Advanced Bash–Scripting HOWTO

A guide to shell scripting, using Bash

Mendel Cooper

thegrendel@theriver.com

This is a major update on version 0.2. –– more bugs swatted, plus much additional material and example scripts added. This project has now reached the proportions of an entire book. See `NEWS` for a revision history.

This document is both a tutorial and a reference on shell scripting with Bash. It assumes no previous knowledge of scripting or programming, but progresses rapidly toward an intermediate/advanced level of instruction. The exercises and heavily–commented examples invite active reader participation. Still, it is a work in progress. The intention is to add much supplementary material in future updates to this HOWTO, so that it will gradually evolve into an LDP "guide", i.e., a complete book.

The latest version of this document, as an archived "tarball" including both the SGML source and rendered HTML, may be downloaded here from the author's home site.

Table of Contents

Table of Contents

Chapter 1. Why Shell Programming?

The shell is a command interpreter. It is the insulating layer between the operating system kernel and the user. Yet, it is also a fairly powerful programming language. A shell program, called a *script* , is an easy−to−use tool for building applications by "gluing" together system calls, tools, utilities, and compiled binaries. Virtually the entire repertoire of UNIX commands, utilities, and tools is available for invocation by a shell script. If that were not enough, internal shell commands, such as testing and loop constructs, give additional power and flexibility to scripts. Shell scripts lend themselves exceptionally well to to administrative system tasks and other routine repetitive jobs not requiring the bells and whistles of a full−blown tightly structured programming language.

A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in `/etc/rc.d` to restore the system configuration and set up services. A detailed understanding of these scripts is important for analyzing the behavior of a system, and possibly modifying it.

Writing shell scripts is not hard to learn, since the scripts can be built in bite−sized sections and there is only a fairly small set of shell−specific operators and options to learn. The syntax is simple and straightforward, similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" to learn. Most short scripts work right the first time, and debugging even the longer ones is straightforward.

A shell script is a "quick and dirty" method of prototyping a complex application. Getting even a limited subset of the functionality to work in a shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl.

Shell scripting hearkens back to the classical UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities. Many consider this a better, or at least more esthetically pleasing approach to problem solving than using one of the new generation of high powered all−in−one languages, such as Perl, which attempt to be all things to all people, but at the cost of forcing you to alter your thinking processes to fit the tool.

When not to use shell scripts

- resource−intensive tasks, especially where speed is a factor
- complex applications, where structured programming is a necessity
- mission−critical applications upon which you are betting the ranch, or the future of the company
- situations where security is important, where you need to protect against hacking
- project consists of subcomponents with interlocking dependencies
- extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line−by−line fashion)
- need to generate or manipulate graphics or GUIs
- need direct access to system hardware
- need port or socket I/O
- need to use libraries or interface with legacy code

If any of the above applies, consider a more powerful scripting language, perhaps Perl, Tcl, Python, or possibly a high−level compiled language such as C, C++, or Java. Even then, prototyping the application as a shell script might still be a useful development step.

We will be using Bash, an acronym for "Born−Again Shell" and a pun on Stephen Bourne's now classic Bourne Shell. Bash has become the de facto standard for shell scripting on all flavors of UNIX. Most of the principles dealt with in this document apply equally well to scripting with other shells, such as the Korn Shell, from which Bash derives some of its features, [1] and the C Shell and its variants. (Note that C Shell programming is not recommended due to certain inherent problems, as pointed out in a news group posting by Tom Christiansen in October of 1993).

The following is a tutorial in shell scripting. It relies heavily on examples to illustrate features of the shell. As far as possible, the example scripts have been tested, and some of them may actually be useful in real life. The reader should use the actual examples in the the source archive (`something−or−other.sh`), give them execute permission (**`chmod u+x scriptname`**), then run them to see what happens. Should the source archive not be available, then cut−and−paste from the HTML, pdf, or text rendered versions. Be aware that some of the scripts below introduce features before they are explained, and this may require the reader to temporarily skip ahead for enlightenment.

Unless otherwise noted, the author of this document wrote the example scripts that follow.

Chapter 2. Starting Off With a Sha−Bang

In the simplest case, a script is nothing more than a list of system commands stored in a file. At the very least, this saves the effort of retyping that particular sequence of commands each time it is invoked.

Example 2−1. cleanup: A script to clean up the log files in /var/log

```
# cleanup
# Run as root, of course.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Logs cleaned up."
```

There is nothing unusual here, just a set of commands that could just as easily be invoked one by one from the command line on the console or in an xterm. The advantages of placing the commands in a script go beyond not having to retype them time and again. The script can easily be modified, customized, or generalized for a particular application.

Example 2−2. cleanup: An enhanced and generalized version of above script.

```
#!/bin/bash
# cleanup, version 2
# Run as root, of course.

if [ -n $1 ]
# Test if command line argument present.
then
  lines=$1
else
  lines=50
  # default, if not specified on command line.
fi


cd /var/log
tail -$lines messages > mesg.temp
# Saves last section of message log file.
mv mesg.temp messages

# cat /dev/null > messages
# No longer needed, as the above method is safer.

cat /dev/null > wtmp
echo "Logs cleaned up."

exit 0
# A zero return value from the script upon exit
# indicates success to the shell.
```

Since you may not wish to wipe out the entire system log, this variant of the first script keeps the last section of the message log intact. You will constantly discover ways of refining previously written scripts for increased effectiveness.

The *sha−bang* (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two byte " magic number", a special marker that designates an executable shell script (**`man magic`** gives more info on this fascinating topic). Immediately following the *sha−bang* is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This enables the specific commands and directives embedded in the shell or program called.

```
#!/bin/sh
#!/bin/bash #!/bin/awk #!/usr/bin/perl #!/bin/sed
#!/usr/bin/tcl
```

Each of the above script header lines calls a different command interpreter, be it `/bin/sh`, the default shell (**bash** in a Linux system) or otherwise. Using **`#!/bin/sh`**, the default Bourne Shell in most commercial variants of UNIX, makes the script portable to non−Linux machines, though you may have to sacrifice a few bash−specific features (the script will conform to the POSIX **sh** standard).

Note that the path given at the "sha−bang" must be correct, otherwise an error message, usually Command not found will be the only result of running the script.

#! can be omitted if the script consists only of a set of generic system commands, using no internal shell directives. Example 2, above, requires the initial #!, since the variable assignment line, **`lines=50`**, uses a shell−specific construct. Note that **`#!/bin/sh`** invokes the default shell interpreter, which defaults to `/bin/bash` on a Linux machine.

> **Important:** This tutorial encourages a modular approach to constructing a script. Make note of and collect "boilerplate" code snippets that might be useful in future scripts. Eventually you can build a quite extensive library of nifty routines. As an example, the following script prolog tests whether the script has been invoked with the correct number of parameters.

```
if [ $# -ne Number_of_expected args ]
then
  echo "Usage: `basename $0` whatever"
  exit $WRONG_ARGS
fi
```

2.1. Invoking the script

Having written the script, you can invoke it by **`sh scriptname`**, or alternately **`bash scriptname`**. (Not recommended is using **`sh <scriptname`**, since this effectively disables reading from stdin within the script.) Much more convenient is to make the script itself directly executable by

Either:

> **`chmod 755 scriptname`** (gives everyone execute permission)

or

> **`chmod +x scriptname`** (gives everyone execute permission)
>
> **`chmod u+x scriptname`** (gives only the script owner execute permission)

In this case, you could try calling the script by **`./scriptname`**.

As a final step, after testing and debugging, you would likely want to move it to `/usr/local/bin` (as root, of course), to make the script available to yourself and all other users as a system−wide executable. The script could then be invoked by simply typing **scriptname [return]** from the command line.

2.2. Shell wrapper, self−executing script

A **sed** or **awk** script (see Appendix B) would normally be invoked from the command line by a **`sed -e`** ***`'commands'`*** or **`awk -e`** ***`'commands'`***. Embedding such a script in a bash script permits calling it more simply, and makes it "reusable". This also enables combining the functionality of **sed** and **awk**, for example piping the output of a set of **sed** commands to **awk**. As a saved executable file, you can then repeatedly invoke it in its original form or modified, without the inconvenience of retyping it on the command line.

Example 2−3. shell wrapper

```
#!/bin/bash

# This is a simple script
# that removes blank lines
# from a file.
# No argument checking.

# Same as
# sed -e '/^$/d $1' filename
# invoked from the command line.

sed -e /^$/d $1
# '^' is beginning of line,
# '$' is end,
# and 'd' is delete.
```

Example 2−4. A slightly more complex shell wrapper

```
#!/bin/bash

# "subst", a script that substitutes one pattern for
# another in a file,
# i.e., "subst Smith Jones letter.txt".

if [ $# -ne 3 ]
# Test number of arguments to script
# (always a good idea).
then
  echo "Usage: `basename $0` old-pattern new-pattern filename"
  exit 1
fi

old_pattern=$1
new_pattern=$2

if [ -f $3 ]
then
```

```
    file_name=$3
else
    echo "File \"$3\" does not exist."
    exit 2
fi

# Here is where the heavy work gets done.
sed -e "s/$old_pattern/$new_pattern/" $file_name
# 's' is, of course, the substitute command in sed,
# and /pattern/ invokes address matching.
# Read the literature on 'sed' for a more
# in-depth explanation.

exit 0
# Successful invocation of the script returns 0.
```

Example 2−5. A shell wrapper around an awk script

```
#!/bin/bash

# Adds up a specified column (of numbers) in the target file.

if [ $# -ne 2 ]
# Check for proper no. of command line args.
then
   echo "Usage: `basename $0` filename column-number"
   exit 1
fi

filename=$1
column_number=$2

# Passing shell variables to the awk part of the script is a bit tricky.
# See the awk documentation for more details.

# A multi-line awk script is invoked by   awk ' ..... '


# Begin awk script.
# -----------------------------
awk '

{ total += $'"${column_number}"'
}
END {
     print total
}

' $filename
# -----------------------------
# End awk script.


exit 0
```

For those scripts needing a single do−it−all tool, a Swiss army knife, there is Perl. Perl combines the capabilities of **sed** and **awk**, and throws in a large subset of **C**, to boot. It is modular and contains support for everything ranging from object−oriented programming up to and including the kitchen sink. Short Perl scripts can be effectively embedded in shell scripts, and there may even be some substance to the claim that Perl can totally replace shell scripting (though the author of this HOWTO remains skeptical).

Example 2−6. Perl embedded in a bash script

```
#!/bin/bash

# Some shell commands may precede the Perl script.

perl −e 'print "This is an embedded Perl script\n"'
# Like sed and awk, Perl also uses the "−e" option.

# Some shell commands may follow.

exit 0
```

Exercise. Write a shell script that performs a simple task.

Chapter 3. Tutorial / Reference

...there are dark corners in the Bourne shell, and people use all of them.

Chet Ramey

3.1. exit and exit status

The **exit** command may be used to terminate a script, just as in a C program. It can also return a value, which is available to the shell.

Every command returns an *exit status* (sometimes referred to as a *return status*). A successful command returns a 0, while an unsuccessful one returns a non−zero value that usually may be interpreted as an error code.

Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status. Within a script, an **`exit nn`** command may be used to deliver an `nn` exit status to the shell (`nn` must be a decimal number in the 0 − 255 range).

`$?` reads the exit status of script or function.

Example 3−1. exit / exit status

```
#!/bin/bash

echo hello
echo $?
# exit status 0 returned
# because command successful.

lskdf
# bad command
echo $?
# non-zero exit status returned.

echo

exit 143
# Will return 143 to shell.
# To verify this, type $? after script terminates.

# By convention, an 'exit 0' shows success,
# while a non-zero exit value indicates an error or anomalous condition.

# It is also appropriate for the script to use the exit status
# to communicate with other processes, as when in a pipe with other scripts.
```

3.2. Special characters used in shell scripts

#

Comments. Lines beginning with a # (with the exception of #!) are comments.

```
# This line is a comment.
```

Comments may also occur at the end of a command.

```
echo "A comment will follow." # Comment here.
```

Comments may also follow white space at the beginning of a line.

```
	# A tab precedes this comment.
```

Caution
A command may not follow after a comment on the same line. There is no method of terminating the comment, in order for "live code" to begin on the same line. Use a new line for the next command.

;

Command separator. Permits putting two or more commands on the same line.

```
echo hello; echo there
```

Note that the ; sometimes needs to be escaped (\).

.

"dot" command. Equivalent to source, explained further on (see Example 3−44).

:

null command. Exit status 0, alias for true

Endless loop:

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done
```

Placeholder in if/then test:

```
if condition
then :   # Do nothing and branch ahead
```

```
else
   take-some-action
fi
```

Provides a placeholder where a binary operation is expected, see Section 3.3.1.

```
: ${username=`whoami`}
# ${username=`whoami`}   without the leading : gives an error
```

Evaluate string of variables using "parameter substitution", see Example 3−6:

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
```

Prints error message if one or more of essential environmental variables not set.

()

command group.

```
(a=hello; echo $a)
```

> **Note:** A listing of commands within `parentheses` starts a subshell (see Section 3.16).

${}

Parameter substitution.

See Section 3.3 for more details.

{xxx,yyy,zzz,...}

Brace expansion.

```
grep Linux {file?.txt,*.list}
# Finds all instances of the work "Linux"
# in the files "fileA.txt", "file2.txt", "word.list", "vegetable.list", etc.
```

A command may act upon a comma−separated list of file specs within `braces`. Filename expansion (globbing) applies to the file specs between the braces.

Warning
No spaces allowed within the braces.

{}

Block of code. Also referred to as an "inline group", this construct, in effect, creates an anonymous function. Similar to a function, a code block permits isolation from the remainder of the script, with its own local variables visible only within the scope of the block.

The code block enclosed in braces may have I/O redirected to and from it. See Section 3.13 for a detailed discussion of I/O redirection.

Example 3−2. Code blocks and I/O redirection

```
#!/bin/bash

{
read fstab
} < /etc/fstab

echo "First line in /etc/fstab is:"
echo "$fstab"

exit 0
```

Example 3−3. Saving the results of a code block to a file

```
#!/bin/bash

#                rpm-check
#                ---------
# Queries an rpm file for description, listing, and whether it can be installed.
# Saves output to a file.
#
# This script illustrates using a code block.

NOARGS=1

if [ -z $1 ]
then
  echo "Usage: `basename $0` rpm-file"
  exit $NOARGS
fi

{
  echo
  echo "Archive Description:"
  rpm -qpi $1  #Query description.
  echo
  echo "Archive Listing:"
  rpm -qpl $1  #Query listing.
  echo
  rpm -i --test $1  #Query whether rpm file can be installed.
  if [ ! $? ]
  then
    echo "$1 can be installed."
  else
    echo "$1 cannot be installed."
  fi
  echo
} > $1.test  # Redirects output of everything in block to file.

echo "Results of rpm test in file $1.test"

# See rpm man page for explanation of options.

exit 0
```

/{}

file pathname. Mostly used in 'find' constructs.

> >& >> <

redirection.

`scriptname >filename` redirects the output of `scriptname` to file `filename`. Overwrite `filename` if it already exists.

`command >&2` redirects output of `command` to stderr.

`scriptname >>filename` appends the output of `scriptname` to file `filename`. If `filename` does not already exist, it will be created.

For a more detailed explanation, see Section 3.13.

<<

redirection used in "here document". See Section 3.24.

|

pipe. Passes the output of previous command to next one, or to shell. This is a method of chaining commands together.

```
echo ls -l | sh
```

passes the output of "ls −l" to the shell, with the same result as a simple "ls −l".

```
cat *.lst | sort | uniq
```

sorts the output of all the `.lst` files and deletes duplicate lines.

> **Note:** If one of the commands in the pipe aborts, this prematurely terminates execution of the pipe. Called a *broken pipe*, this condition sends a *SIGPIPE* signal. (See Section 3.26 for more detail on signals.)

>|

force redirection (even if `noclobber` environmental variable is in effect). This will forcibly overwrite an existing file.

−

redirection from/to stdin or stdout.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xvfp -)
# Move entire file tree from one directory to another
# [courtesy Alan Cox, a.cox@swansea.ac.uk]
#
# More elegant than, but equivalent to:
# cd source-directory
# tar cf - . | (cd ../target-directory; tar xzf -)
```

```
bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --uncompress tar file--    | --then pass it to "tar"--
```

```
# If "tar" has not been patched to handle "bunzip2",
# this needs to be done in two discrete steps, using a pipe.
# The purpose of the exercise is to unarchive "bzipped" kernel source.
```

Note that in this context the "−" is not itself a Bash operator, but rather an option recognized by certain UNIX utilities.

Where a filename is expected, redirects output to stdout (mostly seen with **`tar cf`**)

Example 3−4. Backup of all files changed in last day

```
#!/bin/bash

# Backs up all files in current directory
# modified within last 24 hours
# in a tarred and gzipped file.

if [ $# = 0 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi

tar cvf − `find . −mtime −1 −type f −print` > $1.tar
gzip $1.tar

exit 0
```

−

previous working directory. cd − changes to previous working directory. This uses the $OLDPWD environmental variable (see Section 3.7).

Caution
This is not to be confused with the "−" redirection operator just discussed. How Bash interprets the "−" depends on the context in which it appears.

~

home directory. ~*bozo* is bozo's home directory, and **ls ~bozo** lists the contents of it. ~/ is the current user's home directory, and **ls ~/** lists the contents of it.

White space

functions as a separator, separating commands or variables. White space consists of either spaces, tabs, blank lines, or any combination thereof. In some contexts, such as variable assignment, white space is not permitted, and results in a syntax error.

Blank lines

Blank lines have no effect on the action of a script, and are therefore useful for visually separating functional sections of the script.

3.3. Introduction to Variables and Parameters

Variables are at the heart of every programming and scripting language. They are used for arithmetic operations and manipulation of quantities, string parsing, and working in the abstract with symbols − tokens that represent something else. A variable is nothing more than a location or set of locations in computer memory that holds an item of data.

$

variable substitution. Let us carefully distinguish between the *name* of a variable and its *value*. If **`variable1`** is the name of a variable, then **`$variable1`** is a reference to its *value*, the data item it contains. The only time a variable appears "naked", without the $, is when declared or assigned (or when *exported*). Assignment may be with an = (as in *var1=27*), in a *`read`* statement, and at the head of a loop (*for var2 in 1 2 3*).

Enclosing a referenced value in double quotes (" ") does not interfere with variable substitution. This is called partial quoting, sometimes referred to as "weak quoting". Using single quotes (' ') causes the variable name to be used literally, and no substitution will take place. This is full quoting, sometimes referred to as "strong quoting".

Note that **`$variable`** is actually a simplified alternate form of **`${variable}`**. In contexts where the **`$variable`** syntax causes an error, the longer form may work (see Section 3.3.1 below).

Example 3−5. Variable assignment and substitution

```
#!/bin/bash

# Variables: assignment and substitution

a=37.5
hello=$a
# No space permitted on either side of = sign when initializing variables.

echo hello
# Not a reference.

echo $hello
echo ${hello} #Identical to above.

echo "$hello"
echo "${hello}"

echo '$hello'
# Variable referencing disabled by single quotes,
# because $ interpreted literally.

# Notice the effect of different types of quoting.

# −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

# It is permissible to set multiple variables on the same line,
# separated by white space. Careful, this may reduce legibility.

var1=variable1  var2=variable2  var3=variable3
```

```
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# --------------------------------------------------------------

echo; echo

numbers="one two three"
other_numbers="1 2 3"
# If whitespace within variables, then quotes necessary.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"
echo

echo "uninitialized variable = $uninitialized_variable"
# Uninitialized variable has null value (no value at all).

echo

exit 0
```

Warning
An uninitialized variable has a "null" value − no assigned value at all (not zero!). Using a variable before assigning a value to it will inevitably cause problems.

3.3.1. Parameter Substitution

${parameter}

Same as *$parameter*, i.e., value of the variable *parameter*.

May be used for concatenating variables with strings.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin  #Add /opt/bin to $PATH for duration of script.
echo "New \$PATH = $PATH"
```

${parameter-default}

If parameter not set, use default.

```
echo ${username-`whoami`}
# Echoes the result of `whoami`, but variable "username" is still unset.
```

Note: This is almost equivalent to *${parameter:-default}*. The extra : makes a difference only when *parameter* has been declared, but is null.

```
#!/bin/bash
```

```
username0=
echo "username0 = ${username0-`whoami`}"
# username0 has been declared, but is set to null.
# Will not echo.

echo "username1 = ${username1-`whoami`}"
# username1 has not been declared.
# Will echo.

username2=
echo "username2 = ${username2:-`whoami`}"
# username2 has been declared, but is set to null.
# Will echo because of :- rather than just - in condition test.

exit 0
```

${parameter=default}***, ***${parameter:=default}

If parameter not set, set it to default.

Both forms nearly equivalent. The : makes a difference only when *parameter* has been declared and is null, as above.

```
echo ${username=`whoami`}
# Variable "username" is now set to `whoami`.
```

${parameter+otherwise}***, ***${parameter:+otherwise}

If parameter set, use 'otherwise", else use null string.

Both forms nearly equivalent. The : makes a difference only when *parameter* has been declared and is null, as above.

${parameter?err_msg}***, ***${parameter:?err_msg}

If parameter set, use it, else print err_msg.

Both forms nearly equivalent. The : makes a difference only when *parameter* has been declared and is null, as above.

Example 3−6. Using param substitution and :

```
#!/bin/bash

# Let's check some of the system's environmental variables.
# If, for example, $USER, the name of the person
# at the console, is not set, the machine will not
# recognize you.

: ${HOSTNAME?} ${USER?} ${HOME} ${MAIL?}
  echo
  echo "Name of the machine is $HOSTNAME."
  echo "You are $USER."
  echo "Your home directory is $HOME."
  echo "Your mail INBOX is located in $MAIL."
```

```
  echo
  echo "If you are reading this message,"
  echo "critical environmental variables have been set."
  echo
  echo

# The ':' operator seems fairly error tolerant.
# This script works even if the '$' omitted in front of
# {HOSTNAME}, {USER?}, {HOME?}, and {MAIL?}. Why?

# -------------------------------------------------------

# The ${variablename?} construction can also check
# for variables set within the script.

ThisVariable=Value-of-ThisVariable
# Note, by the way, that string variables may be set
# to characters disallowed in their names.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".
echo
echo

# If ZZXy23AB has not been set...
: ${ZZXy23AB?}
# This will give you an error message and terminate.

echo "You will not see this message."

exit 0
```

Parameter substitution and/or expansion. The following expressions are the complement to the **match** `in` **expr** string operations (see Example 3−52). These particular ones are used mostly in parsing file path names.

${var#pattern}***, ***${var##pattern}

Strip off shortest/longest part of *pattern* if it matches the front end of *variable*.

${var%pattern}***, ***${var%%pattern}

Strip off shortest/longest part of *pattern* if it matches the back end of *variable*.

Version 2 of bash adds additional options.

Example 3−7. Renaming file extensions:

```
#!/bin/bash

#                 rfe
#                 ---

# Renaming file extensions.
#
#         rfe old_extension new_extension
#
# Example:
```

```
# To rename all *.gif files in working directory to *.jpg,
#          rfe gif jpg

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` old_file_suffix new_file_suffix"
  exit 1
fi

for filename in *.$1
# Traverse list of files ending with 1st argument.
do
  mv $filename ${filename%$1}$2
  # Strip off part of filename matching 1st argument,
  # then append 2nd argument.
done

exit 0
```

${var:pos}

Variable *var* expanded, starting from offset *pos*.

${var:pos:len}

Expansion to a max of *len* characters of variable *var*, from offset *pos*. See Example A−6 for an example of the creative use of this operator.

${var/patt/replacement}

First match of *patt*, within *var* replaced with *replacement*.

If *replacement* is omitted, then the first match of *patt* is replaced by *nothing*, that is, deleted.

${var//patt/replacement}

All matches of *patt*, within *var* replaced with *replacement*.

Similar to above, if *replacement* is omitted, then all occurrences *patt* are replaced by *nothing*, that is, deleted.

Example 3−8. Using pattern matching to parse arbitrary strings

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (with everything, up to and including first - stripped out) = $t"
t=${var1%*-*}
echo "var1 (with everything from the last - on stripped out) = $t"

echo
```

```
path_name=/home/bozo/ideas/thoughts.for.today
echo "path_name = $path_name"
t=${path_name##/*/}
# Same effect as   t=`basename $path_name`
echo "path_name, stripped of prefixes = $t"
t=${path_name%/*.*}
# Same effect as   t=`dirname $path_name`
echo "path_name, stripped of suffixes = $t"

echo

t=${path_name:11}
echo "$path_name, with first 11 chars stripped off = $t"
t=${path_name:11:5}
echo "$path_name, with first 11 chars stripped off, length 5 = $t"

echo

t=${path_name/bozo/clown}
echo "$path_name with \"bozo\" replaced  by \"clown\" = $t"
t=${path_name/today/}
echo "$path_name with \"today\" deleted = $t"
t=${path_name//o/O}
echo "$path_name with all o's capitalized = $t"
t=${path_name//o/}
echo "$path_name with all o's deleted = $t"

exit 0
```

3.4. Quoting

Quoting means just that, bracketing a string in quotes. This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning, such as the wild card character, *.)

When referencing a variable, it is generally advisable in enclose it in double quotes (" "). This preserves all special characters within the variable name, except $, ', and \. This allows referencing it, that is, replacing the variable with its value (see Example 3−5, above). Enclosing the arguments to an **echo** statement in double quotes is usually a good practice (and sometimes required, see Section 3.28).

Single quotes (' ') operate similarly to double quotes, but do not permit referencing variables, since the special meaning of $ is turned off. Within single quotes, *every* special character except ' gets interpreted literally. Consider single quotes ("full quoting") to be a stricter method of quoting than double quotes ("partial quoting").

Escaping is a method of quoting single characters. The escape (\) preceding a character will either toggle on or turn off a special meaning for that character, depending on context.

n

means newline

r

means return

\t

means tab

\v

means vertical tab

\b

means backspace

\a

means "alert" (beep or flash)

\0xx

translates to the octal ASCII equivalent of `0xx`

```
# Use the −e option with 'echo' to print these.
echo −e "\v\v\v\v"   # Prints 4 vertical tabs.
echo −e "\042"   # Prints " (quote, ASCII character 42).
```

\"

gives the quote its literal meaning

```
echo "Hello"                  # Hello
echo "\"Hello\", he said."    # "Hello", he said.
```

\$

gives the dollar sign its literal meaning (variable name following \$ will not be referenced)

```
echo "\$variable01"  # results in $variable01
```

\\

gives the backslash its literal meaning

```
echo "\\"  # results in \
```

The escape also provides a means of writing a multi−line command. Normally, each separate line constitutes a different command, but an escape at the end of a line *escapes the newline character*, and the command sequence continues onto the next line.

