UNIX, Shell Scripting and Perl
Introduction

Bart Zeydel
2003

Some useful commands

grep – searches files for a string. Useful for looking for errors in CAD tool output files.
Usage: grep error *
(looks for the string "error" in files in current directory)

find – look for a file.
Usage: find . –name updown.vhd

man – used for help on a command
Usage: man find
(displays the manual for the find command)

which – displays the path to a program
Usage: which dc_shell
(if dc_shell is in the PATH environment variable then the path will be displayed, otherwise the command not found or no <program name> found will be displayed. See later discussion on shells for environment explanation)

chmod – changes the permissions of a file. Look at man page for more detailed explanation.

> - redirect output to a file
Usage: cat foo > newfoo
(creates a new file called newfoor that contains the contents of foo)

>> - redirect output to a file, adding it to the end of an exisiting file
Usage: cat foo2 >> newfoo
(if used after the previous command this will add foo2 to the end of the file newfoo)

>! - redirect output and STDERR to file
Usage: hspice test.spi >! foo.out
(this is useful for running cad tools. You can redirect the output to a file, including errors, then when the run completes you can use the command grep to search for errors)

| - pipe the output to a shell function (such as more, less, grep, find, etc)
Usage: ls –l | more

xargs - used after | takes the result and uses it as an argument for another command

Usage: find . –name "*.spi" | xargs grep adder
(this searches all .spi files in the current directory and it's subdirectories, then performs a grep for updown on each of these files.)

pushd - available in tcsh. This allows directories to be pushed onto a stack.
Usage: pushd ../test
(this places your current directory on a stack then changes directories to ../test)

popd - available in tcsh. This allows directories to be popped off the stack
Usage: popd
(this will place you in the directory that was on the top of the stack)

Shell's

What is a shell?

"The shell is both a command language and a programming language that provides an interface to the UNIX operating system" - S. R. Bourne

Different Types of shell's

ksh – korn shell
sh – Bourne Shell
csh – BSD (Berkeley Software Distribution) C Shell
tcsh – enhanced version of Berkeley UNIX C Shell

So what shell am I using?

To find out at the command prompt type

Env | grep SHELL

What is an environment?

Each shell has an environment, which it uses to operate. For instance when you type ls on the command line to display the contents of the directory, the shell needs to know where to look for the ls command. This is what the PATH environment variable is used for.

Other things of interest with environments…

The environment also contains variables that are needed to run programs. For instance try:

env | grep SYNOPSYS

This will show some of the variables needed to run synopsys.

One or two final things about environments:
Type **env** at the command prompt to display all of the environment variables that are currently being used.

To change environment variables…

For tcsh you can edit a file called .cshrc in your home directory. This can also be done at the command line by typing **setenv <environment variable> <value>** however this is only a one time fix. To learn more about shells search on google for tcsh setup or .cshrc setup, there are a few hundred pages to hunt through.

Here are some manual pages:
csh:
http://unix.about.com/gi/dynamic/offsite.htm?site=http%3A%2F%2Fwww.neosoft.com%2Fneosoft%2Fman%2Fcsh.1.html
tcsh:
http://unix.about.com/gi/dynamic/offsite.htm?site=http%3A%2F%2Fwww.neosoft.com%2Fneosoft%2Fman%2Ftcsh.1.html

Shell Scripting:

What is a shell script?

Shell scripting is essentially a file filled with unix shell commands.

How do I run one?

There are two ways to run a shell script

1) Create a file called scriptfile
    a. On the first line type #!/usr/bin/csh (this needs to be the location of csh)
    b. Exit and change the permissions on the file to be an executable
        i.e. chmod 755 scriptfile
    c. At the command prompt type ./scriptfile
2) Create a file called scriptfile
    a. At the command prompt type source scriptfile

Why use a shell script?

Makes doing repetitious tasks less painful.

Basics of Shell-scripting:

Handling command line arguments.

Arguments can be passed to a shell script.

i.e. ./scriptfile foo1 foo2

To handle these you can refer to them as $1 and $2 respectively in scriptfile.

So if you wanted the script to run hspice on both files you would write the script file as follows:

```
#!/usr/bin/csh
hspice $1
hspice $2
```

Shell scripting has many more capabilities (such as if statements). To find more about how to do this either look through some of the references at the end of this document or search online for cshell script examples.

Here area few places on the web that might be of use.

Probably has all that you could ever want to know about UNIX
http://heather.cs.ucdavis.edu/~matloff/unix.html
Other usefull sites:
Intro to UNIX
http://www.ee.surrey.ac.uk/Teaching/Unix/
A reference site
http://bromide.ocean.washington.edu/unix.tutorial.html
Info on Makefiles
http://www.student.math.uwaterloo.ca/~cs-marks/resources/unix/make/tutorial/

Into to Perl

What is Perl?

