Bash Tutorial

Dr James Martin
Associate Professor
School of Computing
Clemson, SC
jmarty@clemson.edu

Last update: 3/6/2018

Source of information for an;ything Linux is at The Linux Documentation Project or at GNU's site;

- The Linux Documentation Project's home page : www.tldp.org
- The tldp's Advanced Bash-Scripting Guide is a very useful source of information
  - http://www.tldp.org/LDP/abs/html/index.html
  - http://www.tldp.org/LDP/abs/html/abs-guide.html
  - http://www.tldp.org/LDP/abs/abs-guide.pdf

And for anything GNU:

- www.gnu.org
- https://www.gnu.org/manual/manual.html
- Gnu's Bash home : https://www.gnu.org/software/bash/
  - https://www.gnu.org/software/bash/manual/html_node/index.html
  - https://www.gnu.org/software/bash/manual/bash.html
  - https://www.gnu.org/software/bash/manual/bash.pdf

This tutorial is meant to be used to support courses taught at Clemson University. Much of the material is from GNU's bash manual.

Introduction

Bash is an acronym for 'Bourne-Again Shell'. The Bourne shell is a Unix shell written by Stephen Bourne. Bash supports all Bourne builtin commands, uses Posix rules for syntax related to standard. The Bourne shell is a Unix shell written by Stephen Bourne. Bash supports all Bourne builtin commands, uses Posix rules for syntax related to standard shells.

Beyond the shell syntax, the building blocks are:

- Commands
- Control structures
- Shell functions
- Shell parameters
- Shell expansions
- Redirections

The following summarizes the shell's operation as it reads and processes a command:

- Reads its input from either: a file, a string supplied as an argument, or from the user's terminal.
- Breaks the input into words and operators (i.e., tokens), making sure to follow the quoting rules. These tokens are separated by metacharacters.
    - Alias expansion is performed by this step
- Parses the tokens into simple and compound commands
- Performs the various shell expansions, breaking the expanded tokens into lists of filenames and commands and arguments.
- Performs any necessary redirections which removes the redirection operators and their operands from the argument list.
- Executes the command
- Possibly waits for the command to complete, or if the command is operating in background mode, moves onto the next command without waiting.

Bash - bits and pieces….

Quoting Rules/Syntax:

3.1.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the shell metacharacters (see Chapter 2 [Definitions], page 3) has special meaning to the shell and must be quoted if it is to represent itself. When the command history expansion facilities are being used (see Section 9.3 [History Interaction], page 139), the history expansion character, usually '!', must be quoted to prevent history expansion. See Section 9.1 [Bash History Facilities], page 137, for more details concerning history expansion.

There are three quoting mechanisms: the escape character, single quotes, and double quotes.

3.1.2.1 Escape Character

A non-quoted backslash '\' is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of `newline`. If a `\newline` pair appears, and the backslash itself is not quoted, the `\newline` is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

3.1.2.2 Single Quotes

Enclosing characters in single quotes (''') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

3.1.2.3 Double Quotes

Enclosing characters in double quotes ('"') preserves the literal value of all characters within the quotes, with the exception of '$', '`', '\', and, when history expansion is enabled, '!'. When the shell is in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 96), the '!' has no special meaning within double quotes, even when history expansion is enabled. The characters '$' and '`' retain their special meaning within double quotes (see Section 3.5 [Shell Expansions], page 21). The backslash retains its special meaning only when followed by one of the following characters: '$', '`', '"', '\', or `newline`. Within double quotes, backslashes that are followed by one of these characters are removed. Backslashes preceding characters without a special meaning are left unmodified. A double quote may be quoted within double quotes by preceding it with a backslash. If enabled, history expansion will be performed unless an '!' appearing in double quotes is escaped using a backslash. The backslash preceding the '!' is not removed.

The special parameters '*' and '@' have special meaning when in double quotes (see Section 3.5.3 [Shell Parameter Expansion], page 23).

3.1.2.4 ANSI-C Quoting

Words of the form `$'string'` are treated specially. The word expands to string, with backslash-escaped characters replaced as specified by the ANSI C standard. Backslash escape sequences, if present, are decoded as follows:

\a	alert (bell)
\b	backspace
\e \E	an escape character (not ANSI C)
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\'	single quote
\"	double quote
\?	question mark
nnn	the eight-bit character whose value is the octal value *nnn* (one to three digits)
\x*HH*	the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)
\u*HHHH*	the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHH* (one to four hex digits)
\U*HHHHHHHH*	

3.2 Shell Commands

A simple shell command such as `echo a b c` consists of the command itself followed by arguments, separated by spaces.