```
(cd /source/directory && tar cf − . ) | \
(cd /dest/directory && tar xvfp −)
# Repeating Alan Cox's directory tree copy command,
```

```
# but split into two lines for increased legibility.
```

3.5. Tests

The if/then construct tests whether a condition is true, and if so, executes one or more commands. Note that in this context, `0` (zero) will evaluate as true, as will a random string of alphanumerics. Puzzling out the logic of this is left as an exercise for the reader.

Example 3−9. What is truth?

```
#!/bin/bash

if [ 0 ]
#zero
then
  echo "0 is true."
else
  echo "0 is false."
fi

if [ ]
#NULL (empty condition)
then
  echo "NULL is true."
else
  echo "NULL is false."
fi

if [ xyz ]
#string
then
  echo "Random string is true."
else
  echo "Random string is false."
fi

if [ $xyz ]
#string
then
  echo "Undeclared variable is true."
else
  echo "Undeclared variable is false."
fi

exit 0
```

Exercise. Explain the behavior of Example 3−9, above.

```
if [ condition−true ]
then
   command 1
   command 2
   ...
else
   # Optional (may be left out if not needed).
   # Adds default code block executing if original condition tests false.
```

```
   command 3
   command 4
   ...
fi
```

Add a semicolon when 'if' and 'then' are on same line.

```
if [ -x filename ]; then
```

elif

This is a contraction for else if. The effect is to nest an inner if/then construction within an outer one.

```
if [ condition ]
then
   command
   command
   command
elif
# Same as else if
then
   command
   command
else
   default-command
fi
```

The **`test condition-true`** construct is the exact equivalent of **`if [condition-true ]`**. The left bracket [is, in fact, an alias for test. (The closing right bracket] in a test should not therefore be strictly necessary, however newer versions of bash detect it as a syntax error and complain.)

Example 3−10. Equivalence of [ ] and test

```
#!/bin/bash

echo


if test -z $1
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

# Both code blocks are functionally identical.

if [ -z $1 ]
# if [ -z $1
# also works, but outputs an error message.
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi
```

```
echo

exit 0
```

3.5.1. File test operators

Returns true if...

−e

file exists

−f

file is a regular file

−s

file is not zero size

−d

file is a directory

−b

file is a block device (floppy, cdrom, etc.)

−c

file is a character device (keyboard, modem, sound card, etc.)

−p

file is a pipe

−L

file is a symbolic link

−S

file is a socket

−r

file is readable (has read permission)

−w

file has write permission

−x

file has execute permission

−g

group−id flag set on file

−u

user−id flag set on file

−k

"sticky bit" set (if user does not own a directory that has the sticky bit set, she cannot delete files in it, not even files she owns)

−O

you are owner of file

−G

group−id of file same as yours

−t n

file descriptor *n* is open

This usually refers to *stdin*, *stdout*, and *stderr* (file descriptors 0 − 2).

f1 −nt f2

file `f1` is newer than `f2`

f1 −ot f2

file `f1` is older than `f2`

f1 −ef f2

files `f1` and `f2` are links to the same file

!

"not" −− reverses the sense of the tests above (returns true if condition absent).

Example 3−11. Tests, command chaining, redirection

```
#!/bin/bash

# This line is a comment.

filename=sys.log

if [ ! -f $filename ]
then
  touch $filename; echo "Creating file."
else
  cat /dev/null > $filename; echo "Cleaning out file."
fi

# Of course, /var/log/messages must have
# world read permission (644) for this to work.
tail /var/log/messages > $filename
echo "$filename contains tail end of system log."

exit 0
```

3.5.2. Comparison operators (binary)

integer comparison

−eq

is equal to (**`$a -eq $b`**)

−ne

is not equal to (**`$a -ne $b`**)

−gt

is greater than (**`$a -gt $b`**)

−ge

is greater than or equal to (**`$a -ge $b`**)

−lt

is less than (**`$a -lt $b`**)

−le

is less than or equal to (**`$a -le $b`**)

string comparison

=

is equal to (**`$a = $b`**)

!=

is not equal to (**`$a != $b`**)

\<

is less than, in ASCII alphabetical order (**`$a \< $b`**)

Note that the "<" needs to be escaped.

\>

is greater than, in ASCII alphabetical order (**`$a \> $b`**)

Note that the ">" needs to be escaped.

See Example 3−91 for an application of this comparison operator.

−z

string is "null", that is, has zero length

−n

string is not "null".

Caution
This test requires that the string be quoted within the test brackets. You may use **`! -z`** instead, or even just the string itself, without a test operator (see Example 3−13).

Example 3−12. arithmetic and string comparisons

```
#!/bin/bash

a=4
b=5

# Here a and b can be treated either as integers or strings.
# There is some blurring between the arithmetic and integer comparisons.
# Be careful.

if [ $a -ne $b ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo

if [ $a != $b ]
then
```

```
  echo "$a is not equal to $b."
  echo "(string comparison)"
fi

echo

exit 0
```

Example 3−13. testing whether a string is *null*

```
#!/bin/bash

# If a string has not been initialized, it has no defined value.
# This state is called "null" (not the same as zero).


if [ -n $string1 ]   # $string1 has not been declared or initialized.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
# Wrong result.
# Shows $string1 as not null, although it was not initialized.

echo

# Lets try it again.

if [ -n "$string1" ]  # This time, $string1 is quoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi

echo

if [ $string1 ]  # This time, $string1 stands naked.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
# This works fine.
# The [ ] test operator alone detects whether the string is null.

echo

string1=initialized

if [ $string1 ]  # This time, $string1 stands naked.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
# Again, gives correct result.
```

```
exit 0

# Thanks to Florian Wisser for pointing this out.
```

Example 3−14. zmost

```
#!/bin/bash

#View gzipped files with 'most'

NOARGS=1

if [ $# = 0 ]
# same effect as:  if [ -z $1 ]
then
  echo "Usage: `basename $0` filename" >&2
  # Error message to stderr.
  exit $NOARGS
  # Returns 1 as exit status of script
  # (error code)
fi

filename=$1

if [ ! -f $filename ]
then
  echo "File $filename not found!" >&2
  # Error message to stderr.
  exit 2
fi

if [ ${filename##*.} != "gz" ]
# Using bracket in variable substitution.
then
  echo "File $1 is not a gzipped file!"
  exit 3
fi

zcat $1 | most

exit 0

# Uses the file viewer 'most'
# (similar to 'less')
```

compound comparison

−a

logical and

`exp1 -a exp2` returns true if *both* exp1 and exp2 are true.

−o

logical or

exp1 −o exp2 returns true if either exp1 *or* exp2 are true.

These are simpler forms of the comparison operators `&&` and `||`, which require brackets to separate the target expressions.

Refer to Example 3−15 to see compound comparison operators in action.

3.6. Operations and Related Topics

3.6.1. Operations

=

All−purpose assignment operator, which works for both arithmetic and string assignments.

```
var=27
category=minerals
```

May also be used in a string comparison test.

```
if [ $string1 = $string2 ]
then
   command
fi
```

The following are normally used in combination with **expr** or **let**.

arithmetic operators

+

plus

−

minus

*

multiplication

/

division

%

modulo, or mod (returns the remainder of an integer division)

+=

"plus−equal" (increment variable by a constant)

`expr $var+=5` results in `var` being incremented by `5`.

`-=`

"minus−equal" (decrement variable by a constant)

`*=`

"times−equal" (multiply variable by a constant)

`expr $var*=4` results in `var` being multiplied by `4`.

`/=`

"slash−equal" (divide variable by a constant)

`%=`

"mod−equal" (remainder of dividing variable by a constant)

The bitwise logical operators seldom make an appearance in shell scripts. Their chief use seems to be manipulating and testing values read from ports or sockets. "Bit flipping" is more relevant to compiled languages, such as C and C++, which run fast enough to permit its use on the fly.

`<<`

bitwise left shift (multiplies by `2` for each shift position)

`<<=`

"left−shift−equal"

`let "var <<= 2"` results in `var` left−shifted `2` bits (multiplied by `4`)

`>>`

bitwise right shift (divides by `2` for each shift position)

`>>=`

"right−shift−equal" (inverse of `<<=`)

`&`

bitwise and

`&=`

"bitwise and−equal"

|

bitwise OR

|=

"bitwise OR−equal"

~

bitwise negate

!

bitwise NOT

^

bitwise XOR

^=

"bitwise XOR−equal"

relational tests

<

less than

>

greater than

<=

less than or equal to

>=

greater than or equal to

==

equal to (test)

!=

not equal to

&&

and (logical)

```
if [ $condition1 ] && [ $condition2 ]
# if both condition1 and condition2 hold true...
```

Note: && may also, depending on context, be used to in an *and list* to concatenate commands (see Section 3.21).

||

or (logical)

```
if [ $condition1 ] || [ $condition2 ]
# if both condition1 or condition2 hold true...
```

Example 3−15. Compound Condition Tests Using && and ||

```
#!/bin/bash

a=24
b=47

if [ $a -eq 24 ] && [ $b -eq 47 ]
then
  echo "Test #1 succeeds."
else
  echo "Test #1 fails."
fi

# ERROR:
# if [ $a -eq 24 && $b -eq 47 ]


if [ $a -eq 98 ] || [ $b -eq 47 ]
then
  echo "Test #2 succeeds."
else
  echo "Test #2 fails."
fi


# The -a and -o options provide
# an alternative compound condition test.
# Thanks to Patrick Callahan for pointing this out.


if [ $a -eq 24 -a $b -eq 47 ]
then
  echo "Test #3 succeeds."
else
  echo "Test #3 fails."
fi


if [ $a -eq 98 -o $b -eq 47 ]
then
  echo "Test #4 succeeds."
else
```

```
  echo "Test #4 fails."
fi


a=rhino
b=crocodile
if [ $a = rhino ] && [ $b = crocodile ]
then
  echo "Test #5 succeeds."
else
  echo "Test #5 fails."
fi

exit 0
```

3.6.2. Numerical Constants

A shell script interprets a number as decimal (base 10), unless that number has a special prefix or notation. A number preceded by a *0* is *octal* (base 8). A number preceded by *0x* is *hexadecimal* (base 16). A number with an embedded *#* is evaluated as *BASE#NUMBER* (this option is of limited usefulness because of range restrictions).

Example 3−16. Representation of numerical constants:

```
#!/bin/bash

# Representation of numbers.

# Decimal
let "d = 32"
echo "d = $d"
# Nothing out of the ordinary here.


# Octal: numbers preceded by '0'
let "o = 071"
echo "o = $o"
# Expresses result in decimal.

# Hexadecimal: numbers preceded by '0x' or '0X'
let "h = 0x7a"
echo "h = $h"

# Other bases: BASE#NUMBER
# BASE between 2 and 64.
let "b = 32#77"
echo "b = $b"
# This notation only works for a very limited range of numbers.
let "c = 2#47"  # Error: out of range.
echo "c = $c"


exit 0
```

3.7. Variables Revisited

Used properly, variables can add power and flexibility to scripts. This requires learning their subtleties and nuances.

`Internal (builtin) variables`

environmental variables affecting bash script behavior

`$IFS`

input field separator

This defaults to white space, but may be changed, for example, to parse a comma−separated data file.

`$HOME`

home directory of the user, usually `/home/username` (see Example 3−6)

`$HOSTNAME`

name assigned to the system, usually fetched at bootup from `/etc/hosts` (see Example 3−6)

`$UID`

user id number

current user's user identification number, as recorded in `/etc/passwd`

This is the current user's real id, even if she has temporarily assumed another identity through **su** (see Section 3.11). `$UID` is a readonly variable, not subject to change from the command line or within a script.

`$EUID`

"effective" user id number

identification number of whatever identity the current user has assumed, perhaps by means of **su**

`$GROUPS`

groups current user belongs to

This is a listing (array) of the group id numbers for current user, as recorded in `/etc/passwd`.

`$PATH`

path to binaries, usually `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, etc.

When given a command, the shell automatically searches the directories listed in the *path* for the executable. The path is stored in the environmental variable, `$PATH`, a list of directories, separated by colons. Normally, the system stores the `$PATH` definition in `/etc/profile` and/or `~/.bashrc` (see Section 3.23).

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

`PATH=${PATH}:/opt/bin` appends the `/opt/bin` directory to the current path. In a script, it may be expedient to temporarily add a directory to the path in this way. When the script exits, this restores the original `$PATH` (a child process, such as a script, may not change the environment of the parent process, the shell).

Note: The current "working directory", **`./`**, is usually omitted from the `$PATH` as a security measure.

`$PS1`

This is the main prompt, seen at the command line.

`$PS2`

The secondary prompt, seen when additional input is expected. It displays as ">".

`$PS3`

The tertiary prompt, displayed in a **select** loop (see Example 3−37).

`$PS4`

The quartenary prompt, shown at the beginning of each line of output when invoking a script with the −x option. It displays as "+".

`$PWD`

working directory (directory you are in at the time)

```
#!/bin/bash

WRONG_DIRECTORY=33

clear # Clear screen.

TargetDirectory=/home/bozo/projects/GreatAmericanNovel

cd $TargetDirectory
echo "Deleting stale files in $TargetDirectory."

if [ $PWD != $TargetDirectory ]  # Keep from wiping out wrong directory by accident.
then
  echo "Wrong directory!"
  echo "In $PWD, rather than $TargetDirectory!"
  echo "Bailing out!"
  exit $WRONG_DIRECTORY
```

```
fi

rm -rf *
rm .[A-Za-z0-9]*    # Delete dotfiles.

# Various other operations here, as necessary.

echo
echo "Done."
echo "Old files deleted in $TargetDirectory."
echo


exit 0
```

$OLDPWD

old working directory (previous directory you were in)

$DIRSTACK

contents of the directory stack (affected by **pushd** and **popd**)

This builtin variable is the counterpart to the **dirs** command (see Section 3.9).

$PPID

the process id (pid) of the currently running process

This corresponds to the **pidof** command (see Section 3.11).

$MACHTYPE

machine type

Identifies the system hardware.

```
bash$ echo $MACHTYPE
i686-debian-linux-gnu
```

$HOSTTYPE

host type

Like `$MACHTYPE` above, identifies the system hardware.

```
bash$ echo $HOSTTYPE
i686
```

$OSTYPE

operating system type

```
bash$ echo $OSTYPE
```

```
linux-gnu
```

$EDITOR

the default editor invoked by a script, usually **vi** or **emacs**.

$IGNOREEOF

ignore EOF: how many end−of−files (control−D) the shell will ignore before logging out.

$TMOUT

If the *$TMOUT* environmental variable is set to a non−zero value *time*, then the shell prompt will time out after *time* seconds. This will cause a logout.

> **Note:** Unfortunately, this works only while waiting for input at the shell prompt console or in an xterm. While it would be nice to speculate on the uses of this internal variable for timed input, for example in combination with **read**, *$TMOUT* does not work in that context and is virtually useless for shell scripting. (Reportedly the *ksh* version of a timed **read** does work).

Implementing timed input in a script is certainly possible, but hardly seems worth the effort. It requires setting up a timing loop to signal the script when it times out. Additionally, a signal handling routine is necessary to trap (see Example 3−100) the interrupt generated by the timing loop (whew!).

```
#!/bin/bash

# TMOUT=3            useless in a script

TIMELIMIT=3  # Three seconds in this instance, may be set to different value.

PrintAnswer()
{
  if [ $answer = TIMEOUT ]
  then
    echo $answer
  else       # Don't want to mix up the two instances.
    echo "Your favorite veggie is $answer"
    kill $!  # Kills no longer needed TimerOn function running in background.
             #   $! is PID of last job running in background.
  fi

}



TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # Waits 3 seconds, then sends sigalarm to script.
}

Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit 14
}
```

```
trap Int14Vector 14
# Timer interrupt − 14 − subverted for our purposes.

echo "What is your favorite vegetable "
TimerOn
read answer
PrintAnswer


# Admittedly, this is a kludgy implementation of timed input,
# but pretty much as good as can be done with Bash.
# (Challenge to reader: come up with something better.)

# If you need something a bit more elegant...
# consider writing the application in C or C++,
# using appropriate library functions, such as 'alarm' and 'setitimer'.

exit 0
```

$SECONDS

The number of seconds the script has been running.

```
#!/bin/bash

ENDLESS_LOOP=1

echo
echo "Hit Control−C to exit this script."
echo

while [ $ENDLESS_LOOP ]
do
  if [ $SECONDS −eq 1 ]
  then
    units=second
  else
    units=seconds
  fi

  echo "This script has been running $SECONDS $units."
  sleep 1
done


exit 0
```

$REPLY

The default value when a variable is not supplied to **read**. Also applicable to **select** menus, but only supplies the item number of the variable chosen, not the value of the variable itself.

```
#!/bin/bash


echo
echo −n "What is your favorite vegetable? "
read
```

```
echo "Your favorite vegetable is $REPLY."
# REPLY holds the value of last "read" if and only if no variable supplied.


echo
echo -n "What is your favorite fruit? "
read fruit
echo "Your favorite fruit is $fruit."
echo "but..."
echo "Value of \$REPLY is still $REPLY."
# $REPLY is still set to its previous value because
# the variable $fruit absorbed the new "read" value.

echo

exit 0
```

$SHELLOPTS

the list of enabled shell options, a readonly variable

$BASH

the path to the **bash** binary itself, usually `/bin/bash`

$BASH_ENV

an environmental variable pointing to a bash startup file to be read when a script is invoked

$BASH_VERSION

the version of Bash installed on the system

```
bash$ echo $BASH_VERSION
2.04.12(1)-release
```

$0, $1, $2, etc.

positional parameters, passed from command line to script, passed to a function, or **set** to a variable (see Example 3−20 and Example 3−40)

$#

number of command line arguments [2] or positional parameters (see Example 2−4)

$$

process id of script, often used in scripts to construct temp file names (see Example A−5 and Example 3−101)

$?

exit status of command, function, or the script itself (see Example 3−1 and Example 3−56)

*$**

All of the positional parameters, seen as a single word

$@

Same as $*, but each parameter is a quoted string, that is, the parameters are passed on intact, without interpretation or expansion. This means, among other things, that each parameter in the argument list is seen as a separate word.

Example 3−17. arglist: Listing arguments with $* and $@

```
#!/bin/bash
# Invoke this script with several arguments, such as "one two three".

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` argument1 argument2 etc."
  exit 1
fi


echo

index=1

echo "Listing args with \"\$*\":"
for arg in "$*"  # Doesn't work properly if "$*" isn't quoted.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done   # $* sees all arguments as single word.
echo "Entire arg list seen as single word."

echo

index=1

echo "Listing args with \"\$@\":"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done   # $@ sees arguments as separate words.
echo "Arg list seen as separate words."

echo

exit 0
```

The `$@` intrinsic variable finds use as a "general input filter" tool in shell scripts. The **cat "$@"** construction accepts input to a script either from `stdin` or from files given as parameters to the script. See Example 3−59.

$-

Flags passed to script

Caution
This was originally a *ksh* construct adopted into Bash, and unfortunately it does not seem to work reliably in Bash scripts. One possible use for it is to have a script self−test whether it is interactive (see Section 3.29).

$!

PID (process id) of last job run in background

variable assignment

Initializing or changing the value of a variable

=

the assignment operator (*no space before & after*)

Do not confuse this with = and −eq, which test, rather than assign!

Caution
= can be *either* an assignment or a test operator, depending on context.

Example 3−18. Variable Assignment

```
#!/bin/bash

echo

# When is a variable "naked", i.e., lacking the '$' in front?

# Assignment
a=879
echo "The value of \"a\" is $a"

# Assignment using 'let'
let a=16+5
echo "The value of \"a\" is now $a"

echo

# In a 'for' loop (really, a type of disguised assignment)
echo -n "The values of \"a\" in the loop are "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# In a 'read' statement
```

```
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a"

echo

exit 0
```

Example 3−19. Variable Assignment, plain and fancy

```
#!/bin/bash

a=23
# Simple case
echo $a
b=$a
echo $b

# Now, getting a little bit fancier...

a=`echo Hello!`
# Assigns result of 'echo' command to 'a'
echo $a

a=`ls -l`
# Assigns result of 'ls -l' command to 'a'
echo $a

exit 0
```

Variable assignment using the $() mechanism (a newer method than back quotes)

```
# From /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

local variables

variables visible only within a code block or function (see Section 3.19)

environmental variables

variables that affect the behavior of the shell and user interface, such as the path and the prompt

If a script sets environmental variables, they need to be "exported", that is, reported to the environment local to the script. This is the function of the **export** command.

> **Note:** A script can **export** variables only to child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command line *cannot* export variables back to the command line environment. Child processes cannot export variables back to the parent processes that spawned them.

−−−

positional parameters

arguments passed to the script from the command line − $0, $1, $2, $3... ($0 is the name of the script itself, $1 is the first argument, etc.)

Example 3−20. Positional Parameters

```
#!/bin/bash

echo

echo The name of this script is $0
# Adds ./ for current directory
echo The name of this script is `basename $0`
# Strip out path name info (see 'basename')

echo

if [ $1 ]
then
 echo "Parameter #1 is $1"
 # Need quotes to escape #
fi

if [ $2 ]
then
 echo "Parameter #2 is $2"
fi

if [ $3 ]
then
 echo "Parameter #3 is $3"
fi

echo

exit 0
```

Some scripts can perform different operations, depending on which name they are invoked with. For this to work, the script needs to check `$0`, the name it was invoked by. There also have to be symbolic links present to all the alternate names of the same script.

> **Note:** If a script expects a command line parameter but is invoked without one, this may cause a null variable assignment, certainly an undesirable result. One way to prevent this is to append an extra character to both sides of the assignment statement using the expected positional parameter.

```
variable1x=$1x
# This will prevent an error, even if positional parameter is absent.

# The extra character can be stripped off later, if desired, like so.
variable1=${variable1x/x/}
# This uses one of the parameter substitution templates previously discussed.
# Leaving out the replacement pattern results in a deletion.
```

−−−

Example 3−21. wh, whois domain name lookup

```
#!/bin/bash

# Does a 'whois domain-name' lookup
# on any of 3 alternate servers:
# ripe.net, cw.net, radb.net

# Place this script, named 'wh' in /usr/local/bin

# Requires symbolic links:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb


if [ -z $1 ]
then
  echo "Usage: `basename $0` [domain-name]"
  exit 1
fi

case `basename $0` in
# Checks script name and calls proper server
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "Usage: `basename $0` [domain-name]";;
esac

exit 0
```

−−−

The **shift** command reassigns the positional parameters, in effect shifting them to the left one notch.

`$1` <−−− `$2`, `$2` <−−− `$3`, `$3` <−−− `$4`, etc.

The old `$1` disappears, but `$0` does not change. If you use a large number of positional parameters to a script, **shift** lets you access those past `10`.

Example 3−22. Using shift

```
#!/bin/bash

# Name this script something like shift000,
# and invoke it with some parameters, for example
# ./shift000 a b c def 23 skidoo

# Demo of using 'shift'
# to step through all the positional parameters.

until [ -z "$1" ]
do
  echo -n "$1 "
  shift
```

```
done

echo
# Extra line feed.

exit 0
```

3.7.1. Typing variables: declare or typeset

The **declare** or **typeset** keywords (they are exact synonyms) permit restricting the properties of variables. This is a very weak form of the typing available in certain programming languages. The **declare** command is not available in version 1 of **bash**.

−r readonly

```
declare −r var1
```

(**`declare −r var1`** works the same as **`readonly var1`**)

This is the rough equivalent of the C **const** type qualifier. An attempt to change the value of a readonly variable fails with an error message.

−i integer

```
declare −i var2
```

The script treats subsequent occurrences of `var2` as an integer. Note that certain arithmetic operations are permitted for declared integer variables without the need for **expr** or **let**.

−a array

```
declare −a indices
```

The variable `indices` will be treated as an array.

−f functions

```
declare −f  # (no arguments)
```

A **`declare −f`** line within a script causes a listing of all the functions contained in that script.

−x export

```
declare −x var3
```

This declares a variable as available for exporting outside the environment of the script itself.

