Introduction to C Shell Programming

Overview: Shell programming is very similar in concept in many operating systems. If you know how to write "batch" files in MS-DOS, then you know the basic ideas behind shell programming in UNIX. However, the syntax is altogether different. This tutorial focuses solely on the Cshell, not the Bourne shell.

Section — Topic

The need for shell programming

Do you ever find that you often do a number of UNIX commands together and you would like to "bundle" them up into one name? You can do this, in effect, creating a brand new command. Other operating systems permit this convenience, most notably MS-DOS, which calls such files "BAT" or batch files. In UNIX, such a file is called a shell script.

First, make sure you know about the various UNIX shells (Bourne and C-shell). There is information in the glossary menu.

Both the Bourne shell and the C shell permit you to create and use shell scripts, but because the syntax of the commands that these two shells use is slightly different, your shell script must match the shell that is interpreting it, or you will get errors.

A shell script is just a readable file that you create and into which you put shell commands. The first line determines which shell program will interpret or execute a shell script.

* If the first line begins with a C-shell comment (starting with # in position 1) then this will be interpreted by the C-shell and must use C-shell syntax.

* Otherwise, the file will be considered a Bourne shell script.

You can have comments in either type of shell script, although the syntax differs. Bourne shell comments begin with a colon, whereas C-shell comments commence with the pound sign (#).

For the rest of this tutorial, we will concentrate on the C-shell.

How to create simple scripts

Most shell scripts that you write will be very simple. They will consist of a number of UNIX commands that you would have typed at the prompt, possibly substituting a different file name. These substitutions are called positional parameters.

To create a shell script that has no parameters and does the same thing every time it is called, just put the commands in a file. Change the permissions on the file so that it is executable and then use it. The name of the file should be something that you can easily remember and which makes sense given the operation that you are performing.

Let's make one that clears the screen, prints out the date, time, hostname, current working directory, current username, and how many people are logged on. The name of the script will be "status". So edit a file called "status" and put the following lines into it: (Don't type the "frame" of dashes and vertical bars -- these are meant to show you what the file looks like.)

```
+-----------------------------------------------------------
| #
| clear
| echo -n "It is currently: ";date
| echo -n "I am logged on as ";who am i
| echo -n "This computer is called ";hostname
| echo -n "I am currently in the directory ";pwd
| echo -n "The number of people currently logged on is:"
| who | wc -l
+-----------------------------------------------------------
```

How to make a file executable and put it in your path

Make sure that you put the # in line 1. Now set the permissions:

```
% chmod 0700 status
```

This makes it executable and readable, both of which are necessary.
To use, just type

```
% status
```

If you see a cryptic command saying "command not found", it is probably because your path does not include the current directory. To remedy this, put the dot and a slash in front of the name:

```
% ./status
```

or you can modify your path:

```
% set path=($path .)
```

Note the space in front of the period.

Let's explain just a few things in the shell script above. Note that echo -n is used a lot. The echo command just prints lines to the screen, and normally it puts a newline after the thing it prints. -n inhibits this, so that the output looks better.

You can string together more than one command on a line by using a semicolon. Thus, clear;date;whoami;pwd could be put all on one line and all four of the commands would be executed, one after the other. This is similar to the vertical bar (the pipe), although it is simpler.

Parameters

Now let's get more complicated by adding positional parameters. Parameters

are given after the name of the file when you start the shell script. Each parameter has a position, first, second, etc. When the shell interpreter reads and executes each line of the shell script file, it looks for symbols like $1, $2, etc and it substitutes for these symbols the positional parameters.

Let's do a very simple example. Our shell script will attempt to find the word "unix" (irrespective of case) in a file that we give as a positional parameter:

```
+-----------------------------------------------------------
| #
| grep -i unix $1
+-----------------------------------------------------------
```

The -i option says ignore case. Since we are always looking for the word unix (or UNIX, or Unix, etc.), all we need to vary is the file name. Suppose that we called this file "funix" for "find unix", and we made it executable using chmod. Now to use it on a file, we would type

```
% funix myjunk
```

and it would search file "myjunk" for the word unix (or Unix, or UNIX, etc.), printing out each line that it found.

