -s datafile or

[-s datafile] or

[[-s datafile]]

3. How can you check if you have been logged on for more than 20 minutes, and if so, print out a suitable message?

Correct Answer:

test “$SECONDS” -ge 1200 && echo Have a rest, $USER

4. How could you log off, using the kill command?

Correct Answer:

kill -9 $$ or kill $$

(The -9 is not usually necessary, unless a trap has been set.)

5. If you are a DBA is this a desirable command to terminate the <oracle_server>? **kill -KILL <oracle_server>**

Correct Answer:

Probably not — but at least you are the DBA and can clean up the situation.

6. What does this command do? **trap echo you did <Ctrl-c> 2**

Correct Answer:

Nothing! You get an error message indicating invalid syntax. It tries to identify the word 'you' as a signal. (It converts it to uppercase too). Single quotes need to be put around the echo and its arguments: trap 'echo “you did <cntrl-c>”' INT

7. How could you get <Ctrl-c> to log you off?

Correct Answer:

trap 'exit' 2.

Note: In this case, the quotes are not necessary, discipline yourself to use them anyway.

Unit 4 - Flow Control

1. What is wrong with this fragment of shell script?

```
if [ "$x" -eq 5 ]
then
      echo $x
elif [ "$x" -eq 3 ]
else
      echo "x is only 3"
      exit
fi
```

Correct Answer:

There must be a then statement after the elif.

2. What is the fundamental difference between a **while** and an **until** construct?

Correct Answer:

While statements assume “true”, until statements assume “false”.

3. How could we write an endless loop?

Correct Answer:

while true

4. What syntax would we use to perform a loop a finite number of times, resetting an identifier each time?

Correct Answer:

For identifier in word1 word2 word3 ...

Also for ((initialize, test, increment))

5. Which construct is best suited to allow conditional processing, based on pattern matching?

Correct Answer:

case $identifier in

6. What would the following lines produce?

```
select word in To be or not to be
do
        :
done
```

Correct Answer:

As follows:

1) To
2) be
3) or
4) not
5) to
6) be
#?

7. Which construct is best used within the previous **do-done**? block?

Correct Answer:

case statement

8. How can we terminate one iteration of a loop and commence the next?

Correct Answer:

Continue

9. How can we abruptly terminate all iterations of a loop but continue further processing in a shell script?

Correct Answer:

break

Unit 5 - Shell Commands

1. Without using redirection, how could we print information to file descriptor 2?

Correct Answer:

Use -u2 option to the print command.

2. What is wrong with the following command?
   `read speed?"mph" distance?"miles"`

Correct Answer:

read speed? "Enter MPH and DISTANCE" miles.

3. What **getopts** statement would allow you to process options **p**, and **a**, with option **t** expecting an associated value?

Correct Answer:

Specify a : after the t option getopts pat: varname

4. In the Bash shell, print is not built-in. What is the built-in command in bash that performs similarly to Korn's print?

Correct Answer:

The echo command

5. Which **set** option disables metacharacter pathname expansion?

Correct Answer:

set -o noglob or set -f

6. Which **set** options would be most useful in helping to debug a shell script?

Correct Answer:

You can do this by either using the full name options or the single letters.

set -o verbose or set -o xtrace or set -vx.

Unit 6 - Arithmetic

1. Multiply together variables **a** and **b**, using **expr**.

Correct Answer:

expr $a * $b

2. Use **expr** to multiply variable **a** by the sum of **b** and **c**.

Correct Answer:

expr $a * \($b + $c \)

3. Set variable **hex** to contain the hexadecimal value **7c**.

Correct Answer:

hex=16#7c

4. Write a **let** statement to test whether variable **a** is smaller than variable **b**.

Correct Answer:

((a<b)) or let “a < b”

5. Define a variable **num** as numeric only.

Correct Answer:

integer num

6. Increment a numeric variable **numvar**, by three.

Correct Answer:

Assuming the variable has been defined as an integer, we can use an implicit list:

numvar=numvar+3

Otherwise,

((numvar=numvar+3)) or let numvar=numvar+3

((numvar += 3)) or let numvar += 3

7. How would you calculate 6/7 to 6 decimal places?

Correct Answer:

echo “scale=6; 6/7”| bc

or

echo “scale=6 \n 6/7”| bc

answer is 0.857142

8. How would you calculate the square root of 8541976320?

Correct Answer:

echo “sqrt(8541976320)” | bc -l

answer is 92422.81276827707541375356 -- OK, so who cares about this number? Well, if anyone is still awake, that is the only number that uses all 10 digits alphabetically.

Unit 7 - Shell Types, Commands, and Functions

1. How is an array defined?

Correct Answer:

For a new array, we can use: set -A arrayname (values) or set +A arrayname (values).

Or we can simply assign a value to any single element arrayname[17]=99.

2. How do we refer to array elements?

Correct Answer:

By using braces and square brackets:

${arrayname[99]} or we can simply assign a value to any single element.

3. How could we set a variable **users**, to contain the number of users logged onto the system?

Correct Answer:

users=$(who I wc -l) or users=`who I wc -l`

4. How would we write a function to check the readability of a file?

Correct Answer:

```
function caniread
{
if [ -r “$1” ]
then
   echo yes
   return 0
```

```
else
   echo no
   return 1
fi
}
```

5. How do we print out the first and last positional parameter?

Correct Answer:

eval print $1 '$'{$#}

6. How do we define local variables within a function?

Correct Answer:

With the integer or typeset commands.