More complex shell commands are composed of simple commands arranged together in a variety of ways: in a pipeline in which the output of one command becomes the input of a second, in a loop or conditional construct, or in some other grouping.

3.2.1 Simple Commands

A simple command is the kind of command encountered most often. It's just a sequence of words separated by `blanks`, terminated by one of the shell's control operators (see Chapter 2 [Definitions], page 3). The first word generally specifies a command to be executed, with the rest of the words being that command's arguments.

The return status (see Section 3.7.5 [Exit Status], page 39) of a simple command is its exit status as provided by the POSIX 1003.1 `waitpid` function, or 128+*n* if the command was terminated by signal *n*.

3.2.2 Pipelines

A `pipeline` is a sequence of one or more commands separated by one of the control operators '`|`' or '`|&`'.

The format for a pipeline is

```
[time [-p]] [!] command1 [ | or |& command2 ] ...
```

The output of each command in the pipeline is connected via a pipe to the input of the next command. That is, each command reads the previous command's output. This connection is performed before any redirections specified by the command.

If '`|&`' is used, *command1*'s standard error, in addition to its standard output, is connected to *command2*'s standard input through the pipe; it is shorthand for `2>&1 |`. This implicit redirection of the standard error to the standard output is performed after any redirections specified by the command.

The reserved word `time` causes timing statistics to be printed for the pipeline once it finishes. The statistics currently consist of elapsed (wall-clock) time and user and system time consumed by the command's execution. The `-p` option changes the output format to that specified by POSIX. When the shell is in `POSIX` mode (see Section 6.11 [Bash POSIX Mode], page 96), it does not recognize `time` as a reserved word if the next token begins with a '`-`'. The `TIMEFORMAT` variable may be set to a format string that specifies how the timing information should be displayed. See Section 5.2 [Bash Variables], page 71, for a description of the available formats. The use of `time` as a reserved word permits the timing of shell builtins, shell functions, and pipelines. An external `time` command cannot time these easily.

When the shell is in `POSIX` mode (see Section 6.11 [Bash POSIX Mode], page 96), `time` may be followed by a newline. In this case, the shell displays the total user and system time consumed by the shell and its children. The `TIMEFORMAT` variable may be used to specify the format of the time information.

If the pipeline is not executed asynchronously (see Section 3.2.3 [Lists], page 9), the shell waits for all commands in the pipeline to complete.

Shell Parameters -

A parameter is an entity that stores values. It can be a name, a number, or one of the special characters listed below. A variable is a parameter denoted by a name. A variable has a value and zero or more attributes. Attributes are assigned using the declare builtin command A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the unset builtin command. A variable may be assigned to by a statement of the form
name=[value]

If value is not given, the variable is assigned the null string. All values undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (detailed below). If the variable has its integer attribute set, then value is evaluated as an arithmetic expression even if the $((...)) expansion is not used. Word splitting is not performed, with the exception of "$@" as explained below. Filename expansion is not performed. Assignment statements may also appear as arguments to the alias, declare, typeset, export, readonly, and local builtin commands (declaration commands). When in posix mode these builtins may appear in a command after one or more instances of the command builtin and retain these assignment statement
properties.

In the context where an assignment statement is assigning a value to a shell variable orarray index, the '+=' operator can be used to append to oradd to the variable's previous value. This includes arguments to builtin commands such asdeclare that accept assignment statements (declaration commands). When '+=' is applied to a variable for which the integer attribute has been set, value is evaluated as an arithmetic expression and added to the variable's current value, which is also evaluated. When '+=' is applied to an array variable using compound assignment the variable's value is not unset (as it is when using '='), and new values are appended to the array beginning at one greater than the array's maximum index (for indexed arrays), or added as additional key-value pairs in an associative array. When applied to a string-valued variable, value is expanded and appended to the variable's value.