You can have any number of parameters. The second is $2, the third is $3, etc.

Another common variation is to refer to all the parameters at once by using $*. Our little shell script only looks at one file at a time. If we typed

```
funix myjunk yourjunk theirjunk ourjunk
```

it would only search the first file "myjunk". To make it search all, we could do

```
+-----------------------------------------------------------
| #
| foreach i ($*)
|     grep -i unix $i
| end
+-----------------------------------------------------------
```

"foreach" is one of the many control structures of C-shell scripts. It takes a list of items ($*) and assigns each one to the shell variable i in turn. Then this shell variable is referenced (i.e., used) in the grep command by saying $i. All shell variables must have a $ in front when they are used. The end keyword says that this is the end of the foreach construct, not the end of the shell script.

In many situations, UNIX commands themselves are set up to accept multiple filenames, and grep is one of these. So you could have done

```
+-----------------------------------------------------------
| #
| grep -i unix $*
+-----------------------------------------------------------
```

instead. But not all cases work this easily. You just have to know your UNIX commands.

Let us review the syntax of parameters. Each parameter is identified by $1, $2, $3 and so on. The name of the command is $0. A short hand for all the parameters is $*. To find out how many parameters there are, $#argv is used.

Here's an example of the beginning of a shell script which checkes to see if the user entered enough parameters, because some scripts require a certain number. For example, grep needs at least one parameter, which is the string to search for.

```
+-----------------------------------------------------------
| #
| if ($#argv < 2) then
|     echo "Sorry, but you entered too few parameters"
|     echo "usage:  slop file searchstring
|     exit
| endif
+-----------------------------------------------------------
```

This example gives you a flavor of the syntax of the if statement, use of the echo command to act as output from a shell script, and the exit command which terminates the shell script immediately.

The general syntax of if and if-then-else is:

```
if ( expression ) then
     statements
endif
```

```
if ( expression ) then
     true statements
else
     false statements
endif
```

We will discuss what expressions can go inside the parentheses next.

Expressions

The C shell language was meant to be reminiscent of the C language, and it is to some extent. But there are differences. For example, in the above patterns for if statements, the two keywords "then" and "endif" do not appear in C. The curly braces of C are not used in the C shell for the same thing, but for something completely different, which may be quite confusing. So it is wrong to imagine that knowledge of C confers on you the ability to write C shell scripts!

We start off with something that is used a lot in if statements, and is not in C: file queries. These are expressions that are used to determine characteristics of files so that appropriate action may be taken. For example, suppose that you want to see if a certain file exists:

```
if (-e somefile) then
    grep $1 somefile
else
    echo "Grievous error!  Database file does not exist".
endif
```

The name of the file does not have to be "hard coded" into the if statement, but may be a parameter:

```
if (-e $2) then
```

Here is a full list of the Cshell file queries.

```
-e file          file merely exists (may be protected from user)
-r file          file exists and is readable by user
-w file          file is writable by user
-x file          file is executable by user
-o file          file is owned by user
-z file          file has size 0
-f file          file is an ordinary file
-d file          file is a directory
```

All of queries except -e automatically test for file existence. That is, if the file does not exist, then it cannot be writable. But -r will fail for one of two reasons: 1.) the file exists but is not readable by the owner of the process that is running this script, or 2.) the file does not exist at all.

There are several boolean operators that are applied to C shell expressions, including the file queries above. They are:

```
     !   -- negate
     &&  -- logical and
     ||  -- logical or
```

For example the way to test to see if a file does not exist would be:

```
     if (! -e somefile) then
          # does not exist
```

Make sure to put spaces before and after the -e because failure to do os will confuse the C shell. Here's a way to combine two queries:

```
     if (-f somefile && -w somefile) then
          # the file exists, is not a directory and I can write it
```

If there is a doubt as to precedence, use parentheses, but you may need to use spaces before and after the parentheses. The C shell's parser is not as robust as the C compiler's, so it can get confused easily when things are run together without intervening spaces.

Variables

The C shell scripting language permits variables. The variables can have long, mnemonic names and can be either character or numeric, although floating point variables are not allowed. You can also create arrays, which will be discussed in a later section.

