Unix Shell Programming

Shell Programming Basics

Roshan Chitrakar, PhD
roshanchi@gmail.com

What is a Shell?

- A shell is a ***program*** that acts as a ***user interface*** and ***script interpreter***, allowing you to enter commands and, indirectly, to ***access the services of the kernel***.
- First, a shell is a ***Command Processor***: a program that reads and interprets the commands that you enter.
- A shell also ***supports some type of programming language***. Using this language, you can write programs, call Shell Scripts, for the shell to interpret. These scripts can contain regular Unix commands, as well as special shell programming commands.
- Each type of ***shell has its own specific programming language and rules***. As a general rule, however, shells within the same “family” use similar programming languages.

The Bourne Shell Family: sh, ksh, bash

- The very first shell named **sh** was written in 1971 by one of the original creators of Unix, Ken Thompson.
- The original shell ***sh*** was replaced by a new shell (*given the same name; is backward compatible*) written by a team of Bell Lab programmers led by John Mashey in 1976.
- A brand new version of shell (also compatible with the original **sh** shell) was designed by another Bell Lab programmer Steve Bourne; was also named **sh**.
- Ironically, Mashey **sh** and Bourne **sh** were incompatible to each other. Years later, the Bourne **sh** was chosen as the standard shell and became default in the Unix Seventh Edition.

The Bourne Shell Family (contd.)

- In 1982, another Bell Lab scientist, David Korn, created a replacement for the Bourne shell, called the Korn Shell or **ksh**. The new Korn shell offered a history file, command editing, aliasing, and job control.
- With the next release of Unix, the Korn shell was distributed to the world at large, and it soon became a ***permanent replacement for the Bourne shell***.

The Bourne Shell Family (contd.)

- In the 1990s, a number of free, open source shells were created, the most important of which were the FreeBSD shell, Pdksh, the Zsh, and **Bash**. All of these shells complied with the IEEE 1003.2 standard (one of POSIX standards), making them adequate replacements for the Korn shell.
- ***pdksh*** is a modern clone of the Korn shell. ***pdksh*** was written to provide a Korn shell without restrictive licensing terms; hence the name "***public domain Korn shell***". The best way to think of ***pdksh*** is as a modern Korn shell that is free in both senses of the word (no cost + open source).

The Bourne Shell Family (contd.)

- Of all the members of the Bourne shell family, the most important is **bash**.
- ***bash*** was created by ***Brian Fox*** (1987) and later (starting in 1990) maintained by Chet Ramey, all under the auspices of the Free Software Foundation.
- Now, bash is supported by a ***community of programmers*** around the world.
- bash extends the capabilities of the basic Bourne shell.
- bash is not only a command processor ***with a powerful scripting language***; it also supports command line editing, command history, a directory stack, command completion, filename completion, and a lot more.
- bash is free software, distributed by the Free Software Foundation. ***It is the default shell for Linux***, as well as Unix-based Macintoshes, and is available for use with Microsoft Windows (running under a Unix-like system called Cygwin).

The C-Shell Family: csh, tcsh

- In 1978, ***Bill Joy***, a graduate student at U.C. Berkeley developed a brand new shell, which he based on the Unix Sixth Edition ***sh*** program, the predecessor of the Bourne shell.
- He added many important improvements, including aliases, command history, and job control. In addition, he completely revamped the programming facilities, changing the design of the scripting syntax so that ***it resembled the C programming language***. For this reason, he called his new shell the ***C-Shell***, and he changed the name of the program from ***sh*** to ***csh***.
- In the late 1970s, a programmer named Ken Greer from Carnegie-Mellon University, began work on a completely ***free version of the csh***, which he called ***tcsh***. it enhanced the C-Shell by offering a number of advanced features, such as filename completion and command line editing.

Which Shell Should You Use?

- If you like to ***go with the flow, stick with the default*** and use whatever you get on your system when you type sh. Most likely this will be ***bash with Linux***, the ***FreeBSD shell with FreeBSD***, and the ***Korn shell with a commercial Unix*** system.
- If ***you are adventurous***, there are many shells for you to try. For example, although the C-Shell is no longer a default shell, many people do enjoy using it, and ***you may find that tcsh or csh*** is already available on your system. If you want to go further afield, just search the Internet for Unix or Linux shells, and you'll find something new to try. (If you are adventurous and you can't make up your mind, try the ***Zsh***.)
- For ***day-to-day work***, ***you can use whichever shell*** takes your fancy, and you can change whenever you like. However, ***if you write shell scripts***, you should ***stick with the standard Bourne shell programming language*** to ensure that your scripts are portable enough to be used on other systems.

Which Shell Should You Use? (contd.)