A variable can be assigned the nameref attribute using the -n option to the declare or local builtin commands to create a nameref, or a reference to another variable. This allows variables to be manipulated indirectly. Whenever the nameref variable is referenced, assigned to, unset, or has its attributes modified (other than using or changing the nameref attribute itself), the operation is actually performed on the variable specified by the nameref variable's value. A nameref is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function.

For instance, if a variable name is passed to a shell function as its first argument, running declare -n ref=$1 inside the function creates a nameref variable ref whose value is the variable name passed as the first argument. References and assignments to ref, and changes to its attributes, are treated as references, assignments, and attribute modifications to the variable whose name was passed as $1.

If the control variable in a for loop has the nameref attribute, the list of words can be a list of shell variables, and a name reference will be established for each word in the list, in turn, when the loop is executed. Array variables cannot be given the nameref attribute. However, nameref variables can reference array variables and subscripted array variables. Namerefs can be unset using the -n option to the unset builtin Otherwise, if unset is executed with the name of a nameref variable as an argument, the variable referenced by the nameref variable will be unset.

A positional parameter is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the set builtin command. Positional parameter N may be referenced as ${N}, or as $N when N consists of a single digit. Positional parameters may not be assigned to with assignment statements. The set and shift builtins are used to set and unset them (see Chapter 4 [Shell Builtin Commands], page 42). The positional
Chapter 3: Basic Shell Features 20
parameters are temporarily replaced when a shell function is executed (see Section 3.3 [Shell Functions], page 17).
When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.

3.4.2 Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

`*` ($*) Expands to the positional parameters, starting from one. When the expansion is not within double quotes, each positional parameter expands to a separate word. In contexts where it is performed, those words are subject to further word splitting and pathname expansion. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable. That is, `"$*"` is equivalent to `"$1c$2c..."`, where c is the first character of the value of the `IFS` variable. If `IFS` is unset, the parameters are separated by spaces. If `IFS` is null, the parameters are joined without intervening separators.

`@` ($@) Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word. That is, `"$@"` is equivalent to `"$1" "$2" ....` If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the original word, and the expansion of the last parameter is joined with the last part of the original word. When there are no positional parameters, `"$@"` and `$@` expand to nothing (i.e., they are removed).

`#` ($#) Expands to the number of positional parameters in decimal.

`?` ($?) Expands to the exit status of the most recently executed foreground pipeline.

`-` ($-, a hyphen.) Expands to the current option flags as specified upon invocation, by the `set` builtin command, or those set by the shell itself (such as the `-i` option).

`$` ($$) Expands to the process ID of the shell. In a () subshell, it expands to the process ID of the invoking shell, not the subshell.

`!` ($!) Expands to the process ID of the job most recently placed into the background, whether executed as an asynchronous command or using the `bg` builtin (see Section 7.2 [Job Control Builtins], page 101).

`0` ($0) Expands to the name of the shell or shell script. This is set at shell initialization. If Bash is invoked with a file of commands (see Section 3.8 [Shell Scripts], page 40), `$0` is set to the name of that file. If Bash is started with the `-c` option (see Section 6.1 [Invoking Bash], page 82), then `$0` is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the filename used to invoke Bash, as given by argument zero.

`_` ($_, an underscore.) At shell startup, set to the absolute pathname used to invoke the shell or shell script being executed as passed in the environment

Expansion is performed on the command line after it has been split into tokens. There are seven kinds of expansion performed:

- brace expansion
- tilde expansion
- parameter and variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion

The order of expansions is: brace expansion; tilde expansion, parameter and variable expansion, arithmetic expansion, and command substitution (done in a left-to-right fashion);
word splitting; and filename expansion.

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to filename expansion but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional preamble, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional postscript. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right. Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example,

```
bash$ echo a{d,c,b}e
ade ace abe
```

If a word begins with an unquoted tilde character (‘~’), all of the characters up to the first unquoted slash (or all characters, if there is no unquoted slash) are considered a tilde-prefix. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name. If this login name is the null string, the tilde is replaced with the value of the HOME shell variable. If HOME is unset, the home directory of the user executing the shell is substituted instead. Otherwise, the tilde-prefix is replaced with the home directory associated with the specified login name.