Example 3−23. Using declare to type variables

```
#!/bin/bash

declare -f
# Lists the function below.

func1 ()
{
echo This is a function.
}

declare -r var1=13.36
echo "var1 declared as $var1"
# Attempt to change readonly variable.
var1=13.37
# Generates error message.
echo "var1 is still $var1"

echo

declare -i var2
var2=2367
echo "var2 declared as $var2"
var2=var2+1
# Integer declaration eliminates the need for 'let'.
echo "var2 incremented by 1 is $var2."
# Attempt to change variable declared as integer
echo "Attempting to change var2 to floating point value, 2367.1."
var2=2367.1
# results in error message, with no change to variable.
echo "var2 is still $var2"

exit 0
```

3.7.2. Indirect References to Variables

Assume that the value of a variable is the name of a second variable. Is it somehow possible to retrieve the value of this second variable from the first one? For example, if `a=letter_of_alphabet` and `letter_of_alphabet=z`, can a reference to `a` return `z`? This can indeed be done, and it is called an *indirect reference*. It uses the unusual `eval var1=\$$var2` notation.

Example 3−24. Indirect References

```
#!/bin/bash

# Indirect variable referencing.


a=letter_of_alphabet
letter_of_alphabet=z

# Direct reference.
echo "a = $a"

# Indirect reference.
eval a=\$$a
echo "Now a = $a"
```

```
echo


# Now, let's try changing the second order reference.

t=table_cell_3
table_cell_3=24
eval t=\$$t
echo "t = $t"
# So far, so good.

table_cell_3=387
eval t=\$$t
echo "Value of t changed to $t"
# ERROR!
# Cannot indirectly reference changed value of variable this way.
# For this to work, must use ${!t} notation.


exit 0
```

Caution
This method of indirect referencing has a weakness. If the second order variable changes its value, an indirect reference to the first order variable produces an error. Fortunately, this flaw has been fixed in the newer *${!variable}* notation introduced with version 2 of Bash (see Example 3−103).

3.7.3. $RANDOM: generate random integer

> **Note:** $RANDOM is an internal Bash function (not a constant) that returns a *pseudorandom* integer in the range 0 − 32767. $RANDOM should `not` be used to generate an encryption key.

Example 3−25. Generating random numbers

```
#!/bin/bash

# $RANDOM returns a different random integer at each invocation.
# Nominal range: 0 - 32767 (signed integer).

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT random numbers:"
echo "-----------------"
while [ $count -le $MAXCOUNT ]      # Generate 10 ($MAXCOUNT) random integers.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # Increment count.
done
echo "-----------------"
```

```
# If you need a random int within a certain range, then use the 'modulo' operator.

RANGE=500

echo

number=$RANDOM
let "number %= $RANGE"
echo "Random number less than $RANGE  -->  $number"

echo

# If you need a random int greater than a lower bound,
# then set up a test to discard all numbers below that.

FLOOR=200

number=0   #initialize
while [ $number -le $FLOOR ]
do
  number=$RANDOM
done
echo "Random number greater than $FLOOR -->  $number"
echo


# May combine above two techniques to retrieve random number between two limits.
number=0   #initialize
while [ $number -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"
done
echo "Random number between $FLOOR and $RANGE -->  $number"
echo


# May generate binary choice, that is, "true" or "false" value.
BINARY=2
number=$RANDOM
let "number %= $BINARY"
if [ $number -eq 1 ]
then
  echo "TRUE"
else
  echo "FALSE"
fi

echo


# May generate toss of the dice.
SPOTS=7
DICE=2
die1=0
die2=0

# Tosses each die separately, and so gives correct odds.

  while [ $die1 -eq 0 ]   #Can't have a zero come up.
  do
```

```
    let "die1 = $RANDOM % $SPOTS"
  done

  while [ $die2 -eq 0 ]
  do
    let "die2 = $RANDOM % $SPOTS"
  done

let "throw = $die1 + $die2"
echo "Throw of the dice = $throw"
echo


exit 0
```

Note: The variables `$USER`, `$USERNAME`, `$LOGNAME`, `$MAIL`, and `$ENV` are *not* Bash builtins. These are, however, often set as environmental variables in one of the Bash startup files (see Section 3.23). `$SHELL` is a readonly variable set from `/etc/passwd` and is likewise not a Bash builtin.

3.8. Loops

A *loop* is a block of code that iterates (repeats) a list of commands as long as the loop control condition is true.

for (in)

This is the basic looping construct. It differs significantly from its C counterpart.

for [*`arg`*] in [*`list`*]
do
 `command`...
done

Note that *`list`* may contain wild cards.

Note further that if **do** is on same line as **for**, there needs to be a semicolon before list.

for [*`arg`*] in [*`list`*] ; do

Example 3−26. Simple for loops

```
#!/bin/bash

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet
done

echo

# Entire 'list' enclosed in quotes creates a single variable.
```

```
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
do
  echo $planet
done

exit 0
```

Note: Each `[list]` element may contain multiple parameters. This is useful when processing parameters in groups. In such cases, use the **set** command (see Example 3−40) to force parsing of each `[list]` element and assignment of each component to the positional parameters.

Example 3−27. for loop with two parameters in each [list] element

```
#!/bin/bash
# Planets revisited.

# Want to associate name of each planet with its distance from the sun.

for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
  set $planet  # Parses variable "planet" and sets positional parameters.
  # May need to save original positional parameters, since they get overwritten.
  echo "$1              $2,000,000 miles from the sun"
  #-------two  tabs---concatenate zeroes onto parameter $2
done

exit 0
```

Omitting the `in [list]` part of a **for** loop causes the loop to operate on $*, the list of arguments given on the command line to the script.

Example 3−28. Missing `in [list]` in a for loop

```
#!/bin/bash

# Invoke both with and without arguments,
# and see what happens.

for a
do
 echo $a
done

# 'in list' missing, therefore operates on '$*'
# (command-line argument list)

exit 0
```

Example 3−29. Using efax in batch mode

```
#!/bin/bash

if [ $# -ne 2 ]
```

```
# Check for proper no. of command line args.
then
   echo "Usage: `basename $0` phone# text-file"
   exit 1
fi


if [ ! -f $2 ]
then
  echo "File $2 is not a text file"
  exit 2
fi


# Create fax formatted files from text files.
fax make $2

for file in $(ls $2.0*)
# Concatenate the converted files.
# Uses wild card in variable list.
do
  fil="$fil $file"
done

# Do the work.
efax -d /dev/ttyS3 -o1 -t "T$1" $fil

exit 0
```

while

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true.

while [*condition*]
do
 command...
done

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

while [*condition*] ; do

Note that certain specialized **while** loops, as, for example, a **getopts** construct, deviate somewhat from the standard template given here (see Section 3.9).

Example 3−30. Simple while loop

```
#!/bin/bash

var0=0

while [ "$var0" -lt 10 ]
do
  echo -n "$var0 "
  # -n suppresses newline.
  var0=`expr $var0 + 1`
```

```
  # var0=$(($var0+1)) also works.
done

echo

exit 0
```

Example 3−31. Another while loop

```
#!/bin/bash

echo

while [ "$var1" != end ]
do
  echo "Input variable #1 (end to exit) "
  read var1
  # It's not 'read $var1' because value of var1 is being set.
  echo "variable #1 = $var1"
  # Need quotes because of #
  echo
done

# Note: Echoes 'end' because termination condition tested for at top of loop.

exit 0
```

Note: A **while** loop may have multiple conditions. Only the final condition determines when the loop terminates. This necessitates a slightly different loop syntax, however.

Example 3−32. while loop with multiple conditions

```
#!/bin/bash

var1=unset
previous=$var1

while echo "previous-variable = $previous"
      echo
      previous=$var1
      [ "$var1" != end ] # Keeps track of what "var1" was previously.
      # Four conditions on "while", but only last one controls loop.
      # Controlling condition has [ test ] brackets.
do
echo "Input variable #1 (end to exit) "
  read var1
  echo "variable #1 = $var1"
done

# Try to figure out how this all works.
# It's a wee bit tricky.

exit 0
```

Note: A **while** loop may have its *stdin* redirected to a file by a < at its end (see Example 3−73).

until

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of **while** loop).

until [*condition−is−true*]
do
 command...
done

Note that an **until** loop tests for the terminating condition at the top of the loop, differing from a similar construct in some programming languages.

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

until [*condition−is−true*] ; do

Example 3−33. until loop

```
#!/bin/bash

until [ "$var1" = end ]
# Tests condition at top of loop.
do
  echo "Input variable #1 "
  echo "(end to exit)"
  read var1
  echo "variable #1 = $var1"
done

exit 0
```

break, continue

The **break** and **continue** loop control commands correspond exactly to their counterparts in other programming languages. The **break** command terminates the loop (breaks out of it), while **continue** causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Example 3−34. Effects of break and continue in a loop

```
#!/bin/bash

echo
echo Printing Numbers 1 through 20.

a=0

while [ $a -le 19 ]

do
 a=$(($a+1))

 if [ $a -eq 3 ] || [ $a -eq 11 ]
```

```
 # Excludes 3 and 11
 then
   continue
   # Skip rest of this particular loop iteration.
 fi

 echo -n "$a "
done

# Exercise for reader:
# Why does loop print up to 20?

echo
echo

echo Printing Numbers 1 through 20, but something happens after 2.

##################################################################

# Same loop, but substituting 'break' for 'continue'.

a=0

while [ $a -le 19 ]
do
 a=$(($a+1))

 if [ $a -gt 2 ]
 then
   break
   # Skip entire rest of loop.
 fi

 echo -n "$a "
done

echo
echo

exit 0
```

case (in) / esac

The **case** construct is the shell equivalent of **switch** in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

case "$*variable*" in

 "$*condition1*")
 command...
 ;;

 "$*condition2*")
 command...
 ;;

esac

Note:

- Quoting the variables is recommended.
- Each test line ends with a left paren).
- Each condition block ends with a *double* semicolon ;;.
- The entire **case** block terminates with an **esac** (*case* spelled backwards).

Example 3−35. Using case

```
#!/bin/bash

echo
echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "Lowercase letter";;
  [A-Z]   ) echo "Uppercase letter";;
  [0-9]   ) echo "Digit";;
  *       ) echo "Punctuation, whitespace, or other";;
esac
# Allows ranges of characters in [square brackets].

exit 0
```

Example 3−36. Creating menus using case

```
#!/bin/bash

# Crude rolodex-type database

clear
# Clear the screen.

echo "          Contact List"
echo "          ------- ----"
echo "Choose one of the following persons:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# Note variable is quoted.

  "E" | "e" )
  # Accept upper or lowercase input.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
```

```
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
# Note double semicolon to terminate
# each option.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Add info for Smith & Zane later.

          * )
   # Default option.
   echo
   echo "Not yet in database."
  ;;


esac

echo

exit 0
```

select

The **select** construct, adopted from the Korn Shell, is yet another tool for building menus.

select `variable` [in `list`]
do
 `command`...
 break
done

This prompts the user to enter one of the choices presented in the variable list. Note that **select** uses the `PS3` prompt (`#?`) by default, but that this may be changed.

Example 3−37. Creating menus using select

```
#!/bin/bash

PS3='Choose your favorite vegetable: '
# Sets the prompt string.

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
```

```
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break
  # if no 'break' here, keeps looping forever.
done

exit 0
```

If **`in list`** is omitted, then **select** uses the list of command line arguments (`$@`) passed to the script or to the function in which the **select** construct is embedded. (Compare this to the behavior of a

for `variable` [in `list`]

construct with the **`in list`** omitted.)

Example 3−38. Creating menus using select in a function

```
#!/bin/bash

PS3='Choose your favorite vegetable: '

echo

choice_of()
{
select vegetable
# [in list] omitted, so 'select' uses arguments passed to function.
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break
done
}

choice_of beans rice carrots radishes tomatoes spinach
#         $1    $2   $3      $4       $5       $6
#         passed to choice_of() function

exit 0
```

3.9. Internal Commands and Builtins

A *builtin* is a command contained in the bash tool set, literally built in.

getopts

This powerful tool parses command line arguments passed to the script. This is the bash analog of the **getopt** library function familiar to C programmers. It permits passing and concatenating multiple flags[3] and options to a script (for example **`scriptname -abc -e /usr/local`**).

The **getopts** construct uses two implicit variables. `$OPTIND` is the argument pointer (*OPTion INDex*) and `$OPTARG` (*OPTion ARGument*) the (optional) argument attached to a flag. A colon following the flag name in the declaration tags that flag as having an option.

A **getopts** construct usually comes packaged in a **while** loop, which processes the flags and options one at a time, then decrements the implicit `$OPTIND` variable to step to the next.

Note:

1. The arguments must be passed from the command line to the script preceded by a minus (−) or a plus (+), else **getopts** will not process them, and will, in fact, terminate option processing at the first argument encountered lacking these modifiers.
2. The **getopts** template differs slightly from the standard **while** loop, in that it lacks condition brackets.
3. The **getopts** construct replaces the obsolete **getopt** command.

```
while getopts ":abcde:fg" Option
# Initial declaration.
# a, b, c, d, e, f, and g are the flags expected.
# The : after flag 'e' shows it will have an option passed with it.
do
  case $Option in
    a ) # Do something with variable 'a'.
    b ) # Do something with variable 'b'.
    ...
    e)  # Do something with 'e', and also with $OPTARG,
        # which is the associated argument passed with 'e'.
    ...
    g ) # Do something with variable 'g'.
  esac
done
shift $(($OPTIND - 1))
# Move argument pointer to next.

# All this is not nearly as complicated as it looks <grin>.
```

Example 3−39. Using getopts to read the flags/options passed to a script

```
#!/bin/bash

# 'getopts' processes command line args to script.

# Usage: scriptname -options
# Note: dash (-) necessary

# Try invoking this script with
# 'scriptname -mn'
# 'scriptname -oq qOption'
# (qOption can be some arbitrary string.)

OPTERROR=33

if [ -z $1 ]
# Exit and complain if no argument(s) given.
```

```
then
  echo "Usage: `basename $0` options (-mnopqrs)"
  exit $OPTERROR
fi

while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scenario #1: option -m-";;
    n | o ) echo "Scenario #2: option -$Option-";;
    p     ) echo "Scenario #3: option -p-";;
    q     ) echo "Scenario #4: option -q-, with argument \"$OPTARG\"";;
    # Note that option 'q' must have an additional argument,
    # otherwise nothing happens.
    r | s ) echo "Scenario #5: option -$Option-"'';;
    *     ) echo "Unimplemented option chosen.";;
  esac
done

shift $(($OPTIND - 1))
# Decrements the argument pointer
# so it points to next argument.

exit 0
```

exit

Unconditionally terminates a script. The **exit** command may optionally take an integer argument, which is returned to the shell as the *exit status* of the script. It is a good practice to end all but the simplest scripts with an **`exit 0`**, indicating a successful run.

set

The **set** command changes the value of internal script variables. One use for this is to toggle option flags which help determine the behavior of the script (see Section 3.27). Another application for it is to reset the positional parameters that a script sees as the result of a command (**`set `command``**). The script can then parse the fields of the command output.

Example 3−40. Using set with positional parameters

```
#!/bin/bash

# script "set-test"

# Invoke this script with three command line parameters,
# for example, "./set-test one two three".

echo
echo "Positional parameters before  set \`uname -a\` :"
echo "Command-line argument #1 = $1"
echo "Command-line argument #2 = $2"
echo "Command-line argument #3 = $3"

echo

set `uname -a`
# Sets the positional parameters to the output
```

```
# of the command `uname -a`

echo "Positional parameters after  set \`uname -a\` :"
# $1, $2, $3, etc. reinitialized to result of `uname -a`
echo "Field #1 of 'uname -a' = $1"
echo "Field #2 of 'uname -a' = $2"
echo "Field #3 of 'uname -a' = $3"
echo

exit 0
```

unset

The **unset** command deletes an internal script variable. It is a way of negating a previous **set**. Note that this command does not affect positional parameters.

export

The **export** command makes available variables to all child processes of the running script or shell. Unfortunately, there is no way to **export** variables back to the parent process, to the process that called or invoked the script or shell. One important use of **export** command is in startup files, to initialize and make accessible environmental variables to subsequent user processes (see Section 3.23).

> **Note:** It is possible to initialize and export variables in the same operation, as in **export var1=xxx**.

readonly

Same as **`declare -r`**, sets a variable as read−only, or, in effect, as a constant. Attempts to change the variable fail with an error message. This is the shell analog of the C language **const** type qualifier.

basename

Strips the path information from a file name, printing only the file name. The construction **`basename $0`** lets the script know its name, that is, the name it was invoked by. This can be used for "usage" messages if, for example a script is called with missing arguments:

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Strips the **basename** from a file name, printing only the path information.

> **Note: basename** and **dirname** can operate on any arbitrary string. The filename given as an argument does not need to refer to an existing file.

Example 3−41. basename and dirname

```
#!/bin/bash

a=/home/heraclius/daily-journal.txt

```

```
echo "Basename of /home/heraclius/daily-journal.txt = `basename $a`"
echo "Dirname of /home/heraclius/daily-journal.txt = `dirname $a`"

exit 0
```

read

"Reads" the value of a variable from stdin, that is, interactively fetches input from the keyboard. The `-a` option lets **read** get array variables (see Example 3−90).

Example 3−42. Variable assignment, using read

```
#!/bin/bash

echo -n "Enter the value of variable 'var1': "
# -n option to echo suppresses newline

read var1
# Note no '$' in front of var1, since it is being set.

echo "var1 = $var1"


# Note that a single 'read' statement can set multiple variables.

echo

echo -n "Enter the values of variables 'var2' and 'var3' (separated by a space or tab): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
# If you input only one value, the other variable(s) will remain unset (null).

exit 0
```

The **read** command may also "read" its variable value from a file redirected to `stdin` (see Section 3.13). If the file contains more than one line, only the first line is assigned to the variable. If there is more than one parameter to the **read**, then each variable gets assigned a successive whitespace delineated string. Caution!

```
read var1 <data-file
echo "var1 = $var1"
# var1 set to the entire first line of the input file "data-file"

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# Note inconsistent behavior of "read" here.
# 1) Rewinds back to the beginning of input file.
# 2) Each variable is now set to a corresponding string, separated by whitespace,
#    rather than to an entire line of text.
# 3) The final variable gets the remainder of the line.
# 4) If there are more variables to be set than whitespace-terminated strings
#    on the first line of the file, then the excess variable remain unset.
```

true

A command that returns a successful (zero) exit status, but does nothing else.

```
# Endless loop
while true
# alias for :
do
   operation-1
   operation-2
   ...
   operation-n
   # Need a way to break out of loop.
done
```

false

A command that returns an unsuccessful exit status, but does nothing else.

```
# Null loop
while false
do
   # The following code will not execute.
   operation-1
   operation-2
   ...
   operation-n
   # Nothing happens!
done
```

factor

Factor an integer into prime factors.

```
bash$ factor 27417
27417: 3 13 19 37
```

hash [cmds]

Record the path name of specified commands (in the shell hash table), so the shell or script will not need to search the `$PATH` on subsequent calls to those commands. When **hash** is called with no arguments, it simply lists the commands that have been hashed.

pwd

Print Working Directory. This gives the user's (or script's) current directory (see Example 3−43). The effect is identical to reading the value of the builtin variable **$PWD** (see Section 3.7).

pushd**, **popd**, **dirs

This command set is a mechanism for bookmarking working directories, a means of moving back and forth through directories in an orderly manner. A pushdown stack is used to keep track of directory names. Options allow various manipulations of the directory stack.

`pushd dir-name` pushes the path *`dir-name`* onto the directory stack and simultaneously changes the current working directory to *`dir-name`*

popd removes (pops) the top directory path name off the directory stack and simultaneously changes the current working directory to that directory popped from the stack.

dirs lists the contents of the directory stack (counterpart to `$DIRSTACK`, see below). A successful **pushd** or **popd** will automatically invoke **dirs**.

Scripts that require various changes to the current working directory without hard−coding the directory name changes can make good use of these commands. Note that the implicit `$DIRSTACK` array variable, accessible from within a script, holds the contents of the directory stack.

Example 3−43. Changing the current working directory

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Will do an automatic 'dirs'
# (list directory stack to stdout).
echo "Now in directory `pwd`."
# Uses back-quoted 'pwd'.
# Now, do some stuff in directory 'dir1'.
pushd $dir2
echo "Now in directory `pwd`."
# Now, do some stuff in directory 'dir2'.
echo "The top entry in the DIRSTACK array is $DIRSTACK."
popd
echo "Now back in directory `pwd`."
# Now, do some more stuff in directory 'dir1'.
popd
echo "Now back in original working directory `pwd`."

exit 0
```

source, *. (**dot** command), **dirs***

This command, when invoked from the command line, executes a script. Within a script, a **`source file-name`** loads the file `file-name`. This is the shell scripting equivalent of a C/C++ **`#include`** directive. It is useful in situations when multiple scripts use a common data file or function library.

Example 3−44. "Including" a data file

```
#!/bin/bash

# Load a data file.
. data-file
# Same effect as "source data-file"

# Note that the file "data-file", given below
# must be present in working directory.

# Now, reference some data from that file.
```

```
echo "variable1 (from data-file) = $variable1"
echo "variable3 (from data-file) = $variable3"

let "sum = $variable2 + $variable4"
echo "Sum of variable2 + variable4 (from data-file) = $sum"
echo "message1 (from data-file) is \"$message1\""
# Note:                            escaped quotes

print_message This is the message-print function in the data-file.


exit 0
```

File `data-file` for Example 3−44, above. Must be present in same directory.

```
# This is a data file loaded by a script.
# Files of this type may contain variables, functions, etc.
# It may be loaded with a 'source' or '.' command by a shell script.

# Let's initialize some variables.

variable1=22
variable2=474
variable3=5
variable4=97

message1="Hello, how are you?"
message2="Enough for now. Goodbye."

print_message ()
{
# Echoes any message passed to it.

  if [ -z $1 ]
  then
    return 1
    # Error, if argument missing.
  fi

  echo

  until [ -z "$1" ]
  do
    # Step through arguments passed to function.
    echo -n "$1"
    # Echo args one at a time, suppressing line feeds.
    echo -n " "
    # Insert spaces between words.
    shift
    # Next one.
  done

  echo

  return 0
}
```

3.9.1. Job Control Commands

ps

Lists currently executing jobs by owner and process id. This is usually invoked with `ax` options, and may be piped to **grep** or **sed** to search for a specific process (see Example 3−51).

```
bash$ ps ax | grep sendmail
295 ?      S      0:00 sendmail: accepting connections on port 25
```

wait

Stop script execution until all jobs running in background have terminated, or until the job number specified as an option terminates. Sometimes used to prevent a script from exiting before a background job finishes executing (this would create a dreaded orphan process).

Example 3−45. Waiting for a process to finish before proceeding

```
#!/bin/bash

if [ -z $1 ]
then
  echo "Usage: `basename $0` find-string"
  exit 1
fi

echo "Updating 'locate' database..."
echo "This may take a while."
updatedb /usr &
# Must be run as root.

wait
# Don't run the rest of the script until 'updatedb' finished.
# You want the the database updated before looking up the file name.

locate $1

# Lacking the wait command, in the worse case scenario,
# the script would exit while 'updatedb' was still running,
# leaving it as an orphan process.

exit 0
```

suspend

This has the same effect as **Control−Z**, pausing a foreground job.

stop

This has the same effect as **suspend**, but for a background job.

disown

Remove job(s) from the shell's table of active jobs.

jobs

Lists the jobs running in the background, giving the job number. Not as useful as **ps**.

times

Gives statistics on the system time used in executing commands, in the following form:

```
0m0.020s 0m0.020s
```

This capability is of very limited value, since it is uncommon to profile and benchmark shell scripts.

kill

Forcibly terminate a process by sending it an appropriate *terminate* signal (see Example 3−69).

> **Note:** **`kill −l`** lists all the "signals". (See Section 3.26 for more detail on signals).

command

The **command** directive disables aliases and functions. This leaves only shell builtins, system commands, and commands and scripts accessible via `$PATH`.

> **Note:** This is one of three shell directives that effect script command processing. The others are **builtin** and **enable**, see below.

builtin

This disables both functions and commands in the `$PATH`, leaving only shell builtins accessible.

enable

This either enables or disables a shell builtin command. As an example, **enable −n kill** disables the shell builtin **kill**, so that when Bash subsequently encounters **kill**, it invokes `/bin/kill`. The `−a` option lists all the shell builtins, indicating whether or not they are enabled.

3.10. External Filters, Programs and Commands

This is a descriptive listing of standard UNIX commands useful in shell scripts. The power of scripts comes from coupling system commands and shell directives with simple programming constructs.

3.10.1. Basic Commands

echo

prints (to `stdout`) an expression or variable (see Example 3−5).

```
echo Hello
echo $a
```

Normally, each **echo** command prints a terminal newline, but the `−n` option suppresses this.

ls

The basic file "list" command. It is all too easy to underestimate the power of this humble command. For example, using the `−R`, recursive option, **ls** provides a tree−like listing of a directory structure.

Example 3−46. Using ls to create a table of contents for burning a CDR disk

```
#!/bin/bash

# Script to automate burning a CDR.

# Uses Joerg Schilling's "cdrecord" package
# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

# If this script invoked as an ordinary user, need to suid cdrecord
# (chmod u+s /usr/bin/cdrecord, as root).

if [ -z $1 ]
then
  IMAGE_DIRECTORY=/opt
# Default directory, if not specified on command line.
else
    IMAGE_DIRECTORY=$1
fi

ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/contents
# The "l" option gives a "long" file listing.
# The "R" option makes the listing recursive.
# The "F" option marks the file types (directories suffixed by a /).
echo "Creating table of contents."

mkisofs -r -o cdimage.iso $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image (cdimage.iso)."

cdrecord -v -isosize speed=2 dev=0,0 cdimage.iso
# Change speed parameter to speed of your burner.
echo "Burning the disk."
echo "Please be patient, this will take a while."

exit 0
```

cat**, **tac

cat, an acronym for *concatenate*, lists a file to stdout. When combined with redirection (> or >>), it is commonly used to concatenate files.