- A complicated program has ***a lot of capability***, but also ***demands more time*** to master.
- Shells have a lot of features and options that you will never really need. These ***extra facilities are often a distraction***.
- One way ***to measure the complexity*** of a program is by looking at ***the length of the documentation***.
- FreeBSD shell, Bash, and the Zsh have most of the modern features
- Bash comes by default in most of the Linux distros.
- Zsh is complex and adventurous.

NAME OF SHELL	SIZE OF MAN PAGE	RELATIVE COMPLEXITY
Bourne Shell	38,000 bytes	1.0
FreeBSD Shell	57,000 bytes	1.5
C-Shell	64,000 bytes	1.7
Korn Shell	121,000 bytes	3.2
Tcsh	250,000 bytes	6.6
Bash	302,000 bytes	7.9
Zsh	789,000 bytes	20.8

Changing Your Shell

- If you use ***Linux***, your login shell will probably be ***Bash***.
- If you use a ***commercial Unix***, your login shell will probably be the ***Korn shell***.
- If you use ***FreeBSD***, it will probably be the ***Tcsh***.
- To show you the name of the current shell:

echo $SHELL

- If you want to try the Tcsh, just enter:

tcsh

- you can also change the shell by using ***chsh*** command

chsh [***-s*** shell] [userid]

Interactive and Non-interactive Shells

- The shell can act as both a ***user interface*** (interactive) and a ***script interpreter*** (non-interactive).
- When you see the ***shell prompt***, you ***enter a command***. The shell ***processes your command and then displays another prompt***. As you work in this way, the shell is your user interface, and we say that you are using an ***Interactive Shell***.
- Alternatively, you can create a set of commands, called a ***Shell Script***, which you ***save in a file***. When you run your script, the shell reads the commands from the file and processes them one at a time without your input. When this happens, we say that you are using a ***Non-Interactive Shell***.

Environment Variables And Shell Variables

- Environment variables are ***available to all processes***, they are global variables.
- Shell Variables are ***used only within a particular shell***, they are called local variables.
- In C-Shell family,
  - ***Global variables*** are given ***uppercase names*** and ***local variables*** are given ***lowercase names***. e,g, ***HARLEY*** is an environment variable and ***harley*** is a shell variable..
- In Bourne shell family (Bash, Korn shell),
  - ***both shell variables and environment variables are given uppercase names.***
  - every variable is ***either local only***, or ***both local and global***.
  - Within the Bourne shell family, you are ***only allowed to create local variables***.
  - If you want a variable to also be an environment variable, you must use a special command called export.

HAR***LEY=cool***

export HARLEY

Displaying Variables: env, printenv, set

- To display environment variables, use the ***env*** command:

env

- On many systems, there is another command you can use as well:

printenv

- To get them sorted alphabetically

env* | *sort* | *less

printenv* | *sort* | *less

- To display all the shell variables along with their values, you use the set command with no options or arguments:

set

Displaying/Using Shell Variables: ***echo***

- The job of the echo command is to display the value of anything you give it. For example, you enter:

echo I love Unix

echo $HOME $TERM $PATH $SHELL

- To display the value of TERM, you would enter:

echo ${TERM}

- or

echo $TERM

Using Variables in Bourne Shell family

HARLEY=cool

WEEDLY="a cool cat"

export HARLEY WEEDLY

PAGER=less; export PAGER

export PAGER=less

export PAGER=less EDITOR=vi PATH="/usr/local/bin:/usr/bin:/bin"

- If you need to ***delete*** variables, you use ***unset*** command

unset HARLEY WEEDLY

Using variables in C-Shell Family

- To ***set*** (create) and ***unset*** (delete) environment variables, you use ***setenv*** and ***unsetenv***.
- To set and unset shell variables, you use ***set*** and ***unset***.

setenv PATH /usr/local/bin:/usr/bin:/bin

setenv HARLEY cool

setenv WEEDLY "a cool cat"

setenv LITTLENIPPER

unsetenv HARLEY

set term=vt100

set path=(/usr/bin /bin /usr/ucb)

set ignoreeof

unset ignoreeof

The Login Shell

- A terminal is connected to a Unix system through a direct wire, modem, or network. For each physical terminal port on a system, a program called ***getty*** will be active.
- The Unix system—more precisely a program called ***init—automatically starts up a getty program on each terminal*** port whenever the system is allowing users to log in.
- ***getty determines the baud rate, displays the message login: at its assigned terminal.***
- As soon as someone types in some characters followed by Enter, ***the getty program disappears; but before it goes away, it starts up a program called login*** to finish the process of logging in.
- After ***login checks the password*** you typed in against the one stored in ***/etc/shadow***, it then ***checks for the name of a program to execute***. In most cases, this will be ***/usr/bin/sh, /usr/bin/ksh, or /bin/bash***.