The ‘$’ character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name. When braces are used, the matching ending brace is the first ‘}’ not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion. The basic form of parameter expansion is ${parameter}. The value of parameter is substituted. The parameter is a shell parameter as described above

The braces are required when parameter is a positional parameter with more than one digit, or when parameter is followed by a character that is not to be interpreted as part of its name. If the first character of parameter is an exclamation point (!), and parameter is not a nameref, it introduces a level of variable indirection. Bash uses the value of the variable formed from the rest of parameter as the name of the variable; this variable is then expanded and that value is used in the rest of the substitution, rather than the value of parameter itself. This is known as indirect expansion. If parameter is a nameref, this expands to the name of the variable referenced by parameter instead of performing the complete indirect expansion. The exceptions to this are the expansions of ${!prefix*} and ${!name[@]} described below. The exclamation point must immediately follow the left brace in order to introduce indirection.

Here are some examples illustrating substring expansion on parameters and
subscripted arrays:
$ string=01234567890abcdefgh
$ echo ${string:7}
7890abcdefgh
$ echo ${string:7:0}
$ echo ${string:7:2}
78
$ echo ${string:7:-2}
7890abcdef

The following examples illustrate substring expansion using positional parameters:

$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f g h

$ echo ${@:7}

7 8 9 0 a b c d e f g h

$ echo ${@:7:0}

$ echo ${@:7:2}

7 8

$ echo ${@:7:-2}

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows:

$(command)
or
‘command‘

Bash performs the expansion by executing command in a subshell environment and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting. The command substitution $(cat file) can be replaced by the equivalent but faster $(< file). When the old-style backquote

form of substitution is used, backslash retains its literal meaning except when followed by '$', '`', or '\'. The first backquote not preceded by a backslash terminates the command substitution. When using the $(command) form, all characters between the parentheses make up the command; none are treated specially. Command substitutions may be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes. If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results.

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

$((expression))

The expression is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All tokens in the expression undergo parameter and variable expansion, command substitution, and quote removal. The result is treated as the arithmetic expression to be evaluated. Arithmetic expansions may be nested. The evaluation is performed according to the rules listed below If the expression is invalid, Bash prints a message indicating failure to the standard error and no substitution occurs.

After word splitting, unless the -f option has been set Bash scans each word for the characters '*', '?', and '['. If one of these characters appears, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames matching the pattern, If no matching filenames are found, and the shell option nullglob is disabled, the word is left unchanged. If the nullglob option is set, and no matches are found, the word is removed. If the failglob shell option is set, and no matches are found, an error message is printed and the command is not executed. If the shell option nocaseglob is enabled, the match is performed without regard to the case of alphabetic characters.

4.3.1 The Set Builtin

This builtin is so complicated that it deserves its own section. `set` allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables.

`set`

```
set [--abefhkmnptuvxBCEHPT] [-o option-name] [argument ...]
set [+abefhkmnptuvxBCEHPT] [+o option-name] [argument ...]
```

If no options or arguments are supplied, `set` displays the names and values of all shell variables and functions, sorted according to the current locale, in a format that may be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In POSIX mode, only shell variables are listed.

When options are supplied, they set or unset shell attributes. Options, if specified, have the following meanings:

`-a` Each variable or function that is created or modified is given the export attribute and marked for export to the environment of subsequent commands.

`-b` Cause the status of terminated background jobs to be reported immediately, rather than before printing the next primary prompt.

`-e` Exit immediately if a pipeline (see Section 3.2.2 [Pipelines], page 8), which may consist of a single simple command (see Section 3.2.1 [Simple Commands], page 8), a list (see Section 3.2.3 [Lists], page 9), or a compound command (see Section 3.2.4 [Compound Commands], page 9) returns a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a `while` or `until` keyword, part of the test in an `if` statement, part of any command executed in a `&&` or `||` list except the command following the final `&&` or `||`, any command in a pipeline but the last, or if the command's return status is being inverted with `!`. If a compound command other than a subshell returns a non-zero status because a command failed while `-e` was being ignored, the shell does not exit. A trap on `ERR`, if set, is executed before the shell exits.

This option applies to the shell environment and each subshell environment separately (see Section 3.7.3 [Command Execution Environment], page 37), and may cause subshells to exit before executing all the commands in the subshell.

If a compound command or shell function executes in a context where `-e` is being ignored, none of the commands executed within the compound command or function body will be affected by the `-e` setting, even if `-e` is set and a command returns a failure status. If a compound command or shell function sets `-e` while executing in a context where `-e` is ignored, that setting will not have any effect until the compound command or the command containing the function call completes.