7. How can we list which functions are defined?

Correct Answer:

typeset +f (-f option to list the function definitions)

8. Which command would allow you to load a library of functions?

Correct Answer:

The autoload or typeset -fu command

9. How could we create an alias to show how many minutes have elapsed since the current shell began?

Correct Answer:

alias mins='echo $(expr $SECONDS / 60)'

Unit 8 - More on shell Variables

1. What happens when the variable **TMOUT** is set and you enter the following? **TMOUT=${TMOUT:-60}**

Correct Answer:

Nothing, if TMOUT already has a value, otherwise TMOUT is given the value 60.

2. What would your prompt say if you were in your **bin** directory and you entered this: **PS1='${PWD#$HOME/} $'**.

Correct Answer:

Your prompt would read: bin $.

3. How could you find out the number of characters in the variable HOME?

Correct Answer:

Use the # operator; print ${#HOME}.

Unit 9 - Regular Expressions and Text Selection Utilities

1. What regular expression can you use to select surnames?

Correct Answer:

^[A-Z][a-z]*[^a-z]

2. What regular expression can you use to select text with repeated characters in the surname?

Correct Answer:

^.*\(.\)\1.*,

3. What command can you use to select lines in phone.list with four character first names?

Correct Answer:

grep ', [A-Z][a-z]\{3\}[^a-z]' phone.list

4. How could you count the number of processes whose PIDs are in the range 1000-9999?

Correct Answer:

ps -ef | grep '^[a-z]*[0-9]\{4\}'\

'[^0-9]' | wc -l

5. How would you convert spaces to a tab in phone.list?

Correct Answer:

Use the command

tr “ “ “\t” <phone.list >phone.list.nospaces

6. What would this next command accomplish? **cut -d: -f1,3,4 /etc/passwd**

Correct Answer:

This will display the username, userid, and groupid from /etc/passwd file

7. Using the **paste** command, output the /etc/passwd file so that each line of information is separated by a tab and so that the fifth, sixth and seventh fields are on a separate line from the others. (Hint: make each field a line.)

Correct Answer:

tr “:” “\n” </etc/passwd | paste -s -d”\t\t\t\n\t\t\n” -

Unit 10 - The sed Utility

1. Write a command line script that displays a **ps -ef** with your username as the owner of *init*.

Correct Answer:

ps -ef | grep init | sed 's/root/teamXX/'

2. How can I make phone.list appear double spaced?

Correct Answer:

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

3. Cat out the sulog (located in /var/adm/sulog) and change all “+”s to the word successful and all “ - “ to the word unsuccessful using sed.

Correct Answer:

```
cat /var/adm/sulog|sed ‘s/+/successful/
s/ - /unsuccessful/’
```

4. Using sed, insert “#!/usr/bin/ksh” as the first line of a program called program1 and store in program2.

Correct Answer:

```
sed ‘1i\
#usr/bin/ksh’program1>program2
```

Unit 11 - The AWK Program

1. With **awk**, what happens if I don't supply a pattern?

Correct Answer:

The action is applied to each and every line.

2. With **awk**, what happens if I don't supply the action?

Correct Answer:

The pattern is applied and matches will display to STDOUT.

3. **awk** causes the **-f** option to read instructions from a default line.

Correct Answer:

No, the -f tells awk to read instructions from a named file, for example,

awk -f check.sum phone.list.

4. **awk** must have both the **BEGIN** and **END** statements.

Correct Answer:

No, neither is necessary.

5. Using awk, have the output from the dg command only show the % used and indent point.

Correct Answer:

df | awk '{print $4, $7}'

Unit 12 - Good Practices and Review

1. What allows you to document your program for future reference?

Correct Answer:

Comments, #

2. Why is it a good idea to plan and design before you code?

Correct Answer:

It will help you to know when you are finished.

3. Which statement is faster and why? $(< data.file) or $(cat data.file)

Correct Answer:

$(< data.file) because < does not create a new process

4. What set options can help in debugging a script?

Correct Answer:

verbose, xtrace, and noexec

Bibliography

ISBN 0-13-460866-6 — *UNIX shells by Example*, Ellie Quiqley, Prentice Hall, 1997

ISBN 0-201-56324-X — *Korn shell Programming Tutorial*, Barry Rosenberg, Addison-Wesley, 1991

SR28-5706-00 — *The New Korn shell Command and Programming Language*, M. I. Bolsky & D. G. Korn, Prentice Hall, 1995

SR28-4965-00 — *UNIX Power Tools*, Peek, O'Reilly, and Loukides, O'Reilly & Associates, Inc., 1994

SR28-4856-01 — *Essential System Administration*, Aeleen Frisch, O'Reilly & Associates, Inc., 1995

SR285268-00- — *Learning the Korn shell 2nd Edition*, Bill Rosenblatt and Arnold Robbins, O'Reilly & Associates, Inc., 2002

ISBN 1-53592-001-5 — *UNIX in a Nutshell*, Daniel Gilly, O'Reilly & Associates, Inc., 1994

ISBN 1-56592-225-5 — *sed & awk*, Dale Dougherty & Arnold Robbins, O'Reilly &Associates, Inc., 2nd. Edition, 1997.

Learning the bash Shell 2nd edition, Cameron Newham & Bill Rosenblatt, O'Reilly 1998