Aliases

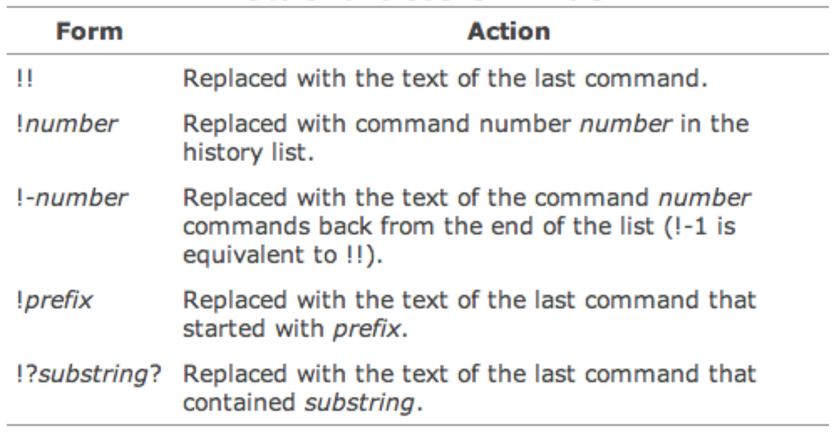
- Shell Command: ***alias*** *[-p] [word[=string]]*
  - If you alias a new command *word* equal to *string*, then when you type the command word the string will be used in its place
  - *alias* prints out all aliases defined
  - put your aliases in your .bashrc file (why?)
  - e.g. *alias dir=”ls -al”*
  - ***unalias***

History

- Shell Command: ***history*** *[-c] [n]*
  - Print out the shell's current command history.
  - If a numeric value *n* is specified, show only the last *n* entries in the history list.
  - If "*-c*" is used, clear the history list.

Command Re-execution

Figure 6-17. Command re-execution metacharacters in Bash.



Form	Action
!!	Replaced with the text of the last command.
!*number*	Replaced with command number *number* in the history list.
!-*number*	Replaced with the text of the command *number* commands back from the end of the list (!-1 is equivalent to !!).
!*prefix*	Replaced with the text of the last command that started with *prefix*.
!?*substring*?	Replaced with the text of the last command that contained *substring*.

History Substitution

- *^string1^string2^*
  - Substitute string2 for string1 in the previous command and executes it.

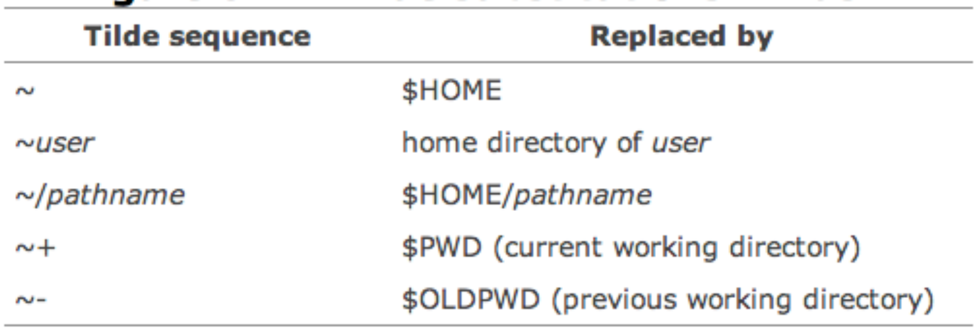
```
$ lp financial_report_july_2003.txt
lp: File not found.
$ ^2003^2004^
lp financial_report_july_2004.txt
request id is lwcs-37 (1 file)
$ _
```

Auto-Completion

- Bash can complete a filename, command name, username or shell variable name
- To have Bash attempt to complete the current argument of your command, type the TAB character.

Tilde Substitution

Figure 6-21. Tilde substitutions in Bash.



Tilde sequence	Replaced by
~	$HOME
~*user*	home directory of *user*
~/*pathname*	$HOME/*pathname*
~+	$PWD (current working directory)
~-	$OLDPWD (previous working directory)

Command substitution

- *$(command)*

```
$ echo there are $(who | wc -l) users on the system
there are 6 users on the system
$ _
```

Arithmetic

- + -   Addition, subtraction.
- ++ --   Increment, decrement.
- * / %   Multiplication, division, remainder.
- **   Exponentiation.
- Shell command: ***declare*** *-i name*
  - This form of declare defines the variable name as an integer value

Conditional Expressions

- Arithmetic conditional operators

<= >= < >	Less than or equal to, greater than or equal to, less than, greater than comparisons
== !=	Equal, not equal
!	Logical NOT
&&	Logical AND
||	Logical OR