```
cat filename
              cat file.1 file.2 file.3 > file.123
```

The *−n* option to **cat** inserts consecutive numbers before each line of the target file(s).

tac, is the inverse of *cat*, listing a file backwards from its end.

rev

reverses each line of a file, and outputs to stdout. This is not the same effect as **tac**, as it preserves the order of the lines, but flips each one around.

```
bash$ cat file1.txt
This is line 1.
 This is line 2.
```

```
bash$ tac file1.txt
This is line 2.
 This is line 1.
```

```
bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cd

The familiar **cd** change directory command finds use in scripts where execution of a command requires being in a specified directory.

```
(cd /source/directory && tar cf − . ) | (cd /dest/directory && tar xvfp −)
```

[from the previously cited example by Alan Cox]

cp

This is the file copy command. **`cp file1 file2`** copies `file1` to `file2`, overwriting `file2` if it already exists (see Example 3−49).

mv

This is the file move command. It is equivalent to a combination of **cp** and **rm**. It may be used to move multiple files to a directory. For some examples of using **mv** in a script, see Example 3−7 and Example A−2.

rm

Delete (remove) a file or files. The `−f` forces removal of even readonly files.

Warning
When used with the recursive flag `−r`, this command removes files all the way down the directory tree.

rmdir

Remove directory. The directory must be empty of all files, including dotfiles, for this command to succeed.

mkdir

Make directory, creates a new directory. **`mkdir -p project/programs/December`** creates the named directory. The *`-p`* option automatically creates any necessary parent directories.

chmod

Changes the attributes of an existing file (see Example 3−51).

```
chmod +x filename
# Makes "filename" executable for all users.
```

```
chmod 644 filename
# Makes "filename" readable/writable to owner, readable to
# others
# (octal mode).
```

```
chmod 1777 directory-name
# Gives everyone read, write, and execute permission in directory,
# however also sets the "sticky bit", which means that
# only the directory owner can change files in the directory.
```

chattr

Change file attributes. This has the same effect as **chmod** above, but with a different invocation syntax.

ln

Creates links to pre−existings files. Most often used with the `-s`, symbolic or "soft" link flag. This permits referencing the linked file by more than one name and is a superior alternative to aliasing (see Example 3−21).

`ln -s oldfile newfile` links the previously existing `oldfile` to the newly created link, `newfile`.

3.10.2. Complex Commands

find

exec *`COMMAND`* \;

Carries out *`COMMAND`* on each file that **find** scores a hit on. *`COMMAND`* terminates with \; (the ; is escaped to make certain the shell passes it to **find** literally, which concludes the command sequence). If *`COMMAND`* contains {}, then **find** substitutes the full path name of the selected file.

Example 3−47. Badname, eliminate file names in current directory containing bad characters and white space.

```
#!/bin/bash
```

```
# Delete filenames in current directory containing bad characters.

for filename in *
do
badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# Files containing those nasties:   + { ; " \ = ? ~ ( ) < > & * | $
rm $badname 2>/dev/null
#           So error messages deep-sixed.
done

# Now, take care of files containing all manner of whitespace.
find . -name "* *" -exec rm -f {} \;
# The path name of the file that "find" finds replaces the "{}".
# The '\' ensures that the ';' is interpreted literally, as end of command.

exit 0
```

See the man page for **find** for more detail.

xargs

A filter for feeding arguments to a command, and also a tool for assembling the commands themselves. It breaks a data stream into small enough chunks for filters and commands to process. Consider it as a powerful replacement for backquotes. In situations where backquotes fail with a too many arguments error, substituting **xargs** often works. Normally, xargs reads from 'stdin' or from a pipe, but it can also be given the output of a file.

`ls | xargs -p -l gzip` gzips every file in current directory, one at a time, prompting before each operation.

One of the more interesting xargs options is `-n` *XX*, which limits the number of arguments passed to *XX*.

`ls | xargs -n 8 echo` lists the files in the current directory in `8` columns.

> **Note:** The default command for **xargs** is **echo**.

Example 3−48. Log file using xargs to monitor system log

```
#!/bin/bash

# Generates a log file in current directory
# from the tail end of /var/log messages.

# Note: /var/log/messages must be readable by ordinary users
#       if invoked by same (#root chmod 755 /var/log/messages).

( date; uname -a ) >>logfile
# Time and machine name
echo --------------------------------------------------------------------- >>logfile
tail -5 /var/log/messages | xargs |  fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0
```

Example 3−49. copydir, copying files in current directory to another, using xargs

```
#!/bin/bash

# Copy (verbose) all files in current directory
# to directory specified on command line.

if [ -z $1 ]
# Exit if no argument given.
then
  echo "Usage: `basename $0` directory-to-copy-to"
  exit 1
fi

ls . | xargs -i -t cp ./{} $1
# This is the exact equivalent of
# cp * $1

exit 0
```

eval arg1, arg2, ...

Translates into commands the arguments in a list (useful for code generation within a script).

Example 3−50. Showing the effect of eval

```
#!/bin/bash

y=`eval ls -l`
echo $y

y=`eval df`
echo $y
# Note that LF's not preserved

exit 0
```

Example 3−51. Forcing a log−off

```
#!/bin/bash

y=`eval ps ax | sed -n '/ppp/p' | awk '{ print $1 }'`
# Finding the process number of 'ppp'

kill -9 $y
# Killing it


# Restore to previous state...

chmod 666 /dev/ttyS3
# Doing a SIGKILL on ppp changes the permissions
# on the serial port. Must be restored.

rm /var/lock/LCK..ttyS3
# Remove the serial port lock file.
```

```
exit 0
```

expr arg1 operation arg2 ...

All−purpose expression evaluator: Concatenates and evaluates the arguments according to the operation given (arguments must be separated by spaces). Operations may be arithmetic, comparison, string, or logical.

expr 3 + 5

returns 8

expr 5 % 3

returns 2

y=`expr $y + 1`

incrementing variable, same as **`let y=y+1`** and **`y=$(($y+1))`**, as discussed elsewhere

z=`expr substr $string28 $position $length`

Note that external programs, such as **sed** and **Perl** have far superior string parsing facilities, and it might well be advisable to use them instead of the built−in bash ones.

Example 3−52. Using expr

```
#!/bin/bash

# Demonstrating some of the uses of 'expr'
# +++++++++++++++++++++++++++++++++++++++

echo

# Arithmetic Operators

echo Arithmetic Operators
echo
a=`expr 5 + 3`
echo 5 + 3 = $a

a=`expr $a + 1`
echo
echo a + 1 = $a
echo \(incrementing a variable\)

a=`expr 5 % 3`
# modulo
echo
echo 5 mod 3 = $a

echo
echo

# Logical Operators

```

```
echo Logical Operators
echo

a=3
echo a = $a
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, therefore...'
echo "If a > 10, b = 0 (false)"
echo b = $b

b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo b = $b


echo
echo

# Comparison Operators

echo Comparison Operators
echo
a=zipper
echo a is $a
if [ `expr $a = snap` ]
# Force re-evaluation of variable 'a'
then
   echo "a is not zipper"
fi

echo
echo



# String Operators

echo String Operators
echo

a=1234zipper43231
echo The string being operated upon is $a.

# index: position of substring
b=`expr index $a 23`
echo Numerical position of first 23 in $a is $b.

# substr: print substring, starting position & length specified
b=`expr substr $a 2 6`
echo Substring of $a, starting at position 2 and 6 chars long is $b.

# length: length of string
b=`expr length $a`
echo Length of $a is $b.

# 'match' operations similarly to 'grep'
b=`expr match $a [0-9]*`
echo Number of digits at the beginning of $a is $b.
b=`expr match $a '\([0-9]*\)'`
echo The digits at the beginning of $a are $b.

echo
```

```
exit 0
```

Note that : can substitute for **match**. **b=`expr $a : [0−9]*`** is an exact equivalent of **b=`expr match $a [0−9]*`** in the above example.

let

The **let** command carries out arithmetic operations on variables. In many cases, it functions as a less complex version of **expr**.

Example 3−53. Letting let do some arithmetic.

```
#!/bin/bash

echo

let a=11
# Same as 'a=11'
let a=a+5
# Equivalent to let "a = a + 5"
# (double quotes makes it more readable)
echo "a = $a"
let "a <<= 3"
# Equivalent of let "a = a << 3"
echo "a left−shifted 3 places = $a"

let "a /= 4"
# Equivalent to let "a = a / 4"
echo $a
let "a −= 5"
# Equivalent to let "a = a − 5"
echo $a
let "a = a * 10"
echo $a
let "a %= 8"
echo $a

exit 0
```

3.10.3. Time / Date Commands

date

Simply invoked, **date** prints the date and time to stdout. Where this command gets interesting is in its formatting and parsing options.

Example 3−54. Using date

```
#!/bin/bash

#Using the 'date' command
```

```
# Needs a leading '+' to invoke formatting.

echo "The number of days since the year's beginning is `date +%j`."
# %j gives day of year.


echo "The number of seconds elapsed since 01/01/1970 is `date +%s`."
# %s yields number of seconds since "UNIX epoch" began,
# but how is this useful?

prefix=temp
suffix=`eval date +%s`
filename=$prefix.$suffix
echo $filename
# It's great for creating "unique" temp filenames,
# even better than using $$.

# Read the 'date' man page for more formatting options.

exit 0
```

time

Outputs very verbose timing statistics for executing a command.

`time ls -l /` gives something like this:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

See also the very similar **times** command in the previous section.

touch

Utility for updating access/modification times of a file to current system time or other specified time, but also useful for creating a new file. The command **`touch zzz`** will create a new file of zero length, named `zzz`, assuming that `zzz` did not previously exist. Time−stamping empty files in this way is useful for storing date information, for example in keeping track of modification times on a project. See Example 3−11.

at

The **at** job control command executes a given set of commands at a specified time. This is a user version of **cron**.

`at 2pm January 15` prompts for a set of commands to execute at that time. These commands may include executable shell scripts.

Using either the `-f` option or input redirection (<), **at** reads a command list from a file. This file can include shell scripts, though they should, of course, be noninteractive.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

The **batch** job control command is similar to **at**, but it runs a command list when the system load drops below `.8`. Like **at**, it can read commands from a file with the `−f` option.

cal

Prints a neatly formatted monthly calendar to `stdout`. Will do current year or a large range of past and future years.

sleep

This is the shell equivalent of a wait loop. It pauses for a specified number of seconds, doing nothing. This can be useful for timing or in processes running in the background, checking for a specific event every so often (see Example 3−101).

```
sleep 3
# Pauses 3 seconds.
```

3.10.4. Text Processing Commands

sort

File sorter, often used as a filter in a pipe. See the man page for options.

diff

Simple file comparison utility. This compares the target files line−by−line sequentially. In some applications, such as comparing word dictionaries, it may be helpful to filter the files through **sort** and **uniq** before piping them to **diff**. **`diff file−1 file−2`** outputs the lines in the files that differ, with carets showing which file each particular line belongs to. A common use for **diff** is generating difference files to be used with **patch** (see below). The `−e` option outputs files suitable for **ed** or **ex** scripts.

```
patch -p1 <patch-file
# Takes all the changes listed in 'patch-file' and applies them
# to the files referenced therein.

cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Upgrading kernel source using 'patch'.
# From the Linux kernel docs "README",
# by anonymous author (Alan Cox?).
```

There are available various fancy frontends for **diff**, such as **spiff**, **wdiff**, **xdiff**, and **mgdiff**.

comm

Versatile file comparison utility. The files must be sorted for this to be useful.

comm ***`−options first−file second−file`***

`comm file−1 file−2` outputs three columns:

- ♦ column 1 = lines unique to `file−1`
- ♦ column 2 = lines unique to `file−2`
- ♦ column 3 = lines common to both.

The options allow suppressing output of one or more columns.

- ♦ `−1` suppresses column `1`
- ♦ `−2` suppresses column `2`
- ♦ `−3` suppresses column `3`
- ♦ `−12` suppresses both columns `1` and `2`, etc.

uniq

This filter removes duplicate lines from a sorted file. It is often seen in a pipe coupled with **sort**.

```
cat list−1 list−2 list−3 | sort | uniq > final.list
# Concatenates the list files,
# sorts them,
# removes duplicate lines,
# and finally writes the result to an output file.
```

expand

A filter than converts tabs to spaces, often seen in a pipe.

cut

A tool for extracting fields from files. It is similar to the **`print $N`** command set in **awk**, but more limited. It may be simpler to use **cut** in a script than **awk**. Particularly important are the `−d` (delimiter) and `−f` (field specifier) options.

Using **cut** to obtain a listing of the mounted filesystems:

```
cat /etc/mtab | cut −d ' ' −f1,2
```

Using **cut** to list the OS and kernel version:

```
uname −a | cut −d" " −f1,3,11,12
```

`cut −d ' ' −f2,3 filename` is equivalent to **`awk '{ print $2, $3 }' filename`**

colrm

Column removal filter. This removes columns (characters) from a file and writes them, lacking the range of specified columns, back to stdout. **`colrm 2 4 <filename`** removes the second through fourth characters from each line of the text file `filename`.

Warning
If the file contains tabs or nonprintable characters, this may cause unpredictable behavior.

paste

Tool for merging together different files into a single, multi−column file. In combination with **cut**, useful for creating system log files.

join

Consider this a more flexible version of **paste**. It works on exactly two files, but permits specifying which fields to paste together, and in which order.

head

lists the first `10` lines of a file to stdout (see Example 3−66).

tail

lists the end of a file to stdout (the default is `10` lines, but this can be changed). Commonly used to keep track of changes to a system logfile, using the `-f` option, which outputs lines appended to the file.

Example 3−48, Example 3−66, and Example 3−101 show *tail* in action.

grep

A multi−purpose file search tool that uses regular expressions. Originally a command/filter in the ancient **ed** line editor, **`g/re/p`**, or *global − regular expression − print*.

grep *`pattern`* [*`file`*...]

search the files `file`, etc. for occurrences of *`pattern`*.

`ls -l | grep '.txt'` has the same effect as **`ls -l *.txt`**.

The *−i* option to **grep** causes a case−insensitive search.

Example 3−101 demonstrates how to use *grep* to search for a keyword in a system log file.

Example 3−55. Emulating "grep" in a script

```
#!/bin/bash

# Very crude reimplementation of 'grep'.

if [ -z $1 ]  # Check for argument to script.
then
  echo "Usage: `basename $0` pattern"
  exit 1
fi

echo

for file in *     # Traverse all files in $PWD.
do
  output=$(sed -n /"$1"/p $file)  # Command substitution.

  if [ ! -z "$output" ]  # Variable $file quoted, otherwise error on multi-line output.
```

```
  then
    echo -n "$file: "
    echo $output
  fi

  echo
done

echo

exit 0

# Exercises for reader:
# -------------------
# 1) Add newlines to output, if more than one match in any given file.
# 2) Add features.
```

> **Note: egrep** is the same as **grep −E**. This uses a somewhat different, extended set of regular expressions, which may make the search somewhat more flexible.

> **Note: fgrep** is the same as **grep −F**. It does a literal string search (no regular expressions), which generally speeds things up quite a bit.

> **Note:** To search compressed files, use **zgrep**. It also works on non−compressed files, though slower than plain **grep**. This is handy for searching through a mixed set of files, some of them compressed, some not.

look

The command **look** works like **grep**, but does a lookup on a "dictionary", a sorted word list. By default, **look** searches for a match in `/usr/dict/words`, but a different dictionary file may be specified.

Example 3−56. Checking words in a list for validity

```
#!/bin/bash
# lookup:
# Does a dictionary lookup on each word in a data file.

file=words.data  # Data file to read words to test from.

echo

while [ "$word" != end ]  # Last word in data file.
do
  read word   # From data file, because of redirection at end of loop.
  look $word > /dev/null  # Don't want to display lines in dictionary file.
  lookup=$?   # Exit value of 'look'.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" is valid."
  else
    echo "\"$word\" is invalid."
  fi

done <$file  # Redirects stdin to $file, so "reads" come from there.
```

```
echo

exit 0
```

sed, awk

Scripting languages especially suited for parsing text files and command output. May be embedded singly or in combination in pipes and shell scripts.

sed

Non−interactive "stream editor", permits using many **ex** commands in batch mode. It finds many uses in shell scripts. See Appendix B.

awk

Programmable file extractor and formatter, good for manipulating and/or extracting fields (columns) in structured text files. Its syntax is similar to C. See Section B.2.

groff, gs, TeX

Text markup languages. Used for preparing copy for printing or formatted video display.

Man pages use **groff** (see Example A−1). *Ghostscript* (**gs**) is the GPL version of Postscript. **TeX** is Donald Knuth's elaborate typsetting system. It is often convenient to write a shell script encapsulating all the options and arguments passed to one of these markup languages.

wc

wc gives a "word count" on a file or I/O stream:

```
$ wc /usr/doc/sed-3.02/README
20     127     838 /usr/doc/sed-3.02/README
[20 lines  127 words  838 characters]
```

`wc -w` gives only the word count.

`wc -l` gives only the line count.

`wc -c` gives only the character count.

`wc -L` gives only the length of the longest line.

Using *wc* to count how many *.txt* files are in current working directory:

```
$ ls *.txt | wc -l
```

See Example 3−66 and Example 3−75.

tr

character translation filter.

Caution
must use quoting and/or brackets, as appropriate.

tr "A−Z" "*" <filename changes all the uppercase letters in `filename` to asterisks (writes to stdout).

tr −d [0−9] <filename deletes all digits from the file `filename`.

Example 3−57. toupper: Transforms a file to all uppercase.

```
#!/bin/bash

# Changes a file to all uppercase.

if [ -z $1 ]
# Standard check whether command line arg is present.
then
  echo "Usage: `basename $0` filename"
  exit 1
fi

tr [a-z] [A-Z] <$1

exit 0
```

Example 3−58. lowercase: Changes all filenames in working directory to lowercase.

```
#! /bin/bash
#
# Changes every filename in working directory to all lowercase.
#
# Inspired by a script of john dubois,
# which was translated into into bash by Chet Ramey,
# and considerably simplified by Mendel Cooper,
# author of this HOWTO.


for filename in *  #Traverse all files in directory.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`  #Change name to lowercase.
   if [ $fname != $n ]  # Rename only files not already lowercase.
   then
     mv $fname $n
   fi
done

exit 0
```

Example 3−59. rot13: rot13, ultra−weak encryption.

```
#!/bin/bash
```

```
# Classic rot13 algorithm, encryption that might fool a 3-year old.
# Usage: ./rot13.sh filename
# or     ./rot13.sh <filename
# or     ./rot13.sh and supply keyboard input (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" goes to "n", "b" to "o", etc.
# The 'cat "$@"' construction permits getting input either from stdin or from a file.

exit 0
```

fold

A filter that wraps inputted lines to a specified width (see Example 3−62).

fmt

Simple−minded file formatter, used as a filter in a pipe to "wrap" long lines of text output (see Example 3−48 and Example 3−62).

ptx

The **ptx [targetfile]** command outputs a permuted index (cross−reference list) of the targetfile. This may be further filtered and formatted in a pipe, if necessary.

column

Column formatter. This filter transforms list−type text output into a "pretty−printed" table by inserting tabs at appropriate places.

Example 3−60. Using column to format a directory listing

```
#!/bin/bash
# This is a slight modification of the example file in the "column" man page.


(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t

# The "sed 1d" in the pipe deletes the first line of output,
# which would be "total        N",
# where "N" is the total number of files found by "ls -l".

# The -t option to "column" pretty-prints a table.

exit 0
```

nl

Line numbering filter. **`nl filename`** lists `filename` to stdout, but inserts consecutive numbers at the beginning of each non−blank line. If `filename` omitted, operates on stdin.

Example 3−61. nl: A self−numbering script.

```
#!/bin/bash

# This file echoes itself twice to stdout with its lines numbered.

# 'nl' sees this as line 3 since it does not number blank lines.
# 'cat -n' sees the above line as number 5.

nl `basename $0`

echo; echo  # Now, let's try it with 'cat -n'

cat -n `basename $0`
# The difference is that 'cat -n' numbers the blank lines.

exit 0
```

pr

Print formatting filter. This will paginate a file (or stdout) into sections suitable for hard copy printing. A particularly useful option is `-d`, forcing double−spacing.

Example 3−62. Formatted file listing.

```
#!/bin/bash

# Get a file listing...

b=`ls /usr/local/bin`

# ...40 columns wide.
echo $b | fmt -w 40

# Could also have been done by
# echo $b | fold - -s -w 40

exit 0
```

printf

The **printf**, formatted print, command is an enhanced **echo**. It is a limited variant of the C language `printf`, and the syntax is somewhat different.

printf *`format-string`*... *`parameter`*...

See the **printf** man page for in−depth coverage.

Caution
Older versions of **bash** may not support **printf**.

Example 3−63. printf in action

```
#!/bin/bash
```

```
# printf demo

PI=3.14159265358979
DecimalConstant=31373
Message1="Greetings,"
Message2="Earthling."

echo

printf "Pi to 2 decimal places = %1.2f" $PI
echo
printf "Pi to 9 decimal places = %1.9f" $PI
# Note correct round off.

printf "\n"
# Prints a line feed, equivalent to 'echo'.

printf "Constant = \t%d\n" $DecimalConstant
# Insert tab (\t)

printf "%s %s \n" $Message1 $Message2

echo

exit 0
```

3.10.5. File and Archiving Commands

tar

The standard UNIX archiving utility. Originally a *Tape ARchiving* program, from whence it derived its name, it has developed into a general purpose package that can handle all manner of archiving with all types of destination devices, ranging from tape drives to regular files to even stdout (see Example 3−4). GNU tar has long since been patched to accept **gzip** compression options, such as **tar czvf archive−name.tar.gz ***, which recursively archives and compresses all files (except "dotfiles") in a directory tree.

cpio

This specialized archiving copy command is rarely used any more, having been supplanted by **tar/gzip**. It still has its uses, such as moving a directory tree.

Example 3−64. Using cpio to move a directory tree

```
#!/bin/bash

# Copying a directory tree using cpio.

if [ $# -ne 2 ]
then
  echo Usage: `basename $0` source destination
  exit 1
fi
```

```
source=$1
destination=$2

find "$source" -depth | cpio -admvp "$destination"

exit 0
```

gzip

The standard GNU/UNIX compression utility, replacing the inferior and proprietary **compress**. The corresponding decompression command is **gunzip**, which is the equivalent of **gzip −d**.

The filter **zcat** decompresses a *gzipped* file to stdout, as possible input to a pipe or redirection. This is, in effect, a **cat** command that works on compressed files (including files processed with the older **compress** utility). See Example 3−14.

bzip2

An alternate compression utility, usually more efficient than **gzip**, especially on large files. The corresponding decompression command is **bunzip2**.

sq

Yet another compression utility, a filter that works only on sorted ASCII word lists. It uses the standard invocation syntax for a filter, **sq < input−file > output−file**. Fast, but not nearly as efficient as **gzip**. The corresponding uncompression filter is **unsq**, invoked like **sq**.

> **Note:** The output of **sq** may be piped to **gzip** for further compression.

shar

Shell archiving utility. The files in a shell archive are concatenated without compression, and the resultant archive is essentially a shell script, complete with #!/bin/sh header, and containing all the necessary unarchiving commands. Shar archives still show up in Internet newsgroups, but otherwise **shar** has been pretty well replaced by **tar/gzip**. The **unshar** command unpacks **shar** archives.

split

Utility for splitting a file into smaller chunks. Usually used for splitting up large files in order to back them up on floppies or preparatory to e−mailing or uploading them.

file

A utility for identifying file types. The command **`file file−name`** will return a file specification for `file−name`, such as `ascii text` or `data`. It references the magic numbers found in `/usr/share/magic`, `/etc/magic`, or `/usr/lib/magic`, depending on the Linux/UNIX distribution.

Example 3−65. stripping comments from C program files

```
#!/bin/bash
```

```
# Strips out the comments (/* comment */) in a C program.

NOARGS=1
WRONG_FILE_TYPE=2

if [ $# = 0 ]
then
  echo "Usage: `basename $0` C-program-file" >&2 # Error message to stderr.
  exit $NOARGS
fi

# Test for correct file type.
type=`eval file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" echoes file type...
# then awk removes the first field of this, the filename...
# then the result is fed into the variable "type".
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "This script works on C program files only."
  echo
  exit $WRONG_FILE_TYPE
fi


# Rather cryptic sed script:
#--------
sed '
/^\/\*/d
/.*\/\*/d
' $1
#--------
# Easy to understand if you take several hours to learn sed fundamentals.


# Need to add one more line to the sed script to deal with
# case where line of code has a comment following it on same line.
# This is left as a non-trivial exercise for the reader.

exit 0
```

uuencode

This utility encodes binary files into ASCII characters, making them suitable for transmission in the body of an e−mail message or in a newsgroup posting.

uudecode

This reverses the encoding, decoding uuencoded files back into the original binaries.