`-f` Disable filename expansion (globbing).

`-h` Locate and remember (hash) commands as they are looked up for execution. This option is enabled by default.

`-k` All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.

`-m` Job control is enabled (see Chapter 7 [Job Control], page 100). All processes run in a separate process group. When a background job completes, the shell prints a line containing its exit status.

`-n` Read commands but do not execute them. This may be used to check a script for syntax errors. This option is ignored by interactive shells.

`-o` *option-name*
Set the option corresponding to *option-name*:

`allexport`
Same as `-a`.

`braceexpand`
Same as `-B`.

`emacs` Use an `emacs`-style line editing interface (see Chapter 8 [Command Line Editing], page 104). This also affects the editing interface used for `read -e`.

`errexit` Same as `-e`.

`errtrace` Same as `-E`.

`functrace`
Same as `-T`.

`hashall` Same as `-h`.

`histexpand`
Same as `-H`.

`history` Enable command history, as described in Section 9.1 [Bash History Facilities], page 137. This option is on by default in interactive shells.

`ignoreeof`
An interactive shell will not exit upon reading EOF.

`keyword` Same as `-k`.

`monitor` Same as `-m`.

`noclobber`
Same as `-C`.

`noexec` Same as `-n`.

`noglob` Same as `-f`.

`nolog` Currently ignored.

`notify` Same as `-b`.

`nounset` Same as `-u`.

`onecmd` Same as `-t`.

`physical` Same as `-P`.

`pipefail` If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

`posix` Change the behavior of Bash where the default operation differs from the POSIX standard to match the standard (see Section 6.11 [Bash POSIX Mode], page 96). This is intended to make Bash behave as a strict superset of that standard.

`privileged`
Same as `-p`.

`verbose` Same as `-v`.

`vi` Use a `vi`-style line editing interface. This also affects

4.3.2 The Shopt Builtin

This builtin allows you to change additional shell optional behavior.

`shopt`

```
shopt [-pqsu] [-o] [optname ...]
```

Toggle the values of settings controlling optional shell behavior. The settings can be either those listed below, or, if the `-o` option is used, those available with the `-o` option to the `set` builtin command (see Section 4.3.1 [The Set Builtin], page 60). With no options, or with the `-p` option, a list of all settable options is displayed, with an indication of whether or not each is set. The `-p` option causes output to be displayed in a form that may be reused as input. Other options have the following meanings:

`-s` Enable (set) each *optname*.

`-u` Disable (unset) each *optname*.

`-q` Suppresses normal output; the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are given with `-q`, the return status is zero if all *optnames* are enabled; non-zero otherwise.

`-o` Restricts the values of *optname* to be those defined for the `-o` option to the `set` builtin (see Section 4.3.1 [The Set Builtin], page 60).

If either `-s` or `-u` is used with no *optname* arguments, `shopt` shows only those options which are set or unset, respectively.

Unless otherwise noted, the `shopt` options are disabled (off) by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a valid shell option.

The list of `shopt` options is:

`autocd` If set, a command name that is the name of a directory is executed as if it were the argument to the `cd` command. This option is only used by interactive shells.

5.1 Bourne Shell Variables

Bash uses certain shell variables in the same way as the Bourne shell. In some cases, Bash assigns a default value to the variable.

`CDPATH` A colon-separated list of directories used as a search path for the `cd` builtin command.

`HOME` The current user's home directory; the default for the `cd` builtin command. The value of this variable is also used by tilde expansion (see Section 3.5.2 [Tilde Expansion], page 22).

`IFS` A list of characters that separate fields; used when the shell splits words as part of expansion.

`MAIL` If this parameter is set to a filename or directory name and the `MAILPATH` variable is not set, Bash informs the user of the arrival of mail in the specified file or Maildir-format directory.

`MAILPATH` A colon-separated list of filenames which the shell periodically checks for new mail. Each list entry can specify the message that is printed when new mail arrives in the mail file by separating the filename from the message with a '`?`'. When used in the text of the message, `$_` expands to the name of the current mail file.

`OPTARG` The value of the last option argument processed by the `getopts` builtin.

`OPTIND` The index of the last option argument processed by the `getopts` builtin.