When you refer to a variable's value in a C shell statement, you must prefix the variable name with a dollar sign. The only time you don't use a dollar sign is in the "set" statement which assigns a value to a variable, or changes the value of an existing variable. The format of set is

```
     set name = expression
```

C shell variables are dynamic. They are not declared but come into existence when they are first set. Consequently, you delete them in a shell by using "unset".

```
     unset name
```

There is a special value, called the NULL value, and it is assigned to a variable by doing

```
     set name =
```

with no expression. Notice that such a variable is still defined, i.e. it still exists, even though it has this special NULL value. To actually get rid of the variable, use unset.

To give a character value to a variable, you can use double quotes or you can forego them. If the character string contains special characters, such as a blank, then you must use double quotes. Here are some examples:

```
    set name = Mark
    echo $name
    if ($name == Mark) then
      ...

    set name = "Mark Meyer"
    echo $name
    set dirname = /usr/local/doc/HELP
    ls $dirname
```

You can find out if a variable is defined or not by using the special form $?var. This could be used in an if statement. For example:

```
    if ($?dirname) then
```

```
        ls $dirname
    else
        ...
```

To change a variable's value, just use set again, but do not use $.

```
    set dirname = /mnt1/dept/glorp
```

To add on to an existing character string variable, you can do something like the following:

```
    set sentence = "Hi"
    set sentence = "$sentence there everybody"
```

Now $sentence, if echoed, would have "Hi there everybody" in it. The following also works:

```
    set sentence = Hi
    set sentence = "$sentence there everybody"
```

There is a special variable called $$ which has the process id number of the process that is running this shell script. Many programmers use this to create unique file names, often in the /tmp directory. Here's an example of copying the first parameter (which is obviously a filename) into a temp file whose name uses the pid number:

```
    cp $1 /tmp/tempfile.$$
```

This will create a file whose name is something like /tmp/tempfile.14506, if the pid number is 14506.

Actually, the computer cycles through the pid numbers eventually, but usually the same pid does not occur for several days, so there is seldom any need to worry.

Use of variables in the shell

One of the nice features about Cshell programming is that there is no clear line between what you can do in a shell script and what you can type in from the prompt. Thus, you can set and unset variables, use for loops and do all sorts of things at the command prompt. Some things will not work, like using the parameters $1, $2, etc because there are none. But other features work, and the use of setting variables is quite handy, especially when you want to use a long, complex pathname repeatedly:

```
    % set X = /usr/local/doc/HELP
    % ls $X
    % ls $X/TUTORIALS
```

You can even embed the shell variables inside other strings, as shown above in $X/TUTORIALS. Obviously, you cannot follow the shell variable with a string that begins with an alphabetic or numeric character because the C shell will not know which variable you are talking about, such as $XHITHERE.

Arithmetic variables

Variables whose values are integer use a slightly different set command. The commercial-at sign is used instead of "set" to indicate "assignment". Otherwise, the Cshell would use a character string "123" instead of the integer 123. Here are a couple of examples:

```
    @ i = 2
    @ k = ($x - 2) * 4
    @ k = $k + 1
```

There is also a -- and a ++ operator to decrement and increment numeric variables. Be careful not to omit the blank that follows the at-sign!

```
@ i--
@ i++
```

Expressions and true and false

The Cshell works very much like C in that it treats 0 as false and anything else as true. Consequently, the expressions that are used in if and while statements use the numeric values. Here's a counting loop that uses a numeric variable:

```
@ n = 5
while ($n)
     # do something
     @ n--
end
```

There are also numeric constants, such as 0, 1, etc. An infinite loop is often seen in Cshell scripts as

```
while (1)
    ...
end
```

To get out of such a loop, use break or exit. Of course, exit also causes the entire shell script to end! In the following while statement, the user is asked to type in something. If 0 is entered, then the while loop ends. Note the use of $< as the input device in the Cshell language, and an abbreviated if statement that foregoes the use of then and endif:

```
while (1)
   echo -n "Gimme something: "
   set x = $<
   if (! $x) break
end
```

If a variable contains the NULL value, then its use in an expression will be the same as if it were 0.