The Shell's Responsibilities

- The shell ***analyzes each line you type*** in and ***initiates execution*** of the selected program.
- The shell also has other responsibilities : -
  - Program Execution
  - Variable and Filename Substitution.
  - I/O Redirection
  - Pipeline Hookup
  - Environment Control
  - Interpret Programming Language

Regular Expressions

- Regular expressions are used by several different Unix commands, including ***ed***, ***sed***, ***awk***, ***grep***, and, to a more limited extent, ***vi***. They provide a convenient and consistent way of specifying patterns to be matched.
- ***Filename substitution*** also involve certain regular expression like ***?***, ******* and ***[...]*** etc.
- The regular expressions recognized by the aforementioned programs are far ***more sophisticated than*** those recognized by the shell.

Matching Any Character: The Period (.)

- A period in a regular expression matches any single character, no matter what it is. So, the regular expression

r. specifies a pattern that matches an r followed by any single character.

.x. matches an x that is surrounded by any two characters.

- In the ***ed*** command

/ ... / searches forward in the file you are editing for the first line that contains any three characters surrounded by blanks.

$ ***ed intro***

248 implies no. of characters in the file intro

1,$p prints all the lines

/ ... / looks for three chars surrounded by blanks

/ repeats last search

1,$s/p.o/XXX/g changes all p.os to XXX

Matching the Beginning of the Line: The Caret (^)

- When the caret character ^ is used as the first character in a regular expression, it matches the beginning of the line. So the regular expression

^George matches the characters George only if they occur at the beginning of the line.

$ ***ed intro***

248

/^the/ finds the line that starts with the

1,$s/^/>>/ inserts >> at the beginning of each line

1,$s/^/ / inserts spaces at the beginning of each line

Matching the End of the Line: The Dollar Sign ($)

- The dollar sign $ is used to match the end of the line. So the regular expression

contents$ matches the characters contents only if they are the last characters on the line.

.$ matches

\.$ matches any line that ends in a period

^\. matches any line that starts with one dot

$ ***ed intro***

248

/\.$/ searches for a line that ends with a period

1,$s/$/>>/ adds >> to the end of each line

1,$s/..$// deletes the last two characters from each line

^$ matches any line that contains no characters

^ $ matches any line that consists of a single space character

Matching a Choice of Characters: The [...] Construct

$ ***ed intro***

248

/[tT]he/ looks for the or The

1,$s/[aeiouAEIOU]//g deletes all vowels

/[0-9]/ finds a line containing a digit

/^[A-Z]/ finds a line that starts with an uppercase letter

1,$s/[A-Z]/*/g changes all uppercase letters to *s

- If a caret (^) appears as the first character after the left bracket, the sense of the match is inverted.

1,$s/[^a-zA-Z]//g deletes all non-alphabetic characters

Matching Zero or More Characters: The Asterisk (*)

- The regular expression

X* matches zero, one, two, ... capital X's,

XX* matches one or more capital X's,

.* specifies zero or more occurrences of any characters

e.*e matches all the characters from the first e on a line to the last one

[A-Za-z][A-Za-z]* any alphabet followed by zero or any alphabet

[-0-9] {What does it match?}

- In ***ed***,

1,$s/ */ /g changes multiple blanks to single blanks

1,$s/e.*e/+++/ replaces ??? with +++

1,$s/[A-Za-z][A-Za-z]*/X/g changes ??? to X

1,$s/[A-Za-z0-9][A-Za-z0-9]*/X/g ???

Search in files with ***grep***

- To look for lines in cars containing four

$ ***grep four cars***

- The first of grep 's arguments is a regular expression (RE) and the second is a file name.
- grep displays more lines if the search pattern is found in more lines
- The search result is not restricted to whole words only; grep displays partial-words too.
- If we supply more than one file name, grep looks in all of the files and displays the file name before any lines containing the RE e.g.

$ ***grep needle ****

- finds in all files in the present directory.

Metacharacters in ***grep***

- RE usually consists of ordinary characters and special characters; the special characters are known as metacharacters viz. $ ^ [. *
- Command examples: -

$ grep 'it$' cars

$ grep '^The' cars

$ grep 't^s$i' cars *(not* metacharacters)

$ grep 'i.n' cars

$ grep 'a..e' cars

$ grep 't\.$' cars *(escaped metacharacters)*

$ grep ' [Ii]n ' cars

$ grep '[qxj]' cars

$ grep '[J-S]' cars

$ grep '[^A-Za-z0-9 .,:]' cars (negation)

$ grep '00*' cars

$ grep 'z' cars

$ grep 'z*' cars

$ grep 'stands.*the' cars

$ grep '^$' cars

$ grep '[A-Z][a-z][a-z]*' cars

grep options and more...