`PATH` A colon-separated list of directories in which the shell looks for commands. A zero-length (null) directory name in the value of `PATH` indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon.

`PS1` The primary prompt string. The default value is '`\s-\v\$ `'. See Section 6.9 [Controlling the Prompt], page 94, for the complete list of escape sequences that are expanded before `PS1` is displayed.

`PS2` The secondary prompt string. The default value is '`> `'.

5.2 Bash Variables

These variables are set or used by Bash, but other shells do not normally treat them specially.

A few variables used by Bash are described in different chapters: variables for controlling the job control facilities (see Section 7.3 [Job Control Variables], page 103).

`BASH` The full pathname used to execute the current instance of Bash.

`BASHOPTS` A colon-separated list of enabled shell options. Each word in the list is a valid argument for the `-s` option to the `shopt` builtin command (see Section 4.3.2 [The Shopt Builtin], page 64). The options appearing in `BASHOPTS` are those reported as '`on`' by '`shopt`'. If this variable is in the environment when Bash starts up, each shell option in the list will be enabled before reading any startup files. This variable is readonly.

`BASHPID` Expands to the process ID of the current Bash process. This differs from `$$` under certain circumstances, such as subshells that do not require Bash to be re-initialized.

`BASH_ALIASES`
An associative array variable whose members correspond to the internal list of aliases as maintained by the `alias` builtin. (see Section 4.1 [Bourne Shell Builtins], page 42). Elements added to this array appear in the alias list; however, unsetting array elements currently does not cause aliases to be removed from the alias list. If `BASH_ALIASES` is unset, it loses its special properties, even if it is subsequently reset.

`BASH_ARGC`
An array variable whose values are the number of parameters in each frame of the current bash execution call stack. The number of parameters to the current subroutine (shell function or script executed with . or `source`) is at the top of the stack. When a subroutine is executed, the number of parameters passed is pushed onto `BASH_ARGC`. The shell sets `BASH_ARGC` only when in extended debugging mode (see Section 4.3.2 [The Shopt Builtin], page 64, for a description of the `extdebug` option to the `shopt` builtin).

`BASH_ARGV`
An array variable containing all of the parameters in the current bash execution call stack. The final parameter of the last subroutine call is at the top of the stack; the first parameter of the initial call is at the bottom. When a subroutine is executed, the parameters supplied are pushed onto `BASH_ARGV`. The shell sets `BASH_ARGV` only when in extended debugging mode (see Section 4.3.2 [The Shopt Builtin], page 64, for a description of the `extdebug` option to the `shopt` builtin).

`BASH_CMDS`
An associative array variable whose members correspond to the internal hash table of commands as maintained by the `hash` builtin (see Section 4.1 [Bourne Shell Builtins], page 42). Elements added to this array appear in the hash table; however, unsetting array elements currently does not cause command names to be removed from the hash table. If `BASH_CMDS` is unset, it loses its special properties, even if it is subsequently reset.

`BASH_COMMAND`
The command currently being executed or about to be executed, unless the shell is executing a command as the result of a trap, in which case it is the command executing at the time of the trap.

6.4 Bash Conditional Expressions

Conditional expressions are used by the `[[` compound command and the `test` and `[` builtin commands.

Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Bash handles several filenames specially when they are used in expressions. If the operating system on which Bash is running provides these special files, Bash will use them; otherwise it will emulate them internally with this behavior: If the *file* argument to one of the primaries is of the form `/dev/fd/N`, then file descriptor *N* is checked. If the *file* argument to one of the primaries is one of `/dev/stdin`, `/dev/stdout`, or `/dev/stderr`, file descriptor 0, 1, or 2, respectively, is checked.

When used with `[[`, the '<' and '>' operators sort lexicographically using the current locale. The `test` command uses ASCII ordering.

Unless otherwise specified, primaries that operate on files follow symbolic links and operate on the target of the link, rather than the link itself.

`-a file` True if *file* exists.

`-b file` True if *file* exists and is a block special file.

`-c file` True if *file* exists and is a character special file.

`-d file` True if *file* exists and is a directory.

`-e file` True if *file* exists.

`-f file` True if *file* exists and is a regular file.

`-g file` True if *file* exists and its set-group-id bit is set.

`-h file` True if *file* exists and is a symbolic link.