Boolean conditions

To wrap up the discussion of operators and conditions, here are the Boolean comparison operators. Note that some of them are used only for strings while some are used for only numbers. A string does not necessarily have to be surrounded by double quotes (unless it contains special characters like spaces or other things.)

Expressions and operators

```
==        equal     (either strings or numbers)
!=        not equal     (either strings or numbers)
=~        string match
!~        string mismatch
<=        numerical less than or equal to
>=        numerical greater than or equal to
>         numerical greater than
<         numerical less than
```

Here's a simple script to illustrate:

```
#
set x = mark
set y = $<
```

```
echo $x, $y
if ($x == $y) then
   echo "They are the same"
endif
```

If you type in "mark" without the double quotes, it will say they are the same. Strangely enough, if you omit the double quotes when you type in mark, the Cshell no longer thinks the variables are equal! Apparently, the double quotes are stored as part of the string when you enter the value by means of $<.

Strings have to match exactly, and 0005 and 5 are two completely different strings. However, they are the same numerical value. The following shell script would say that 0005 and 5 are the same:

```
#
@ x = 5
@ y = $<
echo $x, $y
if ($x == $y) then
   echo "They are the same"
endif
```

But if you were to replace the @'s with set's, they would no longer be the same.

Input and output

Output is fairly simple. You can use echo to show literals and variable values. If you do not want to cause a newline to be printed, use -n. This is especially valuable in prompts, as in the while loop in the last section.

```
echo "Hi there world"
echo -n "Please type in your name: "
echo "The current directory is " $cwd
```

$cwd is the current working directory, and is a built-in variable (discussed next).

To get something from the user, use $<. This causes the shell to pause until the user types a carriage return. What the user typed before the RETURN is the value that $< returns. This can be used in many different settings: in if conditions, while loops, or in set statements.

```
set x = $<
```

Of course, if you expect to get something intelligent from the user, make sure to prompt her for the type of information you are requesting!

Built-in variables

There are a few built-in variables, like $cwd and $HOME. $Cwd is the current working directory, what you see when you use "pwd". $HOME is your home directory. Here are others:

```
$user      -- who am I?
$hostess   -- name of the computer I am logged on to
$path      -- my execution path (list of directories to be searched
              for executables)
$term      -- what kind of terminal I am using
$status    -- a numeric variable, usually used to retun error codes
$prompt    -- what I am currently using for a prompt
$shell     -- which shell am I using  (usu. either /bin/csh or /bin/sh)
```

These variables can be found by typing:

```
     % set
```

from the prompt.

Array variables

Not all variables are single values. Some consist of a list of values, which are variously dubbed "arrays", "multi-word" variables or "lists" (3 names for the same thing). We will call them arrays herein, but they really are just lists of values. The lists are dynamic in size, meaning that they can shrink or grow.

To create an array out of a single value, use the parentheses. For example, the following creates a list of four names:

```
    set name = (mark sally kathy tony)
```

You can still retrieve the value of this variable by doing $name, but in doing so you get the whole list.

A new syntax is used to find out how long an array is: $#name, such as:

```
   echo $#name
```

which will print out 4. The value of $#name is always an integer, and can be used in several settings.

To access elements in an array, square brackets surround a subscript expression such as

```
   echo $name[1]
   echo $name[4]
```

If you give too high a subscript, Cshell prints "Subscript out of range".

There are many handy shortcuts that you can use in Cshell subscripts that are not possible in C. For instance, you can specify a range of subscripts. The range can even be one-ended so that you can specify, for example, all elements from 5 to the end:

```
   echo $name[2-3]
   echo $name[2-]        # all elements from 2 to the end
   echo $name[1-3]
```

The subscript can itself be a variable, such as

```
   echo $name[$i]
```

You can add to an array in several ways, all involving reassignment to the variable using parentheses. For example to add something to the end, you specify the current value of the variable followed by the new item, all surrounded by parentheses:

```
   set name = ($name doran)
```

Likewise you can add to the beginning:

```
   set name = (doran $name)
```

The size of the array also changes, naturally. To add to the middle of the array, you need to specify two ranges. For example, if your array is 5 elements long, say (mark kathy sally tony doran) and you wanted to add alfie between kathy and sally, you could do

```
   set name = ($name[1-2] alfie $name[3-])
```

Likewise, you could remove a middle element by specifying two ranges inside parentheses.

```
@ k = 2
@ j = 4
set name = ($name[1-$k] $name[$j-])
```

Unfortunately, you cannot put arithmetic expressions inside the brackets, so you must use extra variables.

