C-Shell Tutorial

Section 1

Goals:

- Learn how to write a simple shell script and how to run it.
- Learn how to use local and global variables.

About CSH

The Barkley Unix C-shell was originally written with the intent to create a shell with a syntax which resembles the c programming language.

In this section we will describe all of the steps involved in writing and running a simple shell script called "helloworld". If you are already familiar with the c-shell basics you may skip this part and do exercise 1 right away.

Step 1:

Create a file called helloworld in a text editor (xemacs for example).

Step 2:

The first line of a shell script always indicates where the shell interpreter is located. Since we are using csh, the first line should look as follows:

```
#!/bin/csh
```

The characters "#!" are referred to as sha-bang. It is in reality a number which tells the system that this particular file contains a set of commands to be given to a command interpreter.

Within a shell script the character '#" is used to indicate a comment line. So lines that are preceded by '#" are disregarded by the command interpreter.

Variables are declared and initialized in the following way:

```
set var = "test"
```

Any subsequent reference to the variable needs to be preceded by a "$" character. For instance if we wanted to use the echo command to print this particular variable to the screen we would do it as follows:

```
echo $var
```

Our helloworld script will now look like this:

```
#!/bin/csh

#hello world script
set text="Hello World"
echo $text
```

Step 3:
Save the file.

Step 4:
Make the file executable. To do that use the chmod command:

```
chmod +x helloworld
```

Step 5:
Run the script.
To run the script there are two options:

1. You can type ./helloworld in the shell. Since "." denotes the current working directory "./helloworld" gives the system an absolute path to the shell script.
2. If you want to be able to use helloworld just like any other shell command, i.e. by typing "helloword", you need to change a global variable called $PATH.

PATH is an environment variable that contains a list of paths separated by a colon. These paths denote directories where executables are located. Whenever a command name is given to the command interpreter it looks through all of the paths until it finds the directory which contains the corresponding executable.
If the helloworld script is located in the current directory, we could add the current

directory to the path as follows:

```
setenv PATH .:$PATH
```

Note: setenv is used for setting environment variables.
Now you will be able to execute helloworld by just typing:
helloword

Exercise 1

Write a shell script called **getinfo** that will print the following on the screen:

Operating System:
type of operating system on this machine
Machine Name:
name of this machine
User Name:
name of user that is logged in
User Details:
Login name, shell used, directory name, date when mail was read the last time

Sample Output:

```
Operating System:
Linux
Machine Name:
mango
User Name:
aagovic
User Details:
Login: aagovic                    Name: Amrudin Agovic
Directory: /home/grad02/aagovic        Shell: /usr/local/bin/tcsh
Mail last read Sun Jan 16 17:50 2005 (CST)
```

Hints:
The user name is stored in a global variable called $USER. For the remaining information consider using these commands:
uname – operating system
hostname - name of the machine
finger $USER – finger command prints user details

Section 2

Goals:

- Learn how to write loops.
- Learn how to use arrays, in particular how to use the argument array
- Learn about the shift command
- Learn about conditional statements

Arrays:

In addition to regular variables the c shell also allows the use of arrays. One special array called argv is available in every script that you write. It holds the command line arguments.

In general arrays in c-shell are indexed starting at one. In the previous example if we call

```
helloworld greetings form cold Minnesota
```

the word "greetings" will be the value of $argv[1].

To get the length of an array (or list) in c-shell one can prepend $# to the name of the array. For instance :

```
echo $#argv
```

will print the number of arguments that were passed. Note, this can be applied to any array.

Suppose that $ar is a given array. We can add an element to an array in the following way:

```
set ar=($ar "test")
```

Loops:

There are two convenient ways to loop through an array. These two examples simply print the arguments to the screen.

```
foreach arg  ($argv)
  echo $arg
end

while ($#argv  >  0)
  echo $argv[1]
  shift argv
end
```

Note: the shift command shifts an array. If the value of argv was "greetings from cold Minnesota", after shifting the value will be "from cold Minnesota", "from" will become the value of $argv[1]

There is also a third way to create a loop, but it is somewhat less convenient. To describe the third way one needs to know how to perform arithmetic operations. To assign a calculated value in the c-shell the "@" command is used.

For instance:

```
set x=1
set y=2;

@ y=$y+$x
or
@ y++
```

This allows us to write a loop in the following way:

```
set k=1;
#note the spacing in the conditional statement is important the c-shell interpreter will
#give errors if the spacing is not there. The parser that it uses is not very robust
while ($k <= $#argv)
   echo $argv[$k]
   @ k++
end
```

For looping through simple arrays I would recommend one of the previous two options.

If statement:

The syntax for the if statement looks as follows:

```
if (expression) then
  ...
else
  ...
endif
```

There are several useful tests that the shell allows. Here are some of them:
-d $name
  true if the variable name denotes a directory
-f $name
  true if the variable name denotes a file
-e $name
  true if the file/directory exists
-r $name
  true if the file is readable
-w $name
  true if the file is writable
$?name
  true if the variable name exists

Note: these are just some examples, there are many more tests available in csh. You can read about all of the tests in the man pages (the next section of this tutorial will cover the man pages)

Example:

```
if ((-e “test.txt”) && (-f “t est.txt”)) then
  echo “text.txt exists as a file”
endif
```

Exercise 2

Write a shell script called **getreadable** whose syntax looks as follows

getreadable file1 file2 file3 ...