Example 3−66. uudecoding encoded files

```
#!/bin/bash

lines=35
```

```
# Allow 35 lines for the header (very generous).

for File in *
# Test all the files in the current working directory...
do
  search1=`head -$lines $File | grep begin | wc -w`
  search2=`tail -$lines $File | grep end | wc -w`
  # Uuencoded files have a "begin" near the beginning, and an "end" near the end.
  if [ $search1 -gt 0 ]
  then
    if [ $search2 -gt 0 ]
    then
      echo "uudecoding - $File -"
      uudecode $File
    fi
  fi
done

exit 0
```

sum*, *cksum*, *md5sum

These are utilities for generating checksums. A *checksum* is a number mathematically calculated from the contents of a file, for the purpose of checking its integrity. A script might refer to a list of checksums for security purposes, such as ensuring that the contents of key system files have not been altered or corrupted.

strings

Use the **strings** command to find printable strings in a binary or data file. It will list sequences of printable characters found in the target file. This might be handy for a quick 'n dirty examination of a core dump or for looking at an unknown graphic image file (**`strings image-file | more`** might show something like `JFIF`, which would identify the file as a *jpeg* graphic). In a script, you would probably parse the output of **strings** with **grep** or **sed**.

more*, *less

Pagers that display a text file or stream to `stdout`, one screenful at a time. These may be used to filter the output of a script.

3.10.6. Communications Commands

host

Searches for information about an Internet host by name or IP address, using DNS.

vrfy

Verify an Internet e−mail address.

nslookup

Do an Internet "name server lookup" on a host by IP address. This may be run either interactively or noninteractively, i.e., from within a script.

dig

Similar to **nslookup**, do an Internet "name server lookup" on a host. May be run either interactively or noninteractively, i.e., from within a script.

traceroute

Trace the route taken by packets sent to a remote host. This command works within a LAN, WAN, or over the Internet. The remote host may be specified by an IP address. The output of this command may be filtered by **grep** or **sed** in a pipe.

rcp

"Remote copy", copies files between two different networked machines. Using **rcp** and similar utilities with security implications in a shell script may not be advisable. Consider instead, using an **expect** script.

sx, rx

The **sx** and **rx** command set serves to transfer files to and from a remote host using the *xmodem* protocol. These are generally part of a communications package, such as **minicom**.

sz, rz

The **sz** and **rz** command set serves to transfer files to and from a remote host using the *zmodem* protocol. *Zmodem* has certain advantages over *xmodem*, such as greater transmission rate and resumption of interrupted file transfers. Like **sx** and **rx**, these are generally part of a communications package.

uucp

UNIX to UNIX copy. This is a communications package for transferring files between UNIX servers. A shell script is an effective way to handle a **uucp** command sequence.

Since the advent of the Internet and e−mail, **uucp** seems to have faded into obscurity, but it still exists and remains perfectly workable in situations where an Internet connection is not available or appropriate.

3.10.7. Miscellaneous Commands

jot, seq

These utilities emit a sequence of integers, with a user selected increment. This can be used to advantage in a **for** loop.

Example 3−67. Using seq to generate loop arguments

```
#!/bin/bash
```

```
for a in `seq 80`
# Same as   for a in 1 2 3 4 5 ... 80   (saves much typing!).
# May also use 'jot' (if present on system).
do
  echo -n "$a "
done

echo

exit 0
```

which

which <command−xxx> gives the full path to "command−xxx". This is useful for finding out whether a particular command or utility is installed on the system.

`$bash which pgp`

```
/usr/bin/pgp
```

script

This utility records (saves to a file) all the user keystrokes at the command line in a console or an xterm window. This, in effect, create a record of a session.

lp

The **lp** and **lpr** commands send file(s) to the print queue, to be printed as hard copy. [4] These commands trace the origin of their names to the line printers of another era.

`bash$` **`cat file1.txt | lp`**

It is often useful to pipe the formatted output from **pr** to **lp**.

`bash$` **`pr -options file1.txt | lp`**

Formatting packages, such as **groff** and *Ghostscript* may send their output directly to **lp**.

`bash$` **`groff -Tascii file.tr | lp`**

`bash$` **`gs -options | lp file.ps`**

Related commands are **lpq**, for viewing the print queue, and **lprm**, for removing jobs from the print queue.

tee

[UNIX borrows an idea here from the plumbing trade.]

This is a redirection operator, but with a difference. Like the plumber's *tee*, it permits "siponing off" the output of a command or commands within a pipe, but without affecting the result. This is useful for printing an ongoing process to a file or paper, perhaps to keep track of it for debugging purposes.

```
                    tee
                  |------> to file
                  |
  ===============|===============
  command--->----|-operator-->---> result of command(s)
  ===============================
```

```
cat listfile* | sort | tee check.file | uniq > result.file
```

(The file `check.file` contains the concatenated sorted "listfiles", before the duplicate lines are removed by **uniq**.)

clear

The **clear** command simply clears the text screen at the console or in an xterm. The prompt and cursor reappear at the upper lefthand corner of the screen or xterm window. This command may be used either at the command line or in a script. See Example 3−36.

yes

In its default behavior the **yes** command feeds a continuous string of the character `y` followed by a line feed to stdout. A **control−c** terminates the run. A different output string may be specified, as in **`yes different string`**, which would continually output `different string` to stdout. One might well ask the purpose of this. From the command line or in a script, the output of **yes** can be redirected or piped into a program expecting user input. In effect, this becomes a sort of poor man's version of **expect**.

mkfifo

This obscure command creates a *named pipe*, a temporary First−In−First−Out buffer for transferring data between processes. Typically, one process writes to the FIFO, and the other reads from it. See Example A−7.

pathchk

This command checks the validity of a filename. If the filename exceeds the maximum allowable length (255 characters) or one or more of the directories in its path is not searchable, then an error message results. Unfortunately, **pathchk** does not return a recognizable error code, and it is therefore pretty much useless in a script.

dd

This is the somewhat obscure and much feared "data duplicator" command. It simply copies a file (or stdin/stdout), but with conversions. Possible conversions are ASCII/EBCDIC, upper/lower case, swapping of byte pairs between input and output, and skipping and/or truncating the head or tail of the input file. A **`dd --help`** lists the conversion and other options that this powerful utility takes.

The **dd** command can copy raw data and disk images to and from devices, such as floppies and tape drives. It can even be used to create boot floppies.

```
dd if=kernel-image of=/dev/fd0H1440
```

One important use for **dd** is initializing temporary swap files (see Example 3−97).

3.11. System and Administrative Commands

The startup and shutdown scripts in `/etc/rc.d` illustrate the uses (and usefulness) of these comands. These are usually invoked by root and used for system maintenance or emergency filesystem repairs. Use with caution, as some of these commands may damage your system if misused.

uname

Output system specifications (OS, kernel version, etc.) to stdout. Invoked with the `−a` option, gives verbose system info (see Example 3−48).

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown
```

arch

Show system architecture. Equivalent to **uname −m**.

```
bash$ arch
i686
```

```
bash$ uname -m
i686
```

id

The **id** command lists the real and effective user IDs and the group IDs of the current user. This is the counterpart to the `$UID`, `$EUID`, and `$GROUPS` internal Bash variables (see Section 3.7).

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)
```

```
bash$ echo $UID
501
```

who

Show all users logged on to the system.

whoami is a variant of **who** that lists only the current user.

w

Show all logged on users and the processes belonging to them. This is an extended version of **who**. The output of **w** may be piped to **grep** to find a specific user and/or process.

```
bash$ w | grep startx
grendel  tty1     -                 4:22pm  6:41   4.47s  0.45s  startx
```

users

Show all logged on users. This is the approximate equivalent of **who −q**.

groups

Lists the current user and the groups she belongs to. This corresponds to the `$GROUPS` internal variable, but gives the group names, rather than the numbers.

```
bash$ groups
bozita cdrom cdwriter audio xgrp
```

```
bash$ echo $GROUPS
501
```

hostname

Lists the system's host name, as recorded in `/etc/hosts`. This is a counterpart to the `$HOSTNAME` internal variable.

```
bash$ hostname
localhost.localdomain
```

```
bash$ echo $HOSTNAME
localhost.localdomain
```

ulimit

Sets an *upper limit* on system resources. Usually invoked with the `−f` option, which sets a limit on file size (**ulimit −f 1000** limits files to 1 meg maximum). The `−t` option limits the coredump size (**ulimit −c 0** eliminates coredumps). Normally, the value of **ulimit** would be set in `/etc/profile` and/or `~/.bash_profile` (see Section 3.23).

uptime

Shows how long the system has been running, along with associated statistics.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

env

Runs a program or script with certain environmental variables set or changed (without changing the overall system environment). The `[varname=xxx]` permits changing the environmental variable `varname` for the duration of the script. With no options specified, this command lists all the environmental variable settings.

su

Runs a program or script as a *s*ubstitute *u*ser. **su rjones** starts a shell as user *rjones*. A naked **su** defaults to *root*. See Example A−7.

shopt

This command permits changing shell options on the fly (see Example 3−85). It often appears in the Bash setup files, but also has its uses in scripts. Works with version 2 of Bash only.

```
shopt -s cdspell
# Allows minor misspelling directory names with 'cd'
command.
```

lockfile

This utility is part of the **procmail** package (www.procmail.org). It creates a *lock file*, a semaphore file that controls access to a file, device, or resource. The lock file serves as a flag that this particular file, device, or resource is in use by a particular process ("busy"), and permitting only restricted access (or no access) to other processes. Lock files are used in such applications as protecting system mail folders from simultaneously being changed by multiple users, indicating that a modem port is being accessed, and showing that an instance of Netscape is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts create a lock file that already exists, the script will likely hang.

cron

Administrative program scheduler, performing such duties as cleaning up and deleting system log files and updating the slocate database. This is the superuser version of **at**. It runs as a daemon (background process) and executes scheduled entries from `/etc/crontab`.

chroot

CHange ROOT directory. Normally commands are fetched from `$PATH`, relative to `/`, the default root directory. This changes the root directory to a different one (and also changes the working directory to there). A **`chroot /opt`** would cause references to `/usr/bin` to be translated to `/opt/usr/bin`, for example. This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those telnetting in, to a secured portion of the filesystem. Note that after a **chroot**, the execution path for system binaries is no longer valid.

The **chroot** command is also handy when running from an emergency boot floppy (**chroot** to `/dev/fd0`), or as an option to **lilo** when recovering from a system crash. Other uses include installation from a different filesystem (an **rpm** option). Invoke only as root, and use with caution.

umask

User file creation MASK. Limit the default file attributes for a particular user. All files created by that user take on the attributes specified by **umask**. The (octal) value passed to **umask** defines the the file permissions *disabled*. For example, **umask 022** ensures that new files will have at most 755 permissions (777 NAND 022). [5] Of course, the user may later change the attributes of particular files with **chmod**.The usual practice is to set the value of **umask** in `/etc/profile` and/or `~/.bash_profile` (see Section 3.23).

ldd

Show shared lib dependencies for an executable file.

```
bash$ ldd /bin/ls
```

```
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

logname

Show current user's login name (as found in `/var/run/utmp`). This is equivalent to **whoami**, above.

```
bash$ logname
bozo
```

```
bash$ whoami
bozo
```

tty

Echoes the name of the current user's terminal. Note that each separate xterm window counts as a different terminal.

```
bash$ tty
/dev/pts/1
```

stty

Shows and/or changes terminal settings.

Example 3−68. secret password: Turning off terminal echoing

```
#!/bin/bash

echo
echo -n "Enter password "
read passwd
echo "password is $passwd"
echo -n "If someone had been looking over your shoulder, "
echo "your password would have been compromised."

echo && echo  # Two line-feeds in an "and list".

stty -echo   # Turns off screen echo.

echo -n "Enter password again "
read passwd
echo
echo "password is $passwd"
echo

stty echo   # Restores screen echo.

exit 0
```

wall

This is an acronym for "write all", i.e., sending a message to all users every terminal logged on in the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem (see Example 3−94).

```
wall System going down for maintenance in 5 minutes!
```

logger

Appends a user−generated message to the system log (`/var/log/messages`). You do not have to be root to invoke **logger**.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# Now, do a 'tail /var/log/messages'.
```

dmesg

Lists all system bootup messages to stdout. Handy for debugging and ascertaining which device drivers were installed and which system interrupts in use. The output of **dmesg** may, of course, be parsed with **grep**, **sed**, or **awk** from within a script.

fuser

Identifies the processes (by pid) that are accessing a given file, set of files, or directory. May also be invoked with the `-k` option, which kills those processes. This has interesting implications for system security, especially in scripts preventing unauthorized users from accessing system services.

pidof

Identifies *process id (pid)* of a running job. Job control commands, such as **kill** and **renice** act on the *pid* of a process, rather than its name. This is the counterpart of the `$PPID` internal variable (see Section 3.7).

Example 3−69. pidof helps kill a process

```
#!/bin/bash
# kill-process

NOPROCESS=2

process=xxxyyyzzz  # Use nonexistent process.
# For demo purposes only...
# ... don't want to actually kill any actual process with this script.
# If, for example, you wanted to use this script to logoff the Internet     process=pppd

t=`pidof $process`       # Find pid (process id) of $process.
# The pid is needed by 'kill' (can't 'kill' by program name).

if [ -z $t ]    # If process not present, 'pidof' returns null.
then
  echo "Process $process was not running."
  echo "Nothing killed."
  exit $NOPROCESS
fi

kill $t     # May need 'kill -9' for stubborn process.

# Need a check here to see if process allowed itself to be killed.
```

```
# Perhaps another " t=`pidof $process` ".

exit 0
```

nice

Show or change the priority of a background job. Priorities run from 19 (lowest) to −20 (highest). Only *root* may set the negative (higher) priorities. Related commands are **renice**, **snice**, and **skill**.

nohup

Keeps a command running even after user logs off. The command will run as a foreground process unless followed by &. If you use **nohup** within a script, consider coupling it with a **wait** to avoid creating an orphan or zombie process.

free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using **grep**, **awk** or **Perl**.

```
bash$ free
                total       used       free     shared    buffers     cached
   Mem:         30504      28624       1880      15820       1608      16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

sync

Forces an immediate write of all updated data from buffers to hard drive. While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure. In the olden days, a **`sync sync`** was a useful precautionary measure before a system reboot.

init

The **init** command is the parent of all processes. Called in the final step of a bootup, **init** determines the runlevel of the system from `/etc/inittab`. Invoked by its alias **telinit**, and by root only.

telinit

Symlinked to **init**, this is a means of changing the system runlevel, usually done for system maintenance or emergency filesystem repairs. Invoked only by root. This command can be dangerous − be certain you understand it well before using!

runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel `0`), in single−user mode (`1`), in multi−user mode (`2` or `3`), in X Windows (`5`), or rebooting (`6`).

halt, shutdown, reboot

Command set to shut the system down, usually just prior to a power down.

exec

This is actually a system call that replaces the current process with a specified command. It is mostly seen in combination with **find**, to execute a command on the files found (see Example 3−47). When used as a standalone in a script, it forces an exit from the script when the **exec**'ed command terminates. An **exec** is also used to reassign file descriptors. **`exec <zzz-file`** replaces stdin with the file `zzz-file` (see Example 3−72).

Example 3−70. Effects of exec

```
#!/bin/bash

exec echo "Exiting $0."
# Exit from script.

# The following lines never execute.
echo "Still here?"

exit 0
```

ifconfig

Network interface configuration utility.

route

Show info about or make changes to the kernel routing table.

netstat

Show current network information and statistics, such as routing tables and active connections.

mknod

Creates block or character device files (may be necessary when installing new hardware on the system).

mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file `/etc/fstab` provides a handy listing of available filesystems, including options, that may be automatically or manually mounted. The file `/etc/mtab` shows the currently mounted filesystems (including the virtual ones, such as `/proc`).

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Mounts CDROM
mount /mnt/cdrom
# Shortcut, if /mnt/cdrom listed in /etc/fstab
```

umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be **umount**'ed, else filesystem corruption may result.

```
umount /mnt/cdrom
```

lsmod

List installed kernel modules.

insmod

Force insertion of a kernel module. Must be invoked as root.

modprobe

Module loader that is normally invoked automatically in a startup script.

depmod

Creates module dependency file, usually invoked from startup script.

rdev

Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but **rdev** remains useful for setting up a ram disk. This is another dangerous command, if misused.

Using our knowledge of administrative commands, let us examine a system script. One of the shortest and simplest to understand scripts is **killall**, used to suspend running processes at system shutdown.

Example 3−71. killall, from `/etc/rc.d/init.d`

```
#!/bin/sh

# --> Comments added by the author of this HOWTO marked by "-->".

# --> This is part of the 'rc' script package
# --> by Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>

# --> This particular script seems to be Red Hat specific
# --> (may not be present in other distributions).

# Bring down all unneeded services that are still running (there shouldn't
# be any, so this is just a sanity check)

for i in /var/lock/subsys/*; do
        # --> Standard for/in loop, but since "do" is on same line,
        # --> it is necessary to add ";".
        # Check if the script is there.
        [ ! -f $i ] && continue
        # --> This is a clever use of an "and list", equivalent to:
        # --> if [ ! -f $i ]; then continue

        # Get the subsystem name.
        subsys=${i#/var/lock/subsys/}
        # --> Match variable name, which, in this case, is the file name.
        # --> This is the exact equivalent of subsys=`basename $i`.
```

```
        # --> It gets it from the lock file name, and since if there
        # --> is a lock file, that's proof the process has been running.
        # --> See the "lockfile" entry, above.


        # Bring the subsystem down.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
            /etc/rc.d/init.d/$subsys.init stop
        else
            /etc/rc.d/init.d/$subsys stop
        # --> Suspend running jobs and daemons
        # --> using the 'stop' shell builtin.
        fi
done
```

That wasn't so bad. Aside from a little fancy footwork with variable matching, there is no new material there.

Exercise. In `/etc/rc.d/init.d`, analyze the **halt** script. It is a bit longer than **killall**, but similar in concept. Make a copy of this script somewhere in your home directory and experiment with it (do *not* run it as root). Do a simulated run with the `-vn` flags (**`sh -vn scriptname`**). Add extensive comments. Change the "action" commands to "echos".

Now, look at some of the more complex scripts in `/etc/rc.d/init.d`. See if you can understand parts of them. Follow the above procedure to analyze them. For some additional insight, you might also examine the file `sysvinitfiles` in `/usr/doc/initscripts-X.XX`, which is part of the "initscripts" documentation.

3.12. Backticks (`COMMAND`)

Command substitution

Commands within backticks generate command line text.

The output of commands within backticks can be used as arguments to another command or to load a variable.

```
rm `cat filename`
# "filename" contains a list of files to delete.

textfile_listing=`ls *.txt`
# Variable contains names of all *.txt files in current working directory.
echo $textfile_listing
#
textfile_listing2=$(ls *.txt)
echo $textfile_listing
# Also works.
```

Note: Using backticks for command substitution has been superseded by the **$(COMMAND)** form.

Arithmetic expansion (commonly used with expr)

```
z=`expr $z + 3`
```

Note that this particular use of backticks has been superseded by double parentheses **$((...))** or the very convenient **let** construction.

```
z=$(($z+3))
# $((EXPRESSION)) is arithmetic expansion.
# Not to be confused with command substitution.

let z=z+3
let "z += 3"  #If quotes, then spaces and special operators allowed.
```

All these are equivalent. You may use whichever one "rings your chimes".

3.13. I/O Redirection

There are always three default "files" open, *stdin* (the keyboard), *stdout* (the screen), and *stderr* (error messages output to the screen). These, and any other open files, can be redirected. Redirection simply means capturing the output of a file, command, program, script, or even code block within a script (see Example 3−2 and Example 3−3) and sending it as input to another file, command, program, or script.

Each open file gets assigned a file descriptor. [6] The file descriptors for *stdin*, *stdout*, and *stderr* are 0, 1, and 2, respectively. For opening additional files, there remain descriptors 3 to 9. It is sometimes useful to assign one of these additional file descriptors to *stdin*, *stdout*, or *stderr* as a temporary duplicate link. [7] This simplifies restoration to normal after complex redirection and reshuffling (see Example 3−72).

```
    >
      # Redirect stdout to a file.
      # Creates the file if not present, otherwise overwrites it.

      ls -lR > dir-tree.list
      # Creates a file containing a listing of the directory tree.

    >>
      # Redirect stdout to a file.
      # Creates the file if not present, otherwise appends to it.

    2> &1
      # Redirects stderr to stdout.
      # Has the effect of making visible error messages that might otherwise not be seen.

    i> &j
      # Redirects file descriptor i to j
      # All output of file pointed to by i gets sent to file pointed to by j

    <
      # Accept input from a file.
      # Companion command to ">", and often used in combination with it.
      grep search-word <filename

    |
      # Pipe.
      # General purpose process and command chaining tool.
      # Similar to ">", but more general in effect.
      # Useful for chaining commands, scripts, files, and programs together.
```

```
      cat *.txt | sort | uniq > result-file
      # Sorts the output of all the .txt files and deletes duplicate lines,
      # finally saves results to "result-file".
```

Note: Multiple instances of input and output redirection and/or pipes can be combined in a single command line.

```
command < input-file > output-file

command1 | command2 | command3 > output-file
```

n<&−

close input file descriptor *n*

<&−

close stdin

n>&−

close output file descriptor *n*

>&−

close stdout

The **exec <filename** command redirects *stdin* to a file. From that point on, all *stdin* comes from that file, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using **sed** and/or **awk**.

Example 3−72. Redirecting *stdin* using exec

```
#!/bin/bash
# Redirecting stdin using 'exec'.


exec 6<&0   # Link file descriptor #6 with stdin.

exec < data-file   # stdin replaced by file "data-file"

read a1   # Reads first line of file "data-file".
read a2   # Reads second line of file "data-file."

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6   # Now restore stdin from fd #6, where it had been saved.
```

```
echo -n "Enter data  "
read b1  # Now "read" functions as expected, reading from normal stdin.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"

echo

exit
```

Blocks of code, such as **while**, **until**, and **for** loops, even **if/then** test blocks can also incorporate redirection of *stdin*. The < operator at the the end of the code block accomplishes this.

Example 3−73. Redirected *while* loop

```
#!/bin/bash

if [ -z $1 ]
then
  Filename=names.data  # Default, if no filename specified.
else
  Filename="$1"
fi

while [ "$name" != Smith ]  # Why is variable $name in quotes?
do
  read name        # Reads from $Filename, rather than stdin.
  echo $name
done <$Filename   # Redirects stdin to file $Filename.

exit 0
```

Example 3−74. Redirected *until* loop

```
#!/bin/bash
# Same as previous example, but with "until" loop.

if [ -z $1 ]
then
  Filename=names.data  # Default, if no filename specified.
else
  Filename="$1"
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]     # Change  !=  to =.
do
  read name        # Reads from $Filename, rather than stdin.
  echo $name
done <$Filename   # Redirects stdin to file $Filename.

# Same results as with "while" loop in previous example.

exit 0
```

Example 3−75. Redirected *for* loop

```
#!/bin/bash

if [ -z $1 ]
then
  Filename=names.data  # Default, if no filename specified.
else
  Filename="$1"
fi

line_count=`wc $Filename | awk '{ print $1 }'`  # Number of lines in target file.
# Very contrived and kludgy, nevertheless shows that
# it's possible to redirect stdin within a "for" loop...
# if you're clever enough.


for name in `seq $line_count`  # Recall that "seq" prints sequence of numbers.
# while [ "$name" != Smith ]   --   more complicated than a "while" loop   --
do
  read name        # Reads from $Filename, rather than stdin.
  echo $name
  if [ "$name" = Smith ]   # Need all this extra baggage here.
  then
    break
  fi
done <$Filename   # Redirects stdin to file $Filename.

exit 0
```

Example 3−76. Redirected *if/then* test

```
#!/bin/bash

if [ -z $1 ]
then
  Filename=names.data  # Default, if no filename specified.
else
  Filename="$1"
fi

TRUE=1

if [ $TRUE ]
then
 read name
 echo $name
fi <$Filename
# Reads only first line of file.
# An if/then test has no way of iterating unless embedded in a loop.

exit 0
```

Clever use of I/O redirection permits parsing and stitching together snippets of files and command output. One possible application of this might be generating report and log files.

> **Note:** *Here documents* are a special case of I/O redirection. See Section 3.24.

3.14. Recess Time

This bizarre little intermission gives the reader a chance to relax and maybe laugh a bit.

Fellow Linux user, greetings! You are reading something which will bring you luck and good fortune. Just e−mail a copy of this document to 10 of your friends. Before you make the copies, send a 100−line Bash script to the first person on the list given at the bottom of this letter. Then delete their name and add yours to the bottom of the list.

Don't break the chain! Make the copies within 48 hours. Wilfred P. of Brooklyn failed to send out his ten copies and woke the next morning to find his job description changed to "COBOL programmer." Howard L. of Newport News sent out his ten copies and within a month had enough hardware to build a 100−node Beowulf cluster dedicated to playing xbill. Amelia V. of Chicago laughed at this letter and broke the chain. Shortly thereafter, a fire broke out in her terminal and she now spends her days writing documentation for MS Windows.

Don't break the chain! Send out your ten copies today!

Courtesy 'NIX "fortune cookies", with some alterations and many apologies

3.15. Regular Expressions

In order to fully utilize the power of shell scripting, you need to master regular expressions.

3.15.1. A Brief Introduction to Regular Expressions

An expression is a set of characters that has an interpretation above and beyond its literal meaning. A quote symbol ("), for example, may denote speech by a person, ditto, or a meta−meaning for the symbols that follow. Regular expressions are sets of characters that UNIX endows with special features.