Conditional Expressions

```
$ cat divisors.sh
#!/bin/bash
#
declare -i testval=20
declare -i count=2     # start at 2, 1 always works

while (( $count <= $testval )); do
  (( result = $testval % $count ))
  if (( $result == 0 )); then   # evenly divisible
    echo " $testval is evenly divisible by $count"
  fi
  (( count++ ))
done
```

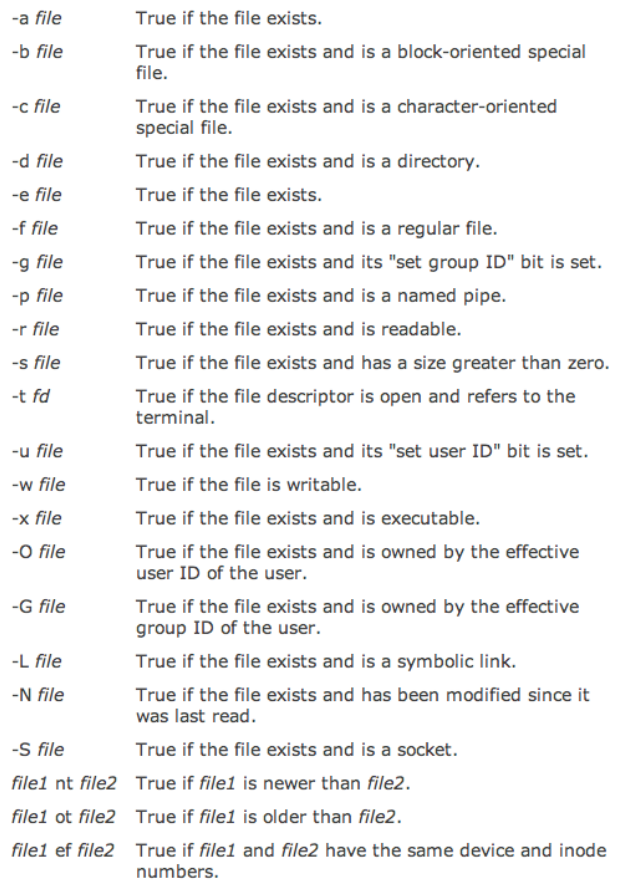
```
$ bash divisors.sh
 20 is evenly divisible by 2
 20 is evenly divisible by 4
 20 is evenly divisible by 5
 20 is evenly divisible by 10
 20 is evenly divisible by 20
$ _
```

String Comparisons

- String conditional operators.
  - -n *string* True if length of string is non-zero.
  - -z *string* True if length of string is zero.
  - *string1* == *string2* True if strings are equal.
  - *string1* != *string2* True if strings are not equal.

File-Oriented Expressions

- file-oriented conditional operators (see Figure 6-29)



-a *file*	True if the file exists.
-b *file*	True if the file exists and is a block-oriented special file.
-c *file*	True if the file exists and is a character-oriented special file.
-d *file*	True if the file exists and is a directory.
-e *file*	True if the file exists.
-f *file*	True if the file exists and is a regular file.
-g *file*	True if the file exists and its "set group ID" bit is set.
-p *file*	True if the file exists and is a named pipe.
-r *file*	True if the file exists and is readable.
-s *file*	True if the file exists and has a size greater than zero.
-t *fd*	True if the file descriptor is open and refers to the terminal.
-u *file*	True if the file exists and its "set user ID" bit is set.
-w *file*	True if the file is writable.
-x *file*	True if the file exists and is executable.
-O *file*	True if the file exists and is owned by the effective user ID of the user.
-G *file*	True if the file exists and is owned by the effective group ID of the user.
-L *file*	True if the file exists and is a symbolic link.
-N *file*	True if the file exists and has been modified since it was last read.
-S *file*	True if the file exists and is a socket.
file1 nt *file2*	True if *file1* is newer than *file2*.
file1 ot *file2*	True if *file1* is older than *file2*.
file1 ef *file2*	True if *file1* and *file2* have the same device and inode numbers.