The shift command gets rid of the first element of an array. For example, if name contains (mark kathy sally), then

```
shift name
```

will get rid of the first element and move the remaining down by 1. If no argument is given, then it shifts the built-in array variable argv.

```
shift
shift names
```

In fact, shift is a holdover from Bourne shell programming which does not have arrays. When a shell script examines its arguments, it often makes note of what options were requested, and then moves on to the next option. Shift makes this a whole lot easier.

Here's a typical example:

```
while ($#argv > 0)
    grep $something $argv[1]
end
```

In conclusion, the arguments to a shell script are put into the array variable argv, so that the first argument is $argv[1], the second is $argv[2], etc. As another holdover from the Bourne shell, $1 is a shorthand for $argv[1], $2 for $argv[2], and so forth. But the Bourne shell expression $* which stood for all arguments will not work. You must use either $argv[*] or just $argv.

Switch statements

The switch statement provides a multi-way branch, much as in C. However, several keywords differ from C. Here's the general format:

```
switch ( expression )
   case a:
          commands
          breaksw
   case b:
          commands
          breaksw
endsw
```

Notice that breaksw is used instead of just break, unlike C. Another major difference is that the commands for a particular case MUST NOT be on the same line. Thus, the following would be wrong:

```
  case a: commands
```

The reason for this lies in the fact that the Cshell language is interpreted, not compiled.

The values in the cases do not need to be integers or scalar values. They can be "words" from an array. The following might be inside a while loop:

```
switch ($argv[$i])
   case quit:
          break        # leave the switch statement
```

```
        case list:
               ls
               breaksw
        case delete:
        case erase:
               @ k = $i + 1
               rm $argv[$k]
               breaksw
      endsw
```

Here document

We know how > and < work in I/O redirection. There is a use for >>, namely to append data to the end of an existing file. What about <<? Logically, this should deal with some form of input, and it does. If you want to create a file inside a shell script and get the data from the script itself, rather than from another separate file or from the user, you create a HERE document.

Following << is a symbol, usually a word, often in capital letters. The cshell takes the next line and all lines following until it finds the same word in column 1 as the input to be sent into the command using the <<. Here's a simple example of a shellscript that looks up your friends and family in a small data-base:

```
#
grep $i <<HERE
John Doe   101 Surrey Lane    London, UK    5E7 J2K
Angela Langsbury   99 Knightsbridge, Apt. K4     Liverpool
John Major  10 Downing Street  London
HERE
```

Here's an example of creating a temporary file:

```
cat > tempdata <<ENDOFDATA
53.3 94.3 67.1
48.3 01.3 99.9
42.1 48.6 92.8
ENDOFDATA
```

You can use any symbol to mark the end of the HERE document. The only require-ment is that it must match << and be in column 1.

Be careful about symbols in your here document because alias, history, and variable substitutions are performed on the lines of the here document. This can actually be quite useful. Just put $variables into your here document if you want to customize the here document.

Remember to clean up files that you might create inside a shell script. For example the numerical data file above, tempdata, is still lingering in the current directory, so you should probably delete it, unless you specifically want it to remain. However, the database of names given to the grep command above does not create an extra file, so it is preferred. But occasionally you need the same file to be given to several commands, and it would be wasteful and error-prone to duplicate it in the shellscript with several here documents.