The main uses for regular expressions (REs) are text searches and string manipulation. An RE *matches* a single character or a set of characters.

- The asterisk * matches any number of characters, *including zero*.
- The dot . matches any one character, except a newline.
- The question mark ? matches zero or one of the previous RE. It is generally used for matching single characters.

* The plus + matches one or more of the previous RE. It serves a role similar to the *, but does *not* match zero occurrences.
* The caret ^ matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.
* The dollar sign $ at the end of a an RE matches the end of a line.
* Brackets [...] enclose a set of characters to match in a single RE.

  `[xyz]` matches the characters `x`, `y`, or `z`.

  `[c−n]` matches any of the characters in the range `c` to `n`.

  `[^b−d]` matches all characters *except* those in the range `b` to `d`. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to `!` in a different context).

* The backslash \ escapes a special character, which means that character gets interpreted literally.

  A `\$` reverts back to its literal meaning of "dollar sign", rather than its RE meaning of end−of−line.

* Escaped "curly brackets" \{ \} indicate the number of occurrences of a preceding RE to match.

  It is necessary to escape the curly brackets since they have a different special character meaning otherwise.

  `[0−9]\{5\}` matches exactly five digits (characters in the range of 0 to 9).

Caution
Curly brackets are not available as an RE in *awk*.

"Sed & Awk", by Dougherty and Robbins (see *Bibliography*) gives a very complete and lucid treatment of REs.

3.15.2. Using REs in scripts

Sed, awk, and Perl, used as filters in scripts, take REs as arguments when "sifting" or transforming files or I/O streams. See Example A−4 and Example A−8 for illustrations of this.

3.16. Subshells

Running a shell script launches another instance of the command processor. Just as your commands are interpreted at the command line prompt, similarly does a script batch process a list of commands in a file. Each shell script running is, in effect, a subprocess of the parent shell, the one that gives you the prompt at the console or in an xterm window.

A shell script can also launch subprocesses. These *subshells* let the script do parallel processing, in effect executing multiple subtasks simultaneously.

* (command1; command2; command3; ...)

  A command list embedded between `parentheses` runs as a subshell.

  Note: Variables in a subshell are *not* visible outside the block of code in the subshell. These are, in effect, local variables.

Example 3−77. Variable scope in a subshell

```
#!/bin/bash

echo

outer_variable=Outer

(
inner_variable=Inner
echo "From subshell, \"inner_variable\" = $inner_variable"
echo "From subshell, \"outer\" = $outer_variable"
)

echo

if [ -z $inner_variable ]
then
  echo "inner_variable undefined in main body of shell"
else
  echo "inner_variable defined in main body of shell"
fi

echo "From main body of shell, \"inner_variable\" = $inner_variable"
# $inner_variable will show as uninitialized because
# variables defined in a subshell are "local variables".

echo

exit 0
```

Example 3−78. Running parallel processes in subshells

```
        (cat list1 list2 list3 | sort | uniq > list123)
        (cat list4 list5 list6 | sort | uniq > list456)
        # Merges and sorts both sets of lists simultaneously.

        wait #Don't execute the next command until subshells finish.

        diff list123 list456
```

  Note: A command block between `curly braces` does *not* launch a subshell.

  { command1; command2; command3; ... }

3.17. Restricted Shells

Running a script or portion of a script in *restricted* mode disables certain commands that would otherwise be available. This is a security measure intended to limit the privileges of the script user and to minimize possible damage from running the script.

Disabled commands in restricted shells

- Using `cd` to change the working directory.
- Changing the values of the `$PATH`, `$SHELL`, `$BASH_ENV`, or `$ENV` environmental variables.
- Reading or changing the `$SHELLOPTS`, shell environmental options.
- Output redirection.
- Invoking commands containing one or more /'s.
- Invoking *exec* to substitute a different process for the shell.
- Various other commands that would enable monkeying with or attempting to subvert the script for an unintended purpose.
- Getting out of restricted mode within the script.

Example 3−79. Running a script in restricted mode

```
#!/bin/bash
# Starting the script with "#!/bin/bash -r" runs entire script in restricted mode.

echo

echo "Changing directory."
cd /usr/local
echo "Now in `pwd`"
echo "Coming back home."
cd
echo "Now in `pwd`"
echo

# Everything up to here in normal, unrestricted mode.

set -r
# set --restricted    has same effect.
echo "==> Now in restricted mode. <=="

echo
echo

echo "Attempting directory change in restricted mode."
cd ..
echo "Still in `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Attempting to change shell in restricted mode."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo
```

```
echo "Attempting to redirect output in restricted mode."
ls -l /usr/bin > bin.files
# Try to list attempted file creation effort.
ls -l bin.files

echo

exit 0
```

3.18. Process Substitution

`Process substitution` is the counterpart to *command substitution*. Command substitution sets a variable to the result of a command, as in *`dir_contents=`ls -al``* or *`xref=$( grep word datafile)`*. Process substitution feeds the output of a process to another process (in other words, it sends the results of a command to another command).

- *`(command)>`*

  `<(command)`

  These initiate process substitution. This uses a *named pipe* (temp file) to send the results of the process within parentheses to another process.

  Note: There are *no* spaces between the parentheses and the "<" or ">". Space there would simply cause redirection from a subshell, rather than process substitution.

```
      cat <(ls -l)
      # Same as     ls -l | cat

      sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
      # Lists all the files in the 3 main 'bin' directories, and sorts by filename.
      # Note that three (count 'em) distinct commands are fed to 'sort'.
```

3.19. Functions

Like "real" programming languages, **bash** has functions, though in a somewhat limited implementation. A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task. Whenever there is repetitive code, when a task repeats with only slight variations, then writing a function should be investigated.

function *`function-name`* {
`command`...
}

or

`function-name` () {
`command`...

}

This second form will cheer the hearts of C programmers.

The opening bracket in the function may optionally be placed on the second line, to more nearly resemble C function syntax.

function−name ()
{
command...
}

Functions are called, *triggered*, simply by invoking their names.

Note that the function definition must precede the first call to it. There is no method of "declaring" the function, as, for example, in C.

Example 3−80. Simple function

```
#!/bin/bash

funky ()
{
  echo This is a funky function.
  echo Now exiting funky function.
}

# Note: function must precede call.

# Now, call the function.

funky

exit 0
```

More complex functions may have arguments passed to them and return exit values to the script for further processing.

```
function−name $arg1 $arg2
```

The function refers to the passed arguments by position (as if they were positional parameters), that is, `$1`, `$2`, and so forth.

Example 3−81. Function Taking Parameters

```
#!/bin/bash

func2 () {
   if [ −z $1 ]
   # Checks if any params.
   then
     echo "No parameters passed to function."
     return 0
```

```
   else
     echo "Param #1 is $1."
   fi

   if [ $2 ]
   then
     echo "Parameter #2 is $2."
   fi
}

func2
# Called with no params
echo

func2 first
# Called with one param
echo

func2 first second
# Called with two params
echo

exit 0
```

> **Note:** In contrast to certain other programming languages, shell scripts permit passing only value parameters to functions. Variable names (which are actually pointers), if passed as parameters to functions, will be treated as string literals and cannot be dereferenced. *Functions interpret their arguments literally.*

exit status

Functions return a value, called an *exit status*. The exit status may be explicitly specified by a **return** statement, otherwise it is the exit status of the last command in the function (0 if successful, and a non−zero error code if not). This exit status may be used in the script by referencing it as *$?*.

return

Terminates a function. The **return** statement optionally takes an integer argument, which is returned to the calling script as the "exit status" of the function, and this exit status is assigned to the variable *$?*.

Example 3−82. Converting numbers to Roman numerals

```
#!/bin/bash

# Arabic number to Roman numeral conversion
# Range 0 - 200
# It's crude, but it works.

# Extending the range and otherwise improving the script
# is left as an exercise for the reader.

# Usage: roman number-to-convert

ARG_ERR=1
OUT_OF_RANGE=200
```

```
if [ -z $1 ]
then
  echo "Usage: `basename $0` number-to-convert"
  exit $ARG_ERR
fi

num=$1
if [ $num -gt $OUT_OF_RANGE ]
then
  echo "Out of range!"
  exit $OUT_OF_RANGE
fi

to_roman ()
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ $remainder -ge 0 ]
do
  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number
}

# Note: must declare function
#       before first call to it.

to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
num=$?
to_roman $num 1 I

echo

exit 0
```

`local variables`

A variable declared as *local* is one that is visible only within the block of code in which it appears. In a shell script, this means the variable has meaning only within its own function.

Example 3−83. Local variable visibility

```
#!/bin/bash

func ()
{
  local a=23
  echo
  echo "a in function is $a"
  echo
}

func

# Now, see if local 'a'
# exists outside function.

echo "a outside function is $a"
echo
# Nope, 'a' not visible globally.

exit 0
```

Local variables permit recursion (a recursive function is one that calls itself), but this practice usually involves much computational overhead and is definitely *not* recommended in a shell script.

Example 3−84. Recursion, using a local variable

```
#!/bin/bash

#               factorial
#               ---------


# Does bash permit recursion?
# Well, yes, but...
# You gotta have rocks in your head to try it.


MAX_ARG=5
WRONG_ARGS=1
RANGE_ERR=2


if [ -z $1 ]
then
  echo "Usage: `basename $0` number"
  exit $WRONG_ARGS
fi

if [ $1 -gt $MAX_ARG ]
then
  echo "Out of range (5 is maximum)."
  # Let's get real now...
  # If you want greater range than this, rewrite it in a real programming language.
  exit $RANGE_ERR
fi

fact ()
```

```
{
  local number=$1
  # Variable "number" must be declared as local otherwise this doesn't work.
  if [ $number -eq 0 ]
  then
    factorial=1
  else
    let "decrnum = number - 1"
    fact $decrnum  # Recursive function call.
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Factorial of $1 is $?."

exit 0
```

3.20. Aliases

A bash *alias* is essentially nothing more than a keyboard shortcut, an abbreviation, a means of avoiding typing a long command sequence. If, for example, we include **alias lm="ls −l | more"** in the *~/.bashrc* file (see Section 3.23), then each `lm` typed at the command line will automatically be replaced by a **ls −l | more**. This can save a great deal of typing at the command line and avoid having to remember complex combinations of commands and options. Setting **alias rm="rm −i"** (interactive mode delete) may save a good deal of grief, since it can prevent inadvertently losing important files.

In a script, aliases have very limited usefulness. It would be quite nice if aliases could assume some of the functionality of the C preprocessor, such as macro expansion, but unfortunately Bash does not expand arguments within the alias body. Moreover, a script fails to expand an alias itself within "compound constructs", such as *if/then* statements, loops, and functions. Almost invariably, whatever we would like an alias to do could be accomplished much more effectively with a function.

Example 3−85. Aliases within a script

```
#!/bin/bash2

shopt -s expand_aliases
# Must set this option, else script will not expand aliases.


# First, some fun.
alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy starring Bob Hope."'
Jesse_James


echo; echo; echo;

alias ll="ls -l"
# May use either single (') or double (") quotes to define an alias.

echo "Trying aliased \"ll\":"
ll /usr/X11R6/bin/mk*   # Alias works.
```

```
echo

directory=/usr/X11R6/bin/
prefix=mk*  # See if wild-card causes problems.
echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "Trying aliased \"lll\":"
lll  # Long listing of all files in /usr/X11R6/bin stating with mk.
# Alias handles concatenated variables, including wild-card o.k.




TRUE=1

echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "Trying aliased \"rr\" within if/then statement:"
  rr /usr/X11R6/bin/mk*   # Error message results!
  # Aliases not expanded within compound statements.
  echo "However, previously expanded alias still recognized:"
  ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
  alias rrr="ls -l"
  echo "Trying aliased \"rrr\" within \"while\" loop:"
  rrr /usr/X11R6/bin/mk*   # Alias will not expand here either.
  let count+=1
done


exit 0
```

Note: The **unalias** command removes a previously set *alias*.

Example 3−86. unalias: Setting and unsetting an alias

```
#!/bin/bash

shopt -s expand_aliases  # Enables alias expansion.

alias llm='ls -al | more'
llm

echo

unalias llm    # Unset alias.
```

```
llm
# Error message results, since 'llm' no longer recognized.

exit 0
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: command not found
```

3.21. List Constructs

The "and list" and "or list" constructs provide a means of processing a number of commands consecutively. These can effectively replace complex nested **if/then** or even **case** statements. Note that the exit status of an "and list" or an "or list" is the exit status of the last command executed.

and list

```
command-1 && command-2 && command-3 && ... command-n
```

Each command executes in turn provided that the previous command has given a return value of true. At the first false return, the command chain terminates (the first command returning false is the last one to execute).

Example 3−87. Using an "and list" to test for command−line arguments

```
#!/bin/bash

# "and list"

if [ ! -z $1 ] && echo "Argument #1 = $1" && [ ! -z $2 ] && echo "Argument #2 = $2"
then
  echo "At least 2 arguments to script."
  # All the chained commands return true.
else
  echo "Less than 2 arguments to script."
  # At least one of the chained commands returns false.
fi
# Note that "if [ ! -z $1 ]" works, but its supposed equivalent,
# "if [ -n $1 ]" does not. This is a bug, not a feature.


# This accomplishes the same thing, coded using "pure" if/then statements.
if [ ! -z $1 ]
then
  echo "Argument #1 = $1"
fi
if [ ! -z $2 ]
then
  echo "Argument #2 = $2"
  echo "At least 2 arguments to script."
else
  echo "Less than 2 arguments to script."
fi
```

```
# It's longer and less elegant than using an "and list".


exit 0
```

or list

```
command−1 || command−2 || command−3 || ... command−n
```

Each command executes in turn for as long as the previous command returns false. At the first true return, the command chain terminates (the first command returning true is the last one to execute). This is obviously the inverse of the "and list".

Example 3−88. Using "or lists" in combination with an "and list"

```
#!/bin/bash

# "Delete", not-so-cunning file deletion utility.
# Usage: delete filename

if [ -z $1 ]
then
  file=nothing
else
  file=$1
fi
# Fetch file name (or "nothing") for deletion message.


[ ! -f $1 ] && echo "$1 not found. Can't delete a nonexistent file."
# AND LIST, to give error message if file not present.

[ ! -f $1 ] || ( rm -f $1; echo "$file deleted." )
# OR LIST, to delete file if present.
# ( command1 ; command2 ) is, in effect, an AND LIST variant.

# Note logic inversion above.
# AND LIST executes on true, OR LIST on false.

[ ! -z $1 ] ||  echo "Usage: `basename $0` filename"
# OR LIST, to give error message if no command line arg (file name).

exit 0
```

Clever combinations of "and" and "or" lists are possible, but the logic may easily become convoluted and require extensive debugging.

3.22. Arrays

Newer versions of **bash** support one−dimensional arrays. Arrays may be declared with the **`variable[xx]`** notation or explicitly by a **`declare -a variable`** statement. To dereference (find the contents of) an array variable, use *curly bracket* notation, that is, **`${variable[xx]}`**.

Example 3−89. Simple array usage

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

# Note that array members need not be consecutive
# or contiguous.

# Some members of the array can be left uninitialized.
# Gaps in the array are o.k.


echo -n "area[11] = "
echo ${area[11]}
echo -n "area[13] = "
echo ${area[13]}
# Note that {curly brackets} needed
echo "Contents of area[51] are ${area[51]}."

# Contents of uninitialized array variable print blank.
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43] unassigned)"

echo

# Sum of two array variables assigned to third
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# This doesn't work because
# adding an integer to a string is not permitted.

echo
echo
echo

# -----------------------------------------------------------------
# Another array, "area2".
# Another way of assigning array variables...
# array_name=( XXX YYY ZZZ ... )

area2=( zero one two three four)

echo -n "area2[0] = "
echo ${area2[0]}
# Aha, zero-based indexing (first element of array is [0], not [1]).

echo -n "area2[1] = "
echo ${area2[1]}  # [1] is second element of array.
# -----------------------------------------------------------------


echo
```

```
echo
echo

# -----------------------------------------------
# Yet another array, "area3".
# Yet another way of assigning array variables...
# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=seventeen [24]=twenty-four)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[24] = "
echo ${area3[24]}
# -----------------------------------------------


exit 0
```

Arrays variables have a syntax all their own, and even standard bash operators have special options adapted for array use.

Example 3−90. Some special properties of arrays

```
#!/bin/bash

declare -a colors
# Permits declaring an array without specifying size.

echo "Enter your favorite colors (separated from each other by a space)."

read -a colors
# Special option to 'read' command,
# allowing it to assign elements in an array.

echo

  element_count=${#colors[@]} # Special syntax to extract number of elements in array.
# element_count=${#colors[*]} works also.
index=0

# List all the elements in the array.
while [ $index -lt $element_count ]
do
  echo ${colors[$index]}
  let "index = $index + 1"
done
# Each array element listed on a separate line.
# If this is not desired, use  echo -n "${colors[$index]} "

echo

# Again, list all the elements in the array, but using a more elegant method.
  echo ${colors[@]}
# echo ${colors[*]} works also.


echo
```

```
exit 0
```

As seen in the previous example, either *${array_name[@]}* or *${array_name[*]}* refers to *all* the elements of the array. Similarly, to get a count of the number of elements in an array, use either *${#array_name[@]}* or *${#array_name[*]}*.

−−

Arrays permit deploying old familiar algorithms as shell scripts. Whether this is necessarily a good idea is left to the reader to decide.

Example 3−91. An old friend: *The Bubble Sort*

```
#!/bin/bash

# Bubble sort, of sorts.

# Recall the algorithm for a bubble sort. In this particular version...

# With each successive pass through the array to be sorted,
# compare two adjacent elements, and swap them if out of order.
# At the end of the first pass, the "heaviest" element has sunk to bottom.
# At the end of the second pass, the next "heaviest" one has sunk next to bottom.
# And so forth.
# This means that each successive pass needs to traverse less of the array.
# You will therefore notice a speeding up in the printing of the later passes.


exchange()
{
  # Swaps two members of the array.
  local temp=${Countries[$1]} # Temporary storage for element getting swapped out.
  Countries[$1]=${Countries[$2]}
  Countries[$2]=$temp

  return
}

declare -a Countries  # Declare array, optional here since it's initialized below.

Countries=(Netherlands Ukraine Zair Turkey Russia Yemen Syria Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England Israel Peru Canada Oman Denmark Wales France Kashmir Qatar Liechtenstein Hungary)
# Couldn't think of one starting with X (darn).

clear  # Clear the screen to start with.

echo "0: ${Countries[*]}"  # List entire array at pass 0.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # Pass number.

while [ $comparisons -gt 0 ]   # Beginning of outer loop
do

  index=0  # Reset index to start of array after each pass.
```

```
  while [ $index -lt $comparisons ] # Beginning of inner loop
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    # If out of order...
    # Recalling that \> is ASCII comparison operator.
    then
      exchange $index `expr $index + 1`  # Swap.
    fi
    let "index += 1"
  done # End of inner loop


let "comparisons -= 1"
# Since "heaviest" element bubbles to bottom, we need do one less comparison each pass.

echo
echo "$count: ${Countries[@]}"
# Print resultant array at end of each pass.
echo
let "count += 1"   # Increment pass count.

done  # End of outer loop

# All done.

exit 0
```

−−

Arrays enable implementing a shell script version of the *Sieve of Erastosthenes*. Of course, a resource−intensive application of this nature should really be written in a compiled language, such as C. It runs excruciatingly slowly as a script.

Example 3−92. Complex array application: *Sieve of Erastosthenes*

```
#!/bin/bash

# sieve.sh
# Sieve of Erastosthenes
# Ancient algorithm for finding prime numbers.

# This runs a couple of orders of magnitude
# slower than equivalent C program.

LOWER_LIMIT=1
# Starting with 1.
UPPER_LIMIT=1000
# Up to 1000.
# (You may set this higher...
#  if you have time on your hands.)

PRIME=1
NON_PRIME=0

let SPLIT=UPPER_LIMIT/2
# Optimization:
# Need to test numbers only
# halfway to upper limit.
```

```
declare -a Primes
# Primes[] is an array.


initialize ()
{
# Initialize the array.

i=$LOWER_LIMIT
until [ $i -gt $UPPER_LIMIT ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
# Assume all array members guilty (prime)
# until proven innocent.
}

print_primes ()
{
# Print out the members of the Primes[] array
# tagged as prime.

i=$LOWER_LIMIT

until [ $i -gt $UPPER_LIMIT ]
do

  if [ ${Primes[i]} -eq $PRIME ]
  then
    printf "%8d" $i
    # 8 spaces per number
    # gives nice, even columns.
  fi

  let "i += 1"

done

}

sift ()
{
# Sift out the non-primes.

let i=$LOWER_LIMIT+1
# We know 1 is prime, so
# let's start with 2.

until [ $i -gt $UPPER_LIMIT ]
do

if [ ${Primes[i]} -eq $PRIME ]
# Don't bother sieving numbers
# already sieved (tagged as non-prime).
then

  t=$i

  while [ $t -le $UPPER_LIMIT ]
  do
```

```
    let "t += $i "
    Primes[t]=$NON_PRIME
    # Tag as non-prime
    # all multiples.
  done

fi

  let "i += 1"
done


}


# Invoke the functions sequentially.
initialize
sift
print_primes
echo
# This is what they call structured programming.

exit 0
```

3.23. Files

- `/etc/profile`

  systemwide defaults, mostly setting the environment (all shells, not just Bash)
- `/etc/bashrc`

  systemwide functions and and aliases for Bash
- `$HOME/.bash_profile`

  user−specific Bash environmental default settings, found in each user's home directory (the local counterpart to `/etc/profile`)
- `$HOME/.bashrc`

  user−specific Bash init file, found in each user's home directory (the local counterpart to `/etc/bashrc`). Only interactive shells and user scripts read this file. See Appendix C for a sample `.bashrc` file.

These are the *startup files* for Bash. They contain the aliases (see Section 3.20) and environmental variables made available to Bash running as a user shell and to all Bash scripts invoked after system initialization.

3.24. Here Documents

A *here document* uses a special form of I/O redirection (see Section 3.13) to feed a command script to an interactive program, such as **ftp**, **telnet**, or **ex**. Typically, the script consists of a command list to the program, delineated by a limit string. The special symbol << precedes the limit string. This has the effect of redirecting the output of a file into the program, similar to

```
interactive-program
        < command-file
```

where `command-file` contains

```
command #1
command #2
...
```

The "here document" alternative looks like this:

```
#!/bin/bash
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

Choose a limit string sufficiently unusual that it will not occur anywhere in the command list and confuse matters.

Note that "here documents" may sometimes be used to good effect with non−interactive utilities and commands.

Example 3−93. dummyfile: Creates a 2−line dummy file

```
#!/bin/bash

# Non-interactive use of 'vi' to edit a file.
# Emulates 'sed'.

if [ -z $1 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi

TARGETFILE=$1

vi $TARGETFILE <<x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23

# Note that ^[ above is a literal escape
# typed by Control-V Escape

exit 0
```

The above script could just as effectively have been implemented with **ex**, rather than **vi**. Here documents containing a list of **ex** commands are common enough to form their own category, known as *ex scripts*.

Example 3−94. broadcast: Sends message to everyone logged in

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
Dees ees a message frrom Central Headquarters:
Do not keel moose!
# Other message text goes here.
# Note: Comment lines printed by 'wall'.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
# wall <message-file

exit 0
```

Example 3−95. Multi−line message using cat

```
#!/bin/bash

# 'echo' is fine for printing single line messages,
#  but somewhat problematic for for message blocks.
#  A 'cat' here document overcomes this limitation.

cat <<End-of-message
-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------
End-of-message

exit 0
```

Example 3−96. upload: Uploads a file pair to "Sunsite" incoming directory

```
#!/bin/bash

# upload
# upload file pair (filename.lsm, filename.tar.gz)
# to incoming directory at Sunsite


if [ -z $1 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi


Filename=`basename $1`
```

```
# Strips pathname out of file name

Server="metalab.unc.edu"
Directory="/incoming/Linux"
# These need not be hard-coded into script,
# may instead be changed to command line argument.