- The -v options makes grep display lines which do not match the RE.

$ ***grep -v 'e' cars***

- The -c option makes grep count the matching lines.

$ ***grep -c '^$' cars***

- you can use both options at once.

$ ***grep -cv 'e' cars***

- grep can be used along with other commands: -

$ ***who | grep roshanchi***

$ ***ls -l | grep '^d' | wc -l***

Line editor: ed

- A line editor is a program that allows us to change a file one line at a time.
- Starting ed : $ ***ed cars***
  - By typing a line number ed displays the line e.g. ***1*** prints the first line
- Substitute command : ***s/demote/waste/***
- Errors : ed has only one error message i.e. a question mark (?)
- Substitute suffix p : ***sXmaleXmanXp*** (X is a delimeter)
  - ed does not display the new version of the line. We can make that happen by adding a p to the end of the command.
- Substitute suffix g : ***s/an/AN/gp***
  - g means global. The basic form of the substitute command only makes one change to the line. If we want to change every occurrence in the current line, we have to add a g suffix.

ed (contd.)

- Regular expressions

  s/AN.*AN/an man wastes more than/p

  s/^/Fact: /p

  s/$/t/p

- Splitting a line

  s/n m/n

  m/p

  -

  u

  p

- The backslash (\ ) immediately before the end of the line tells ed that the new-line is part of the replacement string.
- The **-** command goes to the line above the current line, and the use **u** command undoes the effect of the previous change.

ed (contd.)

- Missing replacement string
  - If no replacement string is supplied, the characters matching the RE will be removed.

    s/...../p

    s/.$//p
- & in the replacement string
  - The ampersand means: whatever the RE matched in the most recent match

    s/man/(&)/p
- Accessing parts of a string
  - We can enclose parts of a complex regular expression in escaped parentheses (\( ) and (\)). Doing so, lets us split the complex regular expression into smaller parts and refer to the smaller parts individually.

    s/\(.*American\).*\(wastes.*\)/\1 person \2/p
  - Here, The first part was up to and including "American"
  - The last was from "wastes" to the end of the line
  - The middle part was all the characters between the first and last parts

ed (contd.)

- Searching

  /spends/ (searches for the text *spends*)
- The search starts at the current line; it continues towards the last line of the file.
- It is possible to search from the current line towards line one e.g.

  ?petrol?
- These ways of referring to lines (***/spends/ ?petrol?***) are known as context addresses; they can be used as if they were line numbers.
- Missing RE
  - If we do not give ed anything where it is expecting an RE, it will re-use the last RE we gave

    /[A-Z][a-z][a-z]*/

    //
  - The same thing applies to the substitute command too

    s//A/p

ed (contd.)

- Line numbers
  - If we put a line number in front of an ed command it will affect only the appropriate line of the file. We can also specify a range of lines by giving two line numbers separated by a comma.

    11p

    2,6p

    2,6d

    2s/car/automobile/p

    1,12s/ it / his car /gp

- More on Line ranges: -

  1,$ means all lines

  ,$ means from the current line to the end of the file

  1, means from line one to the current line

  .-2,.+2 means the five lines centred on the current line

  /parks/,/four/ means from the line containing parks to the line containing four

ed (contd.)

- Global command – g
  - format: g/RE/commands

    g/car/p

    g/automobile/s//car/g
  - The first g means:- all lines containing the RE; the second means:- make all possible changes on each line.

    g/his car/s//it/gp

    g/ it /d
- The inverse of g – v
  - The v command is the inverse of g in that the commands are performed on lines that do not match the RE.

    v/ it /d

Reference file 1: intro

The Unix operating system was pioneered by Ken Thompson and Dennis Ritchie at Bell Laboratories in the late 1960s.
One of the primary goals in the design of the Unix system was to create an environment that promoted efficient program development.

Reference file 2 : cars

The typical American male devotes more than 1,600 hours a year to his car. He sits in it while it goes and while it stands idling. He parks it and searches for it. He earns the money to put down on it and to meet the monthly instalments. He works to pay for petrol, tolls, insurance, taxes and tickets. He spends four of his sixteen waking hours on the road or gathering resources for it. The model American puts in 1,600 hours to get 7,500 miles: less than five miles per hour. In countries deprived of a transportation industry, people manage to do the same, walking wherever they want to go, and they allocate only three to eight percent of their society's time budget to traffic instead of 28 per cent.

Ivan Illich