Perl is an interpreted programming language that supports many of the features of sed, awk, grep, sh, csh, C and C++. By interpreted programming language it means that it doesn't have to be compiled, allowing it to be platform independent code (although it is often environment dependent).

How to get started with perl.

First off check to see that the machine you are using has perl (i.e. type which perl). Write down this location as you will need it for making the perl script an executable. There are two ways of running perl (just like shell scripts)

1) Create the file perlfile
    a. At the command line type: perl perlfile
2) Create the file perlfile
    a. On the first line put #!/usr/local/bin/perl (or wherever your perl is installed)
    b. Change the permissions on the file to be an executable.
        chmod 755 perlfile
        ./perlfile

Argument Passing

In perl command line arguments can be passed.

Ex. ./perlfile updown.vhd output.vhd

These arguments are stored in an array ARGV. In perl an array is reffered to by the @ symbol, while a scalar variable is referred to by a $.

$ARGV[0] and $ARGV[1] store the respective input arguments.

The size of an array is given by $#
So to determine the number of arguments passed would be

$#ARGV

Basic Variable Assignment

To assign a variable in perl use =

(NOTE: each line except loop headers is followed by a ; in perl)

```
$input1 = $ARGV[0];
$input2 = $ARGV[1];
```

To place input1 and input2 into an array.

```
$inputs[0] = $input1;
$inputs[1] = $input2;
```

NOTE: Variables in perl essentially don't have a type, i.e. floating, integer, etc.

DISPLAY I/O

Printing in perl is as follows

```
print <STDOUT> “Input one is $input1 \n”;
```

(STDOUT is optional, but should be used to avoid problems and confusion)

Reading from the keyboard.

```
print <STDOUT> “Enter your name :”;
$name = <STDIN>;
```

This creates a unique problem to perl. The carriage return will be included in the variable name. to eliminate this use the command chop.

i.e. chop ($name);

Loops and conditionals

for loop example

```
for ($lev = 0; $lev < 10; $lev ++)
{
        print <STDOUT> “$lev \n”;
}
```

foreach – used for an array. Assumes @lines is an array of lines of a file. What Foreach does is loop through all of the elements of @lines, placing the current element in the variable $line.

```
foreach $line(@lines)
{
        print <STDOUT> “$line \n”;
}
```

while loop example

```
while ($line != $#lines)
{
        print <STDOUT> “$line\n”;
}
```

if example

```
if ($line == $#lines)
{
        print <STDOUT> “End of File \n”;
}
```

File Handling

To open a file in Perl, first assign it a file handle, then read it into an array, then close it. FILE1 is the filehandle for $filename in the following example

```
$filename = $ARGV[0];
open(FILE1, "$filename");
@lines = <FILE1>;
close (FILE1);
```

To go through each line of the file

```
foreach $line(@lines) {
        print <STDOUT> "$line\n";
}
```

To write to an output file

```
$filename2 = $ARGV[1];
open (OUTFILE, ">$filename2");
print OUTFILE "This is a test \n";
close (OUTFILE);
```

It is also possible to append to the end of a file by using >> instead of >

NOTE: For the command open an actual filename could be used instead of a variable containing the filename.

Pattern Matching / Regular Expressions

This it the most powerful feature of Perl. It allows for searching and replacing. There are many features and capabilities that Perl has for this, so I will only try to provide some useful examples to get you started. All of the examples will assume that $line contains a string (or line) from a file.

To search for subckt on a line and print if found.

```
if ($line =~ /subckt/)   {
        print <STDOUT> "found \n";
}
```

(What is being searched for is between / and /)

To search for subckt at the beginning of a line

```
if ($line =~ /^s*subckt/)
        …
```

^ - means beginning of a line

s* - means 0 spaces after

To search for inv on a line with spaces before and after it.

```
if ($line =~ /\s+inv\s+/)
        …
```

\s+ - means all blank space characters (i.e. space and tab)

So this is searching for blank spaces followed by inv followed by blank spaces.

How to assign variables values found in a search

Assume $line contains

The quick brown fox .stops

Where “The” occurs at the beginning of the line.

Here is the code to assign each word to a variable using / /

```
$line =~ /\^s(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+\.(\S+);
$word[0] = $1;
$word[1] = $2;
$word[2] = $3;
$word[3] = $4;
$word[4] = $5;
```

So how does this work?

\S+ matches all non white space characters.

Using ( )’s stores that group of characters into a variable (starting with $1)

Notice before the last (\S+) there is a \. , what this does is store only stops in $5 instead of .stops

There are many characters that have to be escaped (i.e. \) if you are having trouble with pattern matching then try using a \ on the character causing problems.

Search and Replace

Format:

$line =~ s/<what you are searching for>/<what you are replacing it with>/;

Works the same as pattern matching.

So for example if you wanted to replace all of the instances of updown with downup in a line the code would be as follows:

$line =~ s/updown/downup/g

the “g” at the end means replace multiple times per line if possible, otherwise only one instance per line will be replaced.