Password="your.e-mail.address"
# Change above to suit.

ftp -n $Server <<End-Of-Session
# -n option disables auto-logon

user anonymous $Password
binary
bell
# Ring 'bell' after each file transfer
cd $Directory
put $Filename.lsm
put $Filename.tar.gz
bye
End-Of-Session

exit 0
```

Note: Some utilities will not work in a "here document". The pagers, **more** and **less** are among these.

For those tasks too complex for a "here document", consider using the **expect** scripting language, which is specifically tailored for feeding input into non−interactive programs.

3.25. Of Zeros and Nulls

Uses of `/dev/null`

Think of `/dev/null` as a "black hole". It is the nearest equivalent to a write−only file. Everything written to it disappears forever. Attempts to read or output from it result in nothing. Nevertheless, `/dev/null` can be quite useful from both the command line and in scripts.

Suppressing stdout or stderr (from Example 3−98):

```
rm $badname 2>/dev/null
#           So error messages [stderr] deep-sixed.
```

Deleting contents of a file, but preserving the file itself, with all attendant permissions (from Example 2−1 and Example 2−2):

```
cat /dev/null > /var/log/messages
cat /dev/null > /var/log/wtmp
```

Automatically emptying the contents of a log file (especially good for dealing with those nasty "cookies" sent by Web commercial sites):

```
rm -f ~/.netscape/cookies
```

```
ln -s /dev/null ~/.netscape/cookies
# All cookies now get sent to a black hole, rather than saved to disk.
```

Uses of `/dev/zero`

Like `/dev/null`, `/dev/zero` is a pseudo file, but it actually contains nulls (numerical zeros, not the ASCII kind). Output written to it disappears, and it is fairly difficult to actually read the nulls in `/dev/zero`, though it can be done with **od** or a hex editor. The chief use for `/dev/zero` is in creating an initialized dummy file of specified length intended as a temporary swap file.

Example 3−97. Setting up a swapfile using `/dev/zero`

```
#!/bin/bash

# Creating a swapfile.
# This script must be run as root.

FILE=/swap
BLOCKSIZE=1024
PARAM_ERROR=33
SUCCESS=0


if [ -z $1 ]
then
  echo "Usage: `basename $0` swapfile-size"
  # Must be at least 40 blocks.
  exit $PARAM_ERROR
fi

dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$1

echo "Creating swapfile of size $1 blocks (KB)."

mkswap $FILE $1
swapon $FILE

echo "Swapfile activated."

exit $SUCCESS
```

3.26. Debugging

The Bash shell contains no debugger, nor even any debugging−specific commands or constructs. Syntax errors or outright typos in the script generate cryptic error messages that are often of no help in debugging a non−functional script.

Example 3−98. test23, a buggy script

```
#!/bin/bash

a=37
```

```
if [$a -gt 27 ]
then
  echo $a
fi

exit 0
```

Output from script:

```
./test23: [37: command not found
```

What's wrong with the above script (hint: after the **if**)?

What if the script executes, but does not work as expected? This is the all too familiar logic error.

Example 3−99. test24, another buggy script

```
#!/bin/bash

# This is supposed to delete all filenames
# containing embedded spaces in current directory,
# but doesn't.  Why not?


badname=`ls | grep ' '`

# echo "$badname"

rm "$badname"

exit 0
```

To find out what's wrong with Example 3−99, uncomment the **`echo "$badname"`** line. Echo statements are useful for seeing whether what you expect is actually what you get.

Summarizing the symptoms of a buggy script,

1. It bombs with an error message syntax error, or
2. It runs, but does not work as expected (logic error)
3. It runs, works as expected, but has nasty side effects (logic bomb).

Tools for debugging non−working scripts include

1. echo statements at critical points in the script to trace the variables, and otherwise give a snapshot of what is going on.
2. using the **tee** filter to check processes or data flows at critical points.
3. setting option flags `-n -v -x`

   `sh -n scriptname` checks for syntax errors without actually running the script. This is the equivalent of inserting **`set -n`** or **`set -o noexec`** into the script. Note that certain types of syntax errors can slip past this check.

   `sh -v scriptname` echoes each command before executing it. This is the equivalent of inserting

`set -v` or **`set -o verbose`** in the script.

`sh -x scriptname` echoes the result each command, but in an abbreviated manner. This is the equivalent of inserting **`set -x`** or **`set -o xtrace`** in the script.

Inserting **`set -u`** or **`set -o nounset`** in the script runs it, but gives an unbound variable error message at each attempt to use an undeclared variable.

4. trapping at exit

   The **exit** command in a script actually sends a signal 0, terminating the process, that is, the script itself. It is often useful to trap the **exit**, forcing a "printout" of variables, for example. The **trap** must be the first command in the script.

trap

Specifies an action on receipt of a signal; also useful for debugging.

Note: A *signal* is simply a message sent to a process, either by the kernel or another process, telling it to take some specified action (usually to terminate). For example, hitting a **Control−C**, sends a user interrupt, an INT signal, to a running program.

```
trap 2 #ignore interrupts (no action specified)
trap 'echo "Control-C disabled."' 2
```

Example 3−100. Trapping at exit

```
#!/bin/bash

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
# EXIT is the name of the signal generated upon exit from a script.

a=39

b=36

exit 0
# Note that commenting out the 'exit' command makes no difference,
# since the script exits anyhow after running out of commands.
```

Example 3−101. Cleaning up after Control−C

```
#!/bin/bash

# logon.sh
# A quick 'n dirty script to check whether you are on-line yet.


TRUE=1
LOGFILE=/var/log/messages
# Note that $LOGFILE must be readable (chmod 644 /var/log/messages).
TEMPFILE=temp.$$
# Create a "unique" temp file name, using process id of the script.
```

```
KEYWORD=address
# At logon, the line "remote IP address xxx.xxx.xxx.xxx" appended to /var/log/messages.
ONLINE=22
USER_INTERRUPT=13

trap 'rm −f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
# Cleans up the temp file if script interrupted by control−c.

echo

while [ $TRUE ]  #Endless loop.
do
  tail −1 $LOGFILE> $TEMPFILE
  # Saves last line of system log file as temp file.
  search=`grep $KEYWORD $TEMPFILE`
  # Checks for presence of the "IP address" phrase,
  # indicating a successful logon.

  if [ ! −z "$search" ] # Quotes necessary because of possible spaces.
  then
     echo "On−line"
     rm −f $TEMPFILE  # Clean up temp file.
     exit $ONLINE
  else
     echo −n "." # −n option to echo suppresses newline,
                 # so you get continuous rows of dots.
  fi

  sleep 1
done


# Note: if you change the KEYWORD variable to "Exit",
# this script can be used while on−line to check for an unexpected logoff.

# Exercise: Change the script, as per the above note,
#           and prettify it.

exit 0
```

Note: **`trap '' SIGNAL`** (two adjacent apostrophes) disables SIGNAL for the remainder of the script. **`trap SIGNAL`** restores the functioning of SIGNAL once more. This is useful to protect a critical portion of a script from an undesirable interrupt.

```
        trap '' 2  # Signal 2 is Control−C, now disabled.
        command
        command
        command
        trap 2     # Reenables Control−C
```

3.27. Options

Options are settings that change shell and/or script behavior.

The **set** command (see Section 3.9) enables options within a script. At the point in the script where you want the options to take effect, use **set −o option−name** or, in short form, **set −option−abbrev**. These two forms are equivalent.

```
#!/bin/bash

set −o verbose
# Echoes all commands before executing.
```

```
#!/bin/bash

set −v
# Exact same effect as above.
```

Note: To *disable* an option within a script, use **set +o option−name** or **set +option−abbrev**.

```
#!/bin/bash

set −o verbose
# Command echoing on.
command
...
command

set +o verbose
# Command echoing off.
command
# Not echoed.


set −v
# Command echoing on.
command
...
command

set +v
# Command echoing off.
command

exit 0
```

An alternate method of enabling options in a script is to specify them immediately following the `#!` script header.

```
#!/bin/bash −x
#
# Body of script follows.
```

It is also possible to enable script options from the command line. Some options that will not work with **set** are available this way. Among these are *−i*, force script to run interactive.

`bash −v script−name`

`bash −o verbose script−name`

The following is a listing of some useful options. They may be specified in either abbreviated form or by complete name.

Table 3−1. bash options

Abbreviation	Name	Effect
`−C`	noclobber	Prevent overwriting of files by redirection (may be overridden by >|)
`−D`	(none)	List double−quoted strings prefixed by $, but do not execute commands in script
`−a`	allexport	Export all defined variables
`−b`	notify	Notify when jobs running in background terminate (not of much use in a script)
`−c xxx`	(none)	Read commands from *xxx*
`−f`	noglob	Filename expansion (globbing) disabled
`−i`	interactive	Script runs in *interactive* mode
`−p`	privileged	Script runs as "suid" (caution!)
`−r`	restricted	Script runs in *restricted* mode (see Section 3.17).
`−u`	nounset	Attempt to use undefined variable outputs error message
`−v`	verbose	Print each command to stdout before executing it
`−x`	xtrace	Similar to `−v`, but expands commands
`−e`	errexit	Abort script at first error (when a command exits with non−zero status)
`−n`	noexec	Read commands in script, but do not execute them
`−s`	stdin	Read commands from stdin
`−t`	(none)	Exit after first command
`−`	(none)	End of options flag. All other arguments are positional parameters.
`−−`	(none)	Unset positional parameters. If arguments given (`−−arg1arg2`), positional parameters set to arguments.

3.28. Gotchas

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

Puccini

Assigning reserved words or characters to variable names.

```
var1=case
# Causes problems.
var2=23skidoo
# Also problems. Variable names starting with a digit are reserved by the shell.
```

```
# Try var2=_23skidoo. Starting variables with an underscore is o.k.
var3=xyz((!*
# Causes even worse problems.
```

Using a hyphen or other reserved characters in a variable name.

```
var-1=23
# Use 'var_1' instead.
```

Using white space inappropriately (in contrast to other programming languages **bash** can be finicky about white space).

```
var1 = 23
# 'var1=23' is correct.
let c = $a - $b
# 'let c=$a-$b' or 'let "c = $a - $b"' are correct.
if [ $a -le 5]
# 'if [ $a -le 5 ]' is correct.
```

Using uninitialized variables (that is, using variables before a value is assigned to them). An uninitialized variable has a value of "null", *not* zero.

Mixing up = and *−eq* in a test. Remember, = is for comparing literal variables and *−eq* is for numbers.

```
        if [ $a = 273 ] # Wrong!
        if [ $a -eq 273 ] # Correct.
```

Sometimes variables within "test" brackets ([ ]) need to be quoted (double quotes). Failure to do so may cause unexpected behavior. See Example 3−13, Example 3−73, and Example 3−17.

Commands issued from a script may fail to execute because the script owner lacks execute permission for them. If a user cannot invoke a command from the command line, then putting it into a script will likewise fail. Try changing the attributes of the command in question, perhaps setting the suid bit (as root, of course).

Using bash version 2 functionality (see below) in a script headed with **`#!/bin/bash`** may cause a bailout with error messages. Your system may still have an older version of bash as the default installation (**echo $BASH_VERSION**). Try changing the header of the script to **`#!/bin/bash2`**.

A script may not **export** variables back to its parent process, the shell, or to the environment. Just as we learned in biology, a child process can inherit from a parent, but not vice versa.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER

bash$
```

Sure enough, back at the command prompt, $WHATEVER remains unset.

Making scripts "suid" is generally a bad idea, as it may compromise system security. Administrative scripts should be run by root, not regular users.

Using shell scripts for CGI programming may be problematic. Shell script variables are not "typesafe", and this can cause undesirable behavior as far as CGI is concerned. Moreover, it is difficult to "hacker−proof" shell scripts.

3.29. Miscellany

Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help.

Tom Duff

3.29.1. Interactive and non−interactive scripts

Let us define an *interactive* script as one that requires input from the user, usually with **read** statements (see Example 3−42). "Real life" is actually a bit messier than that, and the formal specifications of an interactive shell (according to Ramey & Fox) are complex and confusing. For now, assume an interactive script is one that is bound to a tty, a script that a user has invoked from the console or an xterm.

Init and startup scripts are necessarily non−interactive, since they must run without human intervention. Many administrative and system maintenance scripts are likewise non−interactive. Unvarying repetitive tasks cry out for automation by non−interactive scripts.

Non−interactive scripts can run in the background, but interactive ones hang, waiting for input that never comes. Handle that difficulty by having an **expect** script or embedded *`here document`* (see Section 3.24) feed input to an interactive script running as a background job. In the simplest case, redirect a file to supply input to a **read** statement (*read variable <file*). These particular workarounds make possible general purpose scripts that run in either interactive or non−interactive modes.

If a script needs to test whether it is running in interactive mode, it is simply a matter of finding whether the *prompt* variable, *`$PS1`* is set. (If the user is being prompted for input, then the script needs to display a prompt.)

```
if [ -z $PS1 ] # no prompt?
then
  # non-interactive
  ...
else
  # interactive
  ...
fi
```

Alternatively, the script can test for the presence of *i* in the $− flag.

```
case $- in
*i*)    # interactive script
;;
*)      # non-interactive script
;;
# (Thanks to "UNIX F.A.Q.", 1993)
```

Note: Scripts may be forced to run in interactive mode with the i option or with a *#!/bin/bash −i* header. Be aware that this may cause erratic script behavior or show error messages where no error is present.

3.29.2. Optimizations

Most shell scripts are quick 'n dirty solutions to non−complex problems. As such, optimizing them for speed is not much of an issue. Consider the case, though, where a script carries out an important task, does it well, but runs too slowly. Rewriting it in a compiled language may not be a palatable option. The simplest fix would be to rewrite the parts of the script that slow it down. Is it possible to apply principles of code optimization even to a lowly shell script?

Check the loops in the script. Time consumed by repetitive operations adds up quickly. Use the **time** and **times** tools to profile computation−intensive commands. Consider rewriting time−critical code sections in C, or even in assembler.

Try to minimize file i/o. Bash is not particularly efficient at handling files, so consider using more appropriate tools for this within the script, such as awk or Perl.

Try to write your scripts in a structured, coherent form, so they can be reorganized and tightened up as necessary. Some of the optimization techniques applicable to high−level languages may work for scripts, but others, such as loop unrolling, are mostly irrelevant. Above all, use common sense.

3.29.3. Assorted Tips

- To keep a record of which user scripts have run during a particular sesssion or over a number of sessions, add the following lines to each script you want to keep track of. This will keep a continuing file record of the script names and invocation times.

```
# Append (>>) following to end of save file.
date>> $SAVE_FILE   #Date and time.
echo $0>> $SAVE_FILE   #Script name.
echo>> $SAVE_FILE   #Blank line as separator.
# Of course, SAVE_FILE defined and exported as environmental variable in ~/.bashrc
# (something like ~/.scripts-run)
```

- A shell script may act as an embedded command inside another shell script, a *Tcl* or *wish* script, or even a Makefile. It can be invoked as as an external shell command in a C program using the `system()` call, i.e., `system("script_name");`.
- Put together a file of your favorite and most useful definitions and functions, then "include" this file in scripts as necessary with either the "dot" (**.**) or **source** command (see Section 3.2).
- It would be nice to be able to invoke X−Windows widgets from a shell script. There do, in fact, exist a couple of packages that purport to do so, namely *Xscript* and *Xmenu*, but these seem to be pretty much defunct. If you dream of a script that can create widgets, try *wish* (a *Tcl* derivative), *PerlTk* (Perl with Tk extensions), or *tksh* (ksh with Tk extensions).

3.30. Bash, version 2

The current version of **bash**, the one you have running on your machine, is actually version 2. This update of the classic **bash** scripting language added array variables, string and parameter expansion, and a better method of indirect variable references, among other features.

Example 3−102. String expansion

```
#!/bin/bash

# String expansion.
# Introduced in version 2 of bash.

# Strings of the form $'xxx'
# have the standard escaped characters interpreted.

echo $'Ringing bell 3 times \a \a \a'
echo $'Three form feeds \f \f \f'
echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'

exit 0
```

Example 3−103. Indirect variable references − the new way

```
#!/bin/bash

# Indirect variable referencing.
# This has a few of the attributes of references in C++.


a=letter_of_alphabet
letter_of_alphabet=z

# Direct reference.
echo "a = $a"

# Indirect reference.
echo "Now a = ${!a}"
# The ${!variable} notation is greatly superior to the old "eval var1=\$$var2"

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"
table_cell_3=387
echo "Value of t changed to ${!t}"
# Useful for referencing members
# of an array or table,
# or for simulating a multi-dimensional array.
# An indexing option would have been nice (sigh).


exit 0
```

Example 3−104. Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards

```
#!/bin/bash2
# Must specify version 2 of bash, else might not work.

# Cards:
# deals four random hands from a deck of cards.

UNPICKED=0
PICKED=1

DUPE_CARD=99

LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUITE=13
CARDS=52

declare -a Deck
declare -a Suites
declare -a Cards
# It would have been easier and more intuitive
# with a single, 3-dimensional array. Maybe
# a future version of bash will support
# multidimensional arrays.


initialize_Deck ()
{
i=$LOWER_LIMIT
until [ $i -gt $UPPER_LIMIT ]
do
  Deck[i]=$UNPICKED
  let "i += 1"
done
# Set each card of "Deck" as unpicked.
echo
}

initialize_Suites ()
{
Suites[0]=C #Clubs
Suites[1]=D #Diamonds
Suites[2]=H #Hearts
Suites[3]=S #Spades
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
# Alternate method of initializing array.
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS"
if [ ${Deck[card_number]} -eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
```

```
  return $card_number
else
  return $DUPE_CARD
fi
}

parse_card ()
{
number=$1
let "suite_number = number / CARDS_IN_SUITE"
suite=${Suites[suite_number]}
echo -n "$suite-"
let "card_no = number % CARDS_IN_SUITE"
Card=${Cards[card_no]}
printf %-4s $Card
# Print cards in neat columns.
}

seed_random ()
{
# Seed random number generator.
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}

deal_cards ()
{
echo

cards_picked=0
while [ $cards_picked -le $UPPER_LIMIT ]
do
  pick_a_card
  t=$?

  if [ $t -ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # Change back to 1-based indexing (temporarily).
    let "u %= $CARDS_IN_SUITE"
    if [ $u -eq 0 ]
    then
     echo
     echo
    fi
    # Separate hands.

    let "cards_picked += 1"
  fi
done

echo

return 0
}


# Structured programming:
# entire program logic modularized in functions.
```

```
#================
seed_random
initialize_Deck
initialize_Suites
initialize_Cards
deal_cards

exit 0
#================



# Exercise 1:
# Add comments to thoroughly document this script.

# Exercise 2:
# Revise the script to print out each hand sorted in suites.
# You may add other bells and whistles if you like.

# Exercise 3:
# Simplify and streamline the logic of the script.
```

Chapter 4. Credits

Philippe Martin translated this document into DocBook/SGML. While not on the job at a small French company as a software developer, he enjoys working on GNU/Linux documentation and software, reading literature, playing music, and for his peace of mind making merry with friends. You may run across him somewhere in France or in the Basque Country, or email him at feloy@free.fr.

I would like to especially thank *Patrick Callahan*, *Mike Novak*, and *Pal Domokos* for catching bugs, pointing out ambiguities, and for suggesting clarifications and changes. Their lively discussion of shell scripting and general documentation issues inspired me to try to make this HOWTO more readable.

I'm grateful to Jim Van Zandt for pointing out errors and omissions in version 0.2 of this HOWTO. He also contributed an instructive example script.

Many thanks to Jordi Sanfeliu mikaku@arrakis.es for giving permission to use his fine tree script (Example A−8).

Emmanuel Rouat suggested corrections and additions on command substitution and aliases (see Section 3.12 and Section 3.20). He also contributed a very nice sample `.bashrc` file (Appendix C).

Florian Wisser enlightened me on some of the fine points of testing strings (see Example 3−13).

Others making helpful suggestions were Gabor Kiss and Leopold Toetsch.

My gratitude to Chet Ramey for writing **Bash**, an elegant and powerful scripting tool.

Thanks most of all to my wife, Anita, for her encouragement and emotional support.

Bibliography

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1−156592−225−5.

To unfold the full power of shell scripting, you need at least a passing familiarity with **sed** and **awk**. This is the standard tutorial. It includes an excellent introduction to "regular expressions". Read this book.

*

Aeleen Frisch, *Essential System Administration*, 2nd edition, O'Reilly and Associates, 1995, 1−56592−127−5.

This excellent sys admin manual has a decent introduction to shell scripting for sys administrators and does a nice job of explaining the startup and initialization scripts. The book is long overdue for a third edition (are you listening, Tim O'Reilly?).

*

Stephen Kochan and Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

The standard reference, though a bit dated by now.

*

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1−56592−347−2.

This is a valiant effort at a decent shell primer, but somewhat deficient in coverage on programming topics and lacking sufficient examples.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

A very handy pocket reference, despite lacking coverage of Bash−specific features.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1−56592−260−3.

Contains a couple of sections of very informative in−depth articles on shell programming, but falls short of being a tutorial. It reproduces much of the regular expressions tutorial from the Dougherty and Robbins book, above.

*

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1−58731−010−5.

Excellent Bash pocket reference (don't leave home without it). A bargain at $4.95, but also available for free download on−line in pdf format.

*

Ellen Siever and and the Staff of O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1−56592−585−8.

The all−around best Linux command reference, even has a Bash section.

*

The UNIX CD Bookshelf, 2nd edition, O'Reilly and Associates, 2000, 1−56592−815−6.

An array of six UNIX books on CD ROM, including *UNIX Power Tools*, *Sed and Awk*, and *Learning the Korn Shell*. A complete set of all the UNIX references and tutorials you would ever need at about $70. Buy this one, even if it means going into debt and not paying the rent.

*

The O'Reilly books on Perl. (Actually, any O'Reilly books.)

The man pages for **bash** and **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **xargs**. The texinfo documentation on **bash**, **dd**, **gawk**, and **sed**.

The excellent "Bash Reference Manual", by Chet Ramey and Brian Fox, distributed as part of the "bash−2−doc" package (available as an rpm). See especially the instructive example scripts in this package.

Ben Okopnik's well−written *introductory Bash scripting* articles in issues 53, 54, 55, 57, and 59 of the Linux Gazette , and his explanation of "The Deep, Dark Secrets of Bash" in issue 56.

Chet Ramey's *bash − The GNU Shell*, a two−part series published in issues 3 and 4 of the Linux Journal, July−August 1994.

Giles Orr's Bash−Prompt HOWTO.

Mike G's Bash−Programming−Intro HOWTO.

Mark Komarinski's Printing−Usage HOWTO.

Trent Fisher's groff tutorial.

Chet Ramey's Bash F.A.Q.

The sed F.A.Q.

Carlos Duarte's instructive "Do It With Sed" tutorial.

The GNU **gawk** reference manual (**gawk** is the extended GNU version of **awk** available on Linux and BSD systems).

There is some nice material on I/O redirection (Section 3.13) in chapter 10 of the textutils documentation at the University of Alberta site.

Appendix A. Contributed Scripts

These scripts, while not fitting into the text of this document, do illustrate some interesting shell programming techniques. They are useful, too. Have fun analyzing and running them.

Example A−1. manview: A script for viewing formatted man pages

```
#!/bin/bash

# Formats the source of a man page for viewing in a user directory.
# This is useful when writing man page source and you want to
# look at the intermediate results on the fly while working on it.

if [ -z $1 ]
then
  echo "Usage: `basename $0` [filename]"
    exit 1
fi

groff -Tascii -man $1 | less
# From the man page for groff.

exit 0
```

Example A−2. rn: A simple−minded file rename utility

This script is a modification of Example 3−58.

```
#! /bin/bash
#
# Very simpleminded filename "rename" utility.
# Based on "lowercase.sh".


if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` old-pattern new-pattern"
  # As in "rn gif jpg", which renames all gif files in working directory to jpg.
  exit 1
fi

number=0    # Keeps track of how many files actually renamed.


for filename in *$1*  #Traverse all matching files in directory.
do
   if [ -f $filename ]  # If finds match...
   then
     fname=`basename $filename`            # Strip off path.
     n=`echo $fname | sed -e "s/$1/$2/"`   # Substitute new for old in filename.
     mv $fname $n                          # Rename.
     let "number += 1"
   fi
done

if [ $number -eq 1 ]   # For correct grammar.
```

```
then
 echo "$number file renamed."
else
 echo "$number files renamed."
fi

exit 0


# Exercise for reader:
# What type of files will this not work on?
# How to fix this?
```

Example A−3. encryptedpw: A script for uploading to an ftp site, using a locally encrypted password

```
#!/bin/bash

# Example 3-71 modified to use encrypted password.

if [ -z $1 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi

Username=bozo
# Change to suit.

Filename=`basename $1`
# Strips pathname out of file name

Server="XXX"
Directory="YYY"
# Change above to actual server name & directory.


password=`cruft <pword`
# "pword" is the file containing encrypted password.
# Uses the author's own "cruft" file encryption package,
# based on onetime pad algorithm,
# and obtainable from:
# Primary-site:   ftp://metalab.unc.edu /pub/Linux/utils/file
#                 cruft-0.2.tar.gz [16k]


ftp -n $Server <<End-Of-Session
# -n option disables auto-logon

user $Username $Password
binary
bell
# Ring 'bell' after each file transfer
cd $Directory
put $Filename
bye
End-Of-Session

exit 0
```

+

The following two scripts are by Mark Moraes of the University of Toronto. See the enclosed file "Moraes−COPYRIGHT" for permissions and restrictions.