File-Oriented Expressions

```
$ cat owner.sh
#!/bin/bash
#

if [ -O /etc/passwd ]; then
   echo "you are the owner of /etc/passwd."
else
   echo "you are NOT the owner of /etc/passwd."
fi



$ bash owner.sh
you are NOT the owner of /etc/passwd.
$_
```

Control Structures

- case .. in .. esac
- if .. then .. elif .. then .. else .. fi
- for .. do .. done
- while/until .. do .. done
- trap

case .. in .. esac

- Shell command: ***case***

  case *word* ***in***
  pattern { | pattern }) commands ;;*
  ...
  esac

  - Execute the commands specified by commands when the value of *word* matches the *pattern* specified by pattern.
  - The ")" indicates the end of the list of patterns to match. The ";;" is required to indicate the end of the commands to be executed.

if .. then .. elif .. then .. else .. fi

- ***if*** *test1;* ***then***
  commands1;
  *[****elif*** *test2;* ***then***
  commands2;]
  *[****else*** *commands3;]*
  fi

  - *test1* is a conditional expression, which, if true, causes the commands specified by *commands1* to be executed.
  - If *test1* tests false, then if an "*elif*" structure is present, the next test, *test2*, is evaluated ("else if"). If *test2* evaluates to true, then the commands in *commands2* are executed. The "*else*" construct is used when you always want to run commands after a test evaluated as false.

for .. do .. done

- Shell command: ***for***

  for *name* **in** *word { word }**
  do
    commands
  done

  - Perform *commands* for each *word* in list with *$name* containing the value of the current *word*.

while/until .. do .. done

- Shell command: ***while/until***

  while *test*
  do
    commands
  done

  until *test*
  do
    commands
  done

  - In a while statement, perform commands as long as the expression test evaluates to true.
  - In an until statement, perform commands as long as the expression test evaluates to false (i.e., until test is true).

trap

- Shell command: ***trap*** [[*command*] { *signal* } +]
  - The trap command instructs the shell to execute *command* whenever any of the numbered signals signal are received.
  - If several signals are received, they are trapped in numeric order.
  - If a signal value of 0 is specified, then *command* is executed when the shell terminates. If *command* is omitted, then the traps of the numbered signals are reset to their original values. If *command* is an empty string, then the numbered signals are ignored. If *trap* is executed with no arguments, a list of all the signals and their trap settings is displayed. For more information on signals and their default actions, see Chapter 12, "Systems Programming."

Functions

- Bash allows to define functions that can be invoked as shell commands

- ***function*** *name*
  {
  list of commands
  }
  or the keyword **function** may be omitted:
  name ()
  {
  list of commands.
  }

Functions

- parameters are accessible based on their positions

```
$ cat func2.sh                   ...list the script.
f ()
{
 echo parameter 1 = $1      # display first parameter.
 echo parameter list = $*   # display entire list.
}
# main program.
f 1                          # call with 1 parameter.
f cat dog goat               # call with 3 parameters.


$ sh func2.sh                   ...execute the script.
parameter 1 = 1
parameter list = 1
parameter 1 = cat
parameter list = cat dog goat
$ _
```

Functions

- ***return*** [*value*] return *value*
- ***export*** *-f functionname* -f option exports function
- ***local*** *name[=value]*
  - defines variable to be local to current function
- ***builtin*** *[command [args]]*
  - runs the named shell built-in command, and passes it args if present.

more ...

- ***select** name [**in** {word }+]*
  do
  list
  done

- Directory access and directory stack
  - pushd
  - popd
  - dirs

Job Control

- ***jobs*** [-*lrs*]   display all of the shell's jobs
- ***bg, fg, kill***

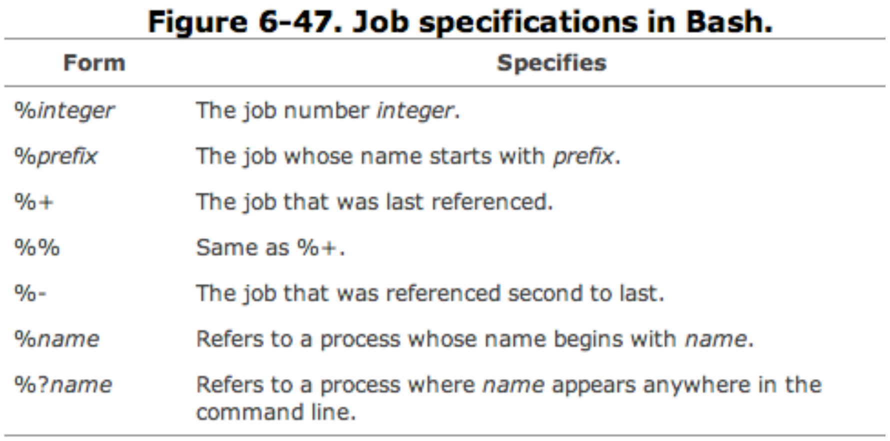


Figure 6-47. Job specifications in Bash.

Form	Specifies
%*integer*	The job number *integer*.
%*prefix*	The job whose name starts with *prefix*.
%+	The job that was last referenced.
%%	Same as %+.
%-	The job that was referenced second to last.
%*name*	Refers to a process whose name begins with *name*.
%?*name*	Refers to a process where *name* appears anywhere in the command line.