`-k file` True if *file* exists and its "sticky" bit is set.

`-p file` True if *file* exists and is a named pipe (FIFO).

`-r file` True if *file* exists and is readable.

`-s file` True if *file* exists and has a size greater than zero.

`-t fd` True if file descriptor *fd* is open and refers to a terminal.

`-u file` True if *file* exists and its set-user-id bit is set.

`-w file` True if *file* exists and is writable.

`-x file` True if *file* exists and is executable.

`-G file` True if *file* exists and is owned by the effective group id.

`-L file` True if *file* exists and is a symbolic link.

`-N file` True if *file* exists and has been modified since it was last read.

`-O file` True if *file* exists and is owned by the effective user id.

`-S file` True if *file* exists and is a socket.

`file1 -ef file2`
True if *file1* and *file2* refer to the same device and inode numbers.

`file1 -nt file2`
True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.

`file1 -ot file2`
True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

`-o optname`
True if the shell option *optname* is enabled. The list of options appears in the description of the `-o` option to the `set` builtin (see Section 4.3.1 [The Set Builtin], page 40).

`-v varname`
True if the shell variable *varname* is set (has been assigned a value).

`-R varname`
True if the shell variable *varname* is set and is a name reference.

`-z string` True if the length of *string* is zero.

`-n string`
`string` True if the length of *string* is non-zero.

`string1 == string2`
`string1 = string2`
True if the strings are equal. When used with the `[[` command, this performs pattern matching as described above (see Section 3.2.4.2 [Conditional Constructs], page 10).

'=' should be used with the `test` command for POSIX conformance.

`string1 != string2`
True if the strings are not equal.

`string1 < string2`
True if *string1* sorts before *string2* lexicographically.

`string1 > string2`
True if *string1* sorts after *string2* lexicographically.

`arg1 OP arg2`
`OP` is one of '`-eq`', '`-ne`', '`-lt`', '`-le`', '`-gt`', or '`-ge`'. These arithmetic binary operators return true if *arg1* is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to *arg2*, respectively. *Arg1* and *arg2* may be positive or negative integers.

6.5 Shell Arithmetic

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by using the ((compound command, the `let` builtin, or the `-i` option to the `declare` builtin.

Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

`id++ id--`	variable post-increment and post-decrement
`++id --id`	variable pre-increment and pre-decrement
`- +`	unary minus and plus
`! ~`	logical and bitwise negation
`**`	exponentiation
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<< >>`	left and right bitwise shifts
`<= >= < >`	comparison
`== !=`	equality and inequality

`&` bitwise AND

`^` bitwise exclusive OR

`|` bitwise OR

`&&` logical AND

`||` logical OR

`expr ? expr : expr`
conditional operator

`= *= /= %= += -= <<= >>= &= ^= |=`
assignment

`expr1 , expr2`
comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the *integer* attribute using '`declare -i`' is assigned a value. A null value evaluates to 0. A shell variable need not have its *integer* attribute turned on to be used in an expression.

Constants with a leading 0 are interpreted as octal numbers. A leading '`0x`' or '`0X`' denotes hexadecimal. Otherwise, numbers take the form [*base*`#`]*n*, where the optional *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base*`#` is omitted, then base 10 is used. When specifying *n*, the digits greater than 9 are represented by the lowercase letters, the uppercase letters, '`@`', and '`_`', in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

Regular Expressions:

Metacharacter	Description
^	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
.	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, a.c matches "abc", etc., but [a.c] matches only "a", ".", or "c".
[ ]	A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", or "z", as does [a-cx-z]. The - character is treated as a literal character if it is the last or the first (after the ^, if present) character within the brackets: [abc-], [-abc]. Note that backslash escapes are not allowed. The] character can be included in a bracket expression if it is the first (after the ^) character: []abc].
[^]	Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than "a", "b", or "c". [^a-z] matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
$	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
( )	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, \n). A marked subexpression is also called a block or capturing group. **BRE mode requires** \(\).
\n	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is vaguely defined in the POSIX.2 standard. Some tools allow referencing more than nine capturing groups.
*	Matches the preceding element zero or more times. For example, ab*c matches "ac", "abc", "abbbc", etc. [xyz]* matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on. (ab)* matches "", "ab", "abab", "ababab", and so on.