Example A−4. behead: A script for removing mail and news message headers

```
#! /bin/sh
# Strips off the header from a mail/News message i.e. till the first
# empty line
# Mark Moraes, University of Toronto

# --> These comments added by author of HOWTO.

if [ $# -eq 0 ]; then
# --> If no command line args present, then works on file redirected to stdin.
	sed -e '1,/^$/d' -e '/^[	]*$/d'
	# --> Delete empty lines and all lines until
	# --> first one beginning with white space.
else
# --> If command line args present, then work on files named.
	for i do
		sed -e '1,/^$/d' -e '/^[	]*$/d' $i
		# --> Ditto, as above.
	done
fi

# --> Exercise for the reader: Add error checking and other options.
# -->
# --> Note that the small sed script repeats, except for the arg passed.
# --> Does it make sense to embed it in a function? Why or why not?
```

Example A−5. ftpget: A script for downloading files via ftp

```
#! /bin/sh
# $Id: ftpget,v 1.2 91/05/07 21:15:43 moraes Exp $
# Script to perform batch anonymous ftp. Essentially converts a list of
# of command line arguments into input to ftp.
# Simple, and quick - written as a companion to ftplist
# -h specifies the remote host (default prep.ai.mit.edu)
# -d specifies the remote directory to cd to - you can provide a sequence
# of -d options - they will be cd'ed to in turn. If the paths are relative,
# make sure you get the sequence right. Be careful with relative paths -
# there are far too many symlinks nowadays.
# (default is the ftp login directory)
# -v turns on the verbose option of ftp, and shows all responses from the
# ftp server.
# -f remotefile[:localfile] gets the remote file into localfile
# -m pattern does an mget with the specified pattern. Remember to quote
# shell characters.
# -c does a local cd to the specified directory
# For example,
#	ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
#		-d ../pub/R3/fixes -c ~/fixes -m 'fix*'
# will get xplaces.shar from ~ftp/contrib on expo.lcs.mit.edu, and put it in
# xplaces.sh in the current working directory, and get all fixes from
# ~ftp/pub/R3/fixes and put them in the ~/fixes directory.
# Obviously, the sequence of the options is important, since the equivalent
# commands are executed by ftp in corresponding order
#
```

```
# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
# --> Angle brackets changed to parens, so Docbook won't get indigestion.
#


# --> These comments added by author of HOWTO.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# --> Above 2 lines from original script probably superfluous.

TMPFILE=/tmp/ftp.$$
# --> Creates temp file, using process id of script ($$)
# --> to construct filename.

SITE=`domainname`.toronto.edu
# --> 'domainname' similar to 'hostname'
# --> May rewrite this to parameterize this for general use.

usage="Usage: $0 [-h remotehost] [-d remotedirectory]... [-f remfile:localfile]... \
                [-c localdirectory] [-m filepattern] [-v]"
ftpflags="-i -n"
verbflag=
set -f          # So we can use globbing in -m
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
        echo $usage
        exit 1
fi
shift
trap 'rm -f ${TMPFILE} ; exit' 0 1 2 3 15
echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
# --> Added quotes (recommended in complex echoes).
echo binary >> ${TMPFILE}
for i in $*
# --> Parse command line args.
do
        case $i in
        -v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
        -h) remhost=$2; shift 2;;
        -d) echo cd $2 >> ${TMPFILE};
            if [ x${verbflag} != x ]; then
                echo pwd >> ${TMPFILE};
            fi;
            shift 2;;
        -c) echo lcd $2 >> ${TMPFILE}; shift 2;;
        -m) echo mget "$2" >> ${TMPFILE}; shift 2;;
        -f) f1=`expr "$2" : "\([^:]*\).*"`; f2=`expr "$2" : "[^:]*:\(.*\)"`;
            echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
        --) shift; break;;
        esac
done
if [ $# -ne 0 ]; then
        echo $usage
        exit 2
fi
if [ x${verbflag} != x ]; then
        ftpflags="${ftpflags} -v"
fi
if [ x${remhost} = x ]; then
        remhost=prep.ai.mit.edu
        # --> Rewrite to match your favorite ftp site.
```

```
fi
echo quit >> ${TMPFILE}
# --> All commands saved in tempfile.

ftp ${ftpflags} ${remhost} < ${TMPFILE}
# --> Now, tempfile batch processed by ftp.

rm -f ${TMPFILE}
# --> Finally, tempfile deleted (you may wish to copy it to a logfile).


# --> Exercises for reader:
# --> 1) Add error checking.
# --> 2) Add bells & whistles.
```

+

Antek Sawicki contributed the following script, which makes very clever use of the parameter substitution operators discussed in Section 3.3.1.

Example A−6. password: A script for generating random 8−character passwords

```
#!/bin/bash
# May need to be invoked with  #!/bin/bash2  on some machines.
#
# Random password generator for bash 2.x by Antek Sawicki <tenox@tenox.tc>,
# who generously gave permission to the HOWTO author to use it here.
#
# ==> Comments added by HOWTO author ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
LENGTH="8"
# ==> May change 'LENGTH' for longer password, of course.


while [ ${n:=1} -le $LENGTH ]
# ==> Recall that := is "default substitution" operator.
# ==> So, if 'n' has not been initialized, set it to 1.
do
        PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
        # ==> Very clever, but tricky.

        # ==> Starting from the innermost nesting...
        # ==> ${#MATRIX} returns length of array MATRIX.
        # ==> $RANDOM%${#MATRIX} returns random number between 1 and length of MATRIX.

        # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
        # ==> returns expansion of MATRIX at random position, by length 1.
        # ==> See {var:pos:len} parameter substitution in Section 3.3.1 and following examples.

        # ==> PASS=... simply pastes this result onto previous PASS (concatenation).

        # ==> To visualize this more clearly, uncomment the following line
        # ==>             echo "$PASS"
        # ==> to see PASS being built up, one character at a time, each iteration of the loop.

        let n+=1
        # ==> Increment 'n' for next pass.
```

```
done

echo "$PASS"
#== Or, redirect to file, as desired.
```

+

James R. Van Zandt contributed this script, which uses named pipes and, in his words, "really exercises quoting and escaping".

Example A−7. fifo: A script for making daily backups, using named pipes

```
#!/bin/bash
# ==> Script by James R. Van Zandt, and used here with his permission.

# ==> Comments added by author of HOWTO.


  HERE=`uname -n`
  # ==> hostname
  THERE=bilbo
  echo "starting remote backup to $THERE at `date +%r`"
  # ==> `date +%r` returns time in 12-hour format, i.e. "08:08:34 PM".

  # make sure /pipe really is a pipe and not a plain file
  rm -rf /pipe
  mkfifo /pipe
  # ==> Create a "named pipe", named "/pipe".

  # ==> 'su xyz' runs commands as user "xyz".
  # ==> 'ssh' invokes secure shell (remote login client).
  su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
  cd /
  tar -czf - bin boot dev etc home info lib man root sbin share usr var >/pipe
  # ==> Uses named pipe, /pipe, to communicate between processes:
  # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.

  # ==> The end result is this backs up the main directories, from / on down.

  # ==> What are the advantages of a "named pipe" in this situation,
  # ==> as opposed to an "anonymous pipe", with |?
  # ==> Will an anonymous pipe even work here?


  exit 0
```

+

Jordi Sanfeliu gave permission to use his elegant *tree* script.

Example A−8. tree: A script for displaying a directory tree

```
#!/bin/sh
#         @(#) tree      1.1  30/11/95       by Jordi Sanfeliu
#                                         email: mikaku@arrakis.es
#
```

```
#         Initial version:  1.0  30/11/95
#         Next version   :  1.1  24/02/97   Now, with symbolic links
#         Patch by       :  Ian Kjos, to support unsearchable dirs
#                           email: beth13@mail.utexas.edu
#
#         Tree is a tool for view the directory tree (obvious :-) )
#

# ==> 'Tree' script used here with the permission of its author, Jordi Sanfeliu.
# ==> Comments added by HOWTO author.


search () {
   for dir in `echo *`
   # ==> `echo *` lists all the files in current working directory, without line breaks.
   # ==> Same effect as     for dir in *
   do
      if [ -d $dir ] ; then   # ==> If it is a directory (-d)...
         zz=0   # ==> Temp variable, keeping track of directory level.
         while [ $zz != $deep ]   # Keep track of inner nested loop.
         do
            echo -n "|   "    # ==> Display vertical connector symbol,
                              # ==> with 2 spaces & no line feed in order to indent.
            zz=`expr $zz + 1` # ==> Increment zz.
         done
         if [ -L $dir ] ; then    # ==> If directory is a symbolic link...
            echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
            # ==> Display horiz. connector and list directory name, but...
            # ==> delete date/time part of long listing.
         else
            echo "+---$dir"    # ==> Display horizontal connector symbol...
                               # ==> and print directory name.
            if cd $dir ; then  # ==> If can move to subdirectory...
               deep=`expr $deep + 1`   # ==> Increment depth.
               search    # with recursivity ;-)
                         # ==> Function calls itself.
               numdirs=`expr $numdirs + 1`   # ==> Increment directory count.
            fi
         fi
      fi
   done
   cd ..   # ==> Up one directory level.
   if [ $deep ] ; then  # ==> If depth = 0 (returns TRUE)...
      swfi=1            # ==> set flag showing that search is done.
   fi
   deep=`expr $deep - 1`   # ==> Decrement depth.
}

# - Main -
if [ $# = 0 ] ; then
   cd `pwd`    # ==> No args to script, then use current working directory.
else
   cd $1       # ==> Otherwise, move to indicated directory.
fi
echo "Initial directory = `pwd`"
swfi=0      # ==> Search finished flag.
deep=0      # ==> Depth of listing.
numdirs=0
zz=0

while [ $swfi != 1 ]   # While flag not set...
do
```

```
   search  # ==> Call function after initializing variables.
done
echo "Total directories = $numdirs"



# ==> Challenge to reader: try to figure out exactly how this script works.
```

Appendix B. A Sed and Awk Micro−Primer

This is a very brief introduction to the **sed** and **awk** text processing utilities. We will deal with only a few basic commands here, but that will suffice for understanding simple sed and awk constructs within shell scripts.

sed: a non−interactive text file editor

awk: a field−oriented pattern processing language

For all their differences, the two utilities share a similar invocation syntax, both use regular expressions (Section 3.15), both read input by default from `stdin`, and both output to `stdout`. These are well−behaved UNIX tools, and they work together well. The output from one can be piped into the other, and their combined capabilities give shell scripts some of the power of Perl.

> **Note:** One important difference between the utilities is that while shell scripts can easily pass arguments to sed, it is more complicated for awk (see Example 2−5).

B.1. Sed

Sed is a non−interactive line editor. It receives text input, whether from `stdin` or from a file, performs certain operations on specified lines of the input, then outputs the result to `stdout` or to a file. Within a shell script, sed is usually one of several tool components in a pipe.

Sed determines which lines of its input that it will operate on from the *address range* passed to it. Specify this address range either by line number or by a pattern to match. For example, *`3d`* signals sed to delete line 3 of the input, and *`/windows/d`* tells sed that you want every line of the input containing a match to "windows" deleted.

Of all the operations in the sed toolkit, we will focus primarily on the three most commonly used ones. These are **p**rinting (to `stdout`), **d**eletion, and **s**ubstitution.

Table B−1. sed operators

Operator	**Name**	**Effect**
`/address-range/p`	print	Print (specified address range)
`/address-range/d`	delete	Delete (specified address range)
`s/pattern1/pattern2/`	substitute	Substitute pattern2 for pattern1
`/address-range/y/pattern1/pattern2/`	transform	replace pattern1 with pattern2 (works just like **tr**)
`g`	global	Operate on *every* pattern match within each matched line of input

> **Note:** Unless the `g` (*global*) operator is appended to a *substitute* command, the substitution operates only on the first instance of a pattern match within each line.

From the command line and in a shell script, a sed operation may require quoting and certain options.

```
sed -e '/^$/d'
# The -e option causes the next string to be interpreted as an instruction.
# The "strong" quotes ('') protect the special characters in the instruction
# from reinterpretation as regular expressions by the body of the script.
# (This reserves RE expansion of the instruction for sed.)
```

> **Note:** Both sed and awk use the `-e` option to specify that the following string is an instruction or set of instructions. If there is only a single instruction contained within the string, then the option may be omitted.

```
sed -n '/xzy/p'
# The -n option tells sed to print only those lines matching the pattern.
# Otherwise all input lines would print.
```

Table B−2. examples

Notation	**Effect**
`8d`	Delete 8th line of input.
`/^$/d`	Delete all blank lines.
`1,/^$/d`	Delete from beginning of input up to, and including first blank line.
`/Jones/p`	Print only lines containing "Jones" (with −n option).
`s/Windows/Linux/`	Substitute "Linux" for first instance of"Windows" found in each input line.
`s/BSOD/stability/g`	Substitute "stability" for every instance of"BSOD" found in each input line.
`/GUI/d`	Delete all lines containing "GUI".
`s/GUI//g`	Delete all instances of "GUI", leaving the remainder of each line intact.

> **Note:** Substituting a zero−length string for another is equivalent to deleting that string within a line of input. This leaves the remainder of the line intact. Applying **`s/GUI//`** to the line **`The most important parts of any application are its GUI and sound effects`** results in
>
> ```
> The most important parts of any application are its  and sound effects
> ```
>
> .

For illustrative examples of sed within shell scripts, see:

1. Example 2−3
2. Example 2−4
3. Example 3−47
4. Example A−2
5. Example 3−55
6. Example 3−60
7. Example A−4
8. Example A−8

9. Example 3−65

For a more extensive treatment of sed, check the appropriate references in the *Bibliography*.

B.2. Awk

Awk is a full−featured text processing language with a syntax reminiscent of **C**. While it possesses an extensive set of operators and capabilities, we will cover only a couple of these here − the ones most useful for shell scripting.

Awk breaks each line of input passed to it into *fields*. By default, a field is a string of consecutive characters separated by white space, though there are options for changing the delimiter. Awk parses and operates on each separate field. This makes awk ideal for handling structured text files, especially tables, data organized into consistent chunks, such as rows and columns.

Strong quoting (single quotes) and curly brackets enclose segments of awk code within a shell script.

```
awk '{print $3}'
# Prints field #3 to stdout.

awk '{print $1 $5 $6}'
# Prints fields #1, #5, and #6.
```

We have just seen the awk **print** command in action. The only other feature of awk we need to deal with here is variables. Awk handles variables similarly to shell scripts, though a bit more flexibly.

```
{ total += ${column_number} }
```

This adds the value of *column_number* to the running total of "total". Finally, to print "total", there needs to be an **END** command to terminate the processing.

```
END { print total }
```

Corresponding to the **END**, there is a **BEGIN**, for a code block to be performed before awk starts processing its input.

For examples of awk within shell scripts, see:

1. Example 3−51
2. Example 3−75
3. Example 3−65
4. Example 2−5

That's all the awk we'll cover here folks, but there's lots more to learn. See the appropriate references in the *Bibliography*.

Appendix C. A Sample `.bashrc` File

The `~/.bashrc` file determines the behavior of the shell, and of shell scripts. A proper understanding of this file can lead to more effective use of scripts.

Emmanuel Rouat contributed the following very elaborate `.bashrc` file. He wrote it for a Solaris system, but it (mostly) works for other flavors of UNIX as well. Study this file carefully, and feel free to reuse code snippets and functions from it in your own `.bashrc` file and even in your scripts.

Example C−1. Sample `.bashrc` file

```
#===============================================================
#
# PERSONAL $HOME/.bashrc FILE for bash-2.04 (or later)
#              by Emmanuel Rouat
#
# This file is read (normally) by interactive shells only.
# Here is the place to define your aliases, functions and
# other interactive features like your prompt.
#
# This file was designed for Solaris
#
#===============================================================



#-----------------------------------
# Source global definitions (if any)
#-----------------------------------

if [ -f /etc/bashrc ]; then
        . /etc/bashrc  # Read system bash init file, if exists.
fi


#-------------------------------------------------------------
# Automatic setting of $DISPLAY (if not set already)
# This works for linux and solaris - your mileage may vary....
#-------------------------------------------------------------


if [ -z ${DISPLAY:=""} ]; then
    DISPLAY=$(who am i)
    DISPLAY=${DISPLAY%%\!*}
    if [ -n "$DISPLAY" ]; then
        export DISPLAY=$DISPLAY:0.0
    else
        export DISPLAY=":0.0"  # fallback
    fi
fi


#---------------
# Some settings
#---------------

set -o notify
```

```
set -o noclobber
set -o ignoreeof
set -o nounset
#set -o xtrace          # useful for debuging

shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s histappend histreedit
shopt -s extglob        # useful for programmable completion



#-----------------------
# Greeting, motd etc...
#-----------------------

# Define some colors first:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'              # No Color

# Looks best on a black background.....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on ${RED}$DISPLAY${NC}\n"
date


function _exit()        # function to run upon exit of shell
{
    echo -e "${RED}Hasta la vista, baby${NC}"
}
trap _exit 0

#---------------
# Shell prompt
#---------------


function fastprompt()
{
    unset PROMPT_COMMAND
    case $TERM in
        xterm | rxvt | dtterm )
            PS1="[\h] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        *)
            PS1="[\h] \W > " ;;
    esac
}


function powerprompt()
{
    _powerprompt()
    {
```

```
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
        TIME=$(date +%H:%M)
    }

    PROMPT_COMMAND=_powerprompt
    case $TERM in
        xterm | dtterm | rxvt  )
            PS1="${cyan}[\$TIME \$LOAD]$NC\n[\h \#] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        linux )
            PS1="${cyan}[\$TIME - \$LOAD]$NC\n[\h \#] \w > " ;;
        * )
            PS1="[\$TIME - \$LOAD]\n[\h \#] \w > " ;;
    esac
}

powerprompt     # this is the default prompt - might be slow
                # If too slow, use fastprompt instead....



#===============================================================
#
# ALIASES AND FUNCTIONS
#
# Arguably, some functions defined here are quite big
# (ie 'lowercase') but my workstation has 512Meg of RAM, so .....
# If you want to make this file smaller, these functions can
# be converted into scripts.
#
#===============================================================

#-------------------
# Personnal Aliases
#-------------------

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias h='history'
alias j='jobs -l'
alias r='rlogin'
alias which='type -a'
alias ..='cd ..'
alias path='echo -e ${PATH//:/\\n}'
alias print='/usr/bin/lp -o nobanner -d $LPDEST'

alias la='ls -Al'
alias lr='ls -lR'
alias lt='ls -ltr'
alias lm='ls -al |more'

# spelling typos

alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'


#-----------------------------------------
# Environment dependent aliases/variables
```

```
#-----------------------------------------


if [ -d $FREE/bin ] ; then      # use gnu/free stuff
# Condition test unnecessary on a Linux or BSD system.

    alias vi='vim'
    alias csh='tcsh'
    alias du='du -h'
    alias df='df -kh'
    alias ls='ls -hF --color'
    alias lx='ls -lXB'
    alias lk='ls -lSr'
    alias pjet='enscript -h -G -fCourier9 -d $LPDEST '
    alias background='xv -root -quit -max -rmode 5'

    alias more='less'
    export PAGER=less
    export LESSCHARSET='latin1'
    export LESSOPEN='|lesspipe.sh %s'
    export LESS='-i  -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

else                            # use regular solaris stuff

    alias df='df -k'
    alias ls='ls -F'

fi


#----------------
# a few fun ones
#----------------

function xtitle ()
{
    case $TERM in
        xterm* | dtterm | rxvt)
            echo -n -e "\033]0;$*\007" ;;
        *)  ;;
    esac
}

alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'
alias ncftp="xtitle ncFTP ; ncftp"


#---------------
# and functions
#---------------

function man ()
{
    xtitle The $(basename $1|tr -d .[:digit:]) manual
    /usr/bin/man -a "$*"
}

#-----------------------------------------
# Environment dependent functions
#-----------------------------------------
```

```
# Note: we mustn't mix these with alias definitions in the same 'if/fi'
# construct because alias expansion wouldn't occur in some functions here,
# like 'll' that uses ls (which is an alias).


if [ -d $FREE/bin ] ; then      # use gnu/free stuff

    function ll(){ ls -l  $*| egrep "^d" ; ls -lh  $* 2>&-| egrep -v "^d|total "; }
    function xemacs() { { command xemacs -private $* 2>&- & } && disown ;}
    function te()  # wrapper around xemacs/gnuserv
    {
        if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
            gnuclient -q $@;
        else
            ( xemacs $@ & );
        fi
    }

else                            # use solaris stuff

    function ll(){ ls -l $* |egrep "^d"; ls -l $* 2>&- |egrep -v "^d|total" ;}
    function lk() { \ls -lF  $* | egrep -v "^d|^total" | sort -n -k 5,5 ;}
    function te() { ( dtpad "$@" &)  ;}

fi


#-----------------------------------
# File & strings related functions:
#-----------------------------------

function ff() { find . -name '*'$1'*' ; }
function fe() { find . -name '*'$1'*' -exec $2 {} \; ; }
function fstr() # find a string in a set of files
{
    if [ "$#" -gt 2 ]; then
        echo "Usage: fstr \"pattern\" [files] "
    return;
    fi
    find . -type f -name "${2:-*}" -print | xargs grep -n "$1"
}
function cuttail() # cut last n lines in file
{
    nlines=$1
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $2
}

function lowercase()  # move filenames to lowercase
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
```

```
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # swap 2 filenames around
{
    local TMPFILE=tmp.$$
    mv $1 $TMPFILE
    mv $2 $1
    mv $TMPFILE $2
}


# Process/system related functions:

alias my_ps='/usr/bin/ps -u "$USER" -o user,pid,ppid,pcpu,pmem,args'
function pp() { my_ps | nawk '!/nawk/ && $0~pat' pat=${1:-".*"} ; }
function killps()       # Kill process by name
{                       # works with gawk too
    local pid pname sig="-TERM" # default signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Usage: killps [-SIGNAL] pattern"
        return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps | nawk '!/nawk/ && $0~pat { print $2 }' pat=${!#}) ; do
        pname=$(my_ps | nawk '$2~var { print $6 }' var=$pid )
        if ask "Kill process $pid <$pname> with signal $sig ? "
            then kill $sig $pid
        fi
    done
}

function ii()   # get current host related info
{
    echo -e "\nYou are logged on ${RED}$HOST"
    echo -e "\nAdditionnal information:$NC " ; uname -a
    echo -e "\n${RED}IP Address :$NC" ; ypmatch $HOSTNAME hosts
    echo -e "\n${RED}Users logged on:$NC " ; /usr/ucb/users
    echo -e "\n${RED}Current date :$NC " ; date
    echo -e "\n${RED}Machine stats :$NC " ; uptime
    echo -e "\n${RED}Memory stats :$NC " ; vmstat
    echo -e "\n${RED}NIS Server :$NC " ; ypwhich
    echo
}
function corename()   # get name of app that created core
{
    local file name;
    file=${1:-"core"}
    set -- $(adb $file < /dev/null 2>&1 | sed 1q)
    name=${7#??}
    echo $file: ${name%??}
}
# Misc utilities:

function repeat()       # repeat n times command
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do
        eval "$@";
```

```
    done
}


function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}



#=========================================================================
#
# PROGRAMMABLE COMPLETION - ONLY IN BASH-2.04
#
#=========================================================================

if [ "${BASH_VERSION%.*}" \< "2.04" ]; then
    echo "No programmable completion available"
    return
fi

shopt -s extglob        # necessary

complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
complete -A command    nohup exec eval trace truss strace sotruss gdb
complete -A command    command type which
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # currently same as builtins
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir

complete -f -X '*.gz'   gzip
complete -f -X '!*.ps'  gs ghostview gv
complete -f -X '!*.pdf' acroread
complete -f -X '!*.+(gif|jpg|jpeg|GIF|JPG|bmp)' xv gimp


_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # if prev argument is -f, return possible filename completions.
    # we could be a little smarter here and return matches against
    # `makefile Makefile *.mk', whatever exists
```

```
    case "$prev" in
        -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # if we want an option, return the possible posix options
    case "$cur" in
        -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac

    # make reads `makefile' before `Makefile'
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk               # local convention
    fi

    # before we scan for targets, see if a makefile name was specified
    # with -f
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -*f ]]; then
            eval makef=${COMP_WORDS[i+1]}       # eval for tilde expansion
            break
        fi
    done

        [ -z "$makef" ] && makef=$mdef

    # if we have a partial word to complete, restrict completions to
    # matches of that word
    if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi

    # if we don't want to use *.mk, we can take out the cat and use
    # test -f $makef and input redirection
    COMPREPLY=( $(cat $makef 2>/dev/null | awk 'BEGIN {FS=":"} /^[^.#   ][^=]*:/ {print $1}' | tr -s ' ' '\012' | sort -u | eval $gcmd ) )
}

complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake


_configure_func ()
{
    case "$2" in
        -*)     ;;
        *)      return ;;
    esac

    case "$1" in
        \~*)    eval cmd=$1 ;;
        *)      cmd="$1" ;;
    esac

    COMPREPLY=( $("$cmd" --help | awk '{if ($1 ~ /--.*/) print $1}' | grep ^"$2" | sort -u) )
}

complete -F _configure_func configure

_killps ()
{
    local cur prev
    COMPREPLY=()
```

```
    cur=${COMP_WORDS[COMP_CWORD]}

    # get a list of processes (the first sed evaluation
    # takes care of swapped out processes, the second
    # takes care of getting the basename of the process)
    COMPREPLY=( $( /usr/bin/ps −u $USER −o comm  | \
        sed −e '1,1d' −e 's#[]\[]##g' −e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}
complete −F _killps killps



# Local Variables:
# mode:shell−script
# sh−shell:bash
# End:
```

Appendix D. Copyright

A Korean translation of this HOWTO is in process. If you wish to translate it into another language, please feel free to do so, subject to the terms stated above. The author would appreciate being notified of such efforts.

If this document is incorporated into a printed book, the author requests a courtesy copy. This is a request, not a requirement.

Notes

[1]
: Many of the features of *ksh88*, not the newer *ksh93* have been merged into Bash.

[2]
: The words "argument" and "parameter" are often used interchangeably. In the context of this document, they have the same precise meaning, that of a variable passed to a script or function.

[3]
: A flag is an argument that acts as a signal, switching script behaviors on or off.

[4]
: The *print queue* is the group of jobs "waiting in line" to be printed.

[5]
: NAND is the logical "not−and" operator. Its effect is somewhat similar to subtraction.

[6]
: A *file descriptor* is simply a number that the operating system assigns to an open file to keep track of it. Consider it a simplified version of a file pointer. It is analogous to a *file handle* in C.

[7]
: Using `file descriptor 5` might cause problems. When Bash forks a child process, as with **exec**, the child inherits fd 5 (see Chet Ramey's archived e−mail, SUBJECT: RE: File descriptor 5 is held open). Best leave this particular fd alone.