Your script should go through all of the files that were passed to it. It should print those files that were readable to the screen. If the user specifies a name which does not represent a file it should let the user know. If no arguments are passed to the script it should print an error message.

Section 3

Goals:

- Learn how to use input/output redirection
- Learn how to use pipes

Redirection:

The output of a program can redirected into a file by using redirection. Likewise one can use a file as input into a program that reads from standard input stream.

For instance:

```
ls > output.txt
```

will create a file called output.txt which contains the output of the ls command.

You can also append the output of ls to an existing file in the following way:

```
ls >> output.txt
```

Pipes:

Pipes can be used to redirect the output of one program as input into another.
For example the command wc -l can be used to count lines.

If we combine:

```
ls | wc -w
```

we will get the line count in the ls output. A pipe is denoted by the "|" character. Note, several commands can be chained in this manner.

Exercise 5

Write a shell script called **mycount**. Given a file and a word it should count how frequently that word appears in the given file.
Syntax:

mycount word filename

The script should print the number to the screen.
Hint: look at the grep command in the man pages.

Section 4

Goals:

- Learn how to handle different types of quoting.
- Learn how to capture the output of a command into an array
- Learn how to use the man pages.

Quoting is very important in the c-shell. This is where a lot of people make mistakes. There are three types of quotes in c-shell.

Double Quotes:

```
set var="*.txt"
```

Certain wild characters have a special meaning within the shell. If we type an expression such as *.txt the command interpreter will attempt to expand it by matching all files that end in txt.

For example if we call

```
ls *.png
```

The interpreter first expands the *.png expression into all png files in the current directory. These files are then passed as arguments to the ls command. When double quotes are used the shell does not interpret whatever is within the quotes. However it does perform variable substitution.

For example:

```
echo "$var"
```

will print

```
*.txt
```

on the screen. The shell replaces the variable name $var with its value. However, it does

not attempt to interpret it any further.

In contrast if we call

```
echo $var
```

A list of all of all txt files in the current directory is printed. Double quotes are used if we want to prevent the contents of a variable to be interpreted by the shell. Suppose that a script takes a pattern such as *.png as an argument. When dealing with this particular argv entry it is advisable to use double quotes to prevent unwanted interpretation by the shell.

Single Quotes:

When single quotes are used the quoted text is taken literally. Not even variable substitution takes place.

For example if we call:

```
echo '  $var  '
```

$var will be printed on the screen. Single quotes can be used whenever we want to prevent unwanted interpretation by the command interpreter. Since no variable substitution takes place, single quotes are usually not used in conjunction with variables.

Back Quotes:

When back quotes are used, a given command within the back quotes is executed and the output of the command is returned.

For instance if we use:

```
set ar=`ls`
```

ar will be an array containing all of the files within the current directory. This is very useful for capturing the output of a command.

The Escape Character:

To prevent interpretation of special characters one can also use the escape character "\". Suppose that we wanted to print * to the screen. We could do it like this:

```
echo \*
```

Man Pages

The man pages will probably be your best friend this semester, at least as far as this class is concerned. The man pages are the "manual pages" for Linux. In the man pages documentation is provided for almost all Linux commands and for a number of c functions.

To find out how to use a given command you can type

man command_name

for example:

```
man kill
```

The man pages are divided into sections. If you are interested in a c function rather than a shell command you can get the documentation as follows:

```
man -S 2 kill
```

Note: On Linux an upper case letter "-S" is required when specifying a section. On Solaris you have to use a lower case letter "-s".

Exercise 3:

Write a script called **printstuff** that prints the following:
$><^&*
"*"
'$*'!'
to the screen. Try this without quoting and then think about what you would use here to make this work.

Exercise 4:

Write a shell script called **myrm**. The syntax should be as follows:

myrm dir pattern

parameters:

pattern – specifies the type of files
directory_name – specifies the directory of interest

The script should delete all files specified by the pattern from the given directory.

For example:
myrm testdir '*.txt'
removes all .txt files from the testdir directory.

Using the man pages to find out how the "find" command can be used to find all files of a given type in a certain directory.
Note: the find command works recursively, so it will take care of subdirectories. It will return the paths to the files of interest. Once you have the paths you can use the "rm" command to delete them one by one. The rm command is a dangerous command, so be careful not to delete anything important. You might want to consider using the "-i" option of the rm command. With that option the user is asked before deleting a file.
You should only attempt to delete files that are readable and writable. If a file is not writable or readable you should let the user know.
If an incorrect number of arguments are passed to the script it should print an error message.

Additional Exercises:

Exercise 6

Write a shell script called **sayhi** It should take a list of email addresses and send the following email to each address:
Hi! How are you doing?

Syntax:
sayhi emailaddr1 emailaddr2 ...

For example

sayhi “aagovic@cs.umn.edu”

Will send the above message to the specified email address.

Once you have written this script you can test it by sending the email to yourself.

Hints: read the man pages for the command “sendmail”.

Exercise 7

Write a script called **getids**. Given a command name of a program in execution it should return the corresponding process id(s).

For Example:
getids xemacs
should return all of the process ids associated with the currently running xemacs process(es).
The output might look something like this:

12238
12241
12242

Hint: Use the ps command. Consider using either the awk or the cut command. Take a look at the man pages to find out how they work.

Further C-Shell References:

- http://www.eng.hawaii.edu/Tutor/csh.html
- http://archive.ncsa.uiuc.edu/General/Training/InterUnix/scripting/