Executing commands

Occasionally we need to execute a command inside another command in order to get its output or its return code. To get the output, use the backquotes. For example, the following could be put inside a shell script:

```
echo "Hello there `whoami`.  How are you today?"
echo "You are currently using `hostname` and the time is `date`"
echo "Your directory is `pwd`"
```

Of course all of these commands have equivalents in Cshell variables except the date command. Following is a better example:

```
echo "There are `wc -l $1` lines in file $1"
```

Another use of commands is to use their return codes inside conditional expressions. For example, the -s option of the grep command stands for silent mode. It causes grep to do its job without producing any output, but the return code is then used. You cannot see the return code, but you can use it if you surround the command in curly braces:

```
if ({grep -s junk $1}) then
      echo "We found junk in file $1"
endif
```

Notice that if (`grep junk $1`) would not work because this would cause grep's output to be substituted into the expression, but in silent mode, there is no output.

The return code of a shell script is set by the exit statement, which can take an integer argument:

```
exit -1
exit 0
exit 12
```

Good script programmers follow the convention that 0 means "all ok" while a non-zero value indicates some error code. If you use "exit" with no argument, 0 is assumed.

The return code of a C program is set by the exit() system call, which also takes an integer argument. The same convention is followed that 0 is "all ok".

Now the odd thing is that 0 usually means "false" which would cause the if statement to do the false statements. To get around this weird mismatch of conventions, the curly braces invert the return code. That is, if grep finds the string it normally returns 0. But the curly braces turn this into 1 so that the if statement will trigger properly. You do not necessarily have to be aware of this to program Cshell scripts properly. Just follow the conventions.

Recursion

Cshell scripts (and also Bourne shell scripts) can be recursive. This works because each UNIX command is started in its own shell, with its own process and its own process id. This is also the reason shell scripts run so much more slowly, because starting processes is slow. So if the same shell script filename appears inside itself, UNIX just blindly starts up another process and runs the Cshell in it, interpreting the commands in the file.

Recursive shell scripts are very common when the script is naturally recursive with regard to the tree structure. Many built-in UNIX commands allow the -R option to specify that the command is done recursively to all components of the directory:

```
% ls -RC /
```

As an example of a recursive shellscript, here's one that prints the head of each file in the current directory and in every subdirectory. Let us suppose that this shellscript is in a file called "headers":

```
#
foreach i (*)
    if (-f $i) then
        echo "============= $i ==================="
        head $i
    endif
```

```
        if (-d $i) then
            (cd $i; headers)
        endif
    end
```

Note the use of parentheses in (cd $i;headers). The parentheses here mean to do the commands in a new shell, for the cd command normally changes the current directory, which would be disastrous for later functioning of the script, which would have no way to return to the previous directory when it finished. But isolating the commands in their own shell makes this secure and modular.

To run headers, just do

```
% cd <whatever dir you want>
% headers
```

Debugging

There is not much support for debugging in Cshell scripts. You can always rely on the good old standard way of debugging: peppering your code with output statements, echo in this case, to see what is going on. To deactivate some of them without getting rid of them, comment them out by putting a pound sign in column 1.

About the only other support for debugging is using some options to the Cshell. -v is verbose and -x echoes the commands after ALL substitutions are made. In order to use these, however, you cannot run your shellscript by just typing in the name followed by arguments. Rather, you must give the name of the file as an argument itself to the csh command, followed by the arguments to your own script:

```
% csh -vx somescript args
```

Both options are needed because the Cshell does a lot of substitutions (history, alias, and variable) after it reads each line. -vx causes both the original line from the script file to be printed, as well as the revised form after the substitutions are made.

Another handy option is -n, which parses the script commands without execution in order to check for Cshell syntax errors.

```
% csh -n somescript
```

Performance considerations

Shell scripts are interpreted in UNIX. That is, there does not exist a compiler to translate the code into machine language, such as the C or Ada compiler does. As you might have heard, interpreted languages tend to be very slow in execution speed, so do not write a numerical analysis program in the C-shell script language! Most shell scripts run very slowly.

The interpreter of shell scripts is the /bin/csh program itself. The cshell "knows" when it is executing commands from a file as opposed to reading them from the user sitting at a terminal, but it is still the same interpreter program.

Whenever you run a command in a shell, a new copy of the shell is started up (or "forked off", to use proper UNIX lingo). The new copy is actually another process that is also running the csh interpreter. When it starts, it reads the .cshrc file from your home directory in order to learn about any aliases and paths that you may have customized. Thus, if your .cshrc file is long, startup time for a command is long.

There is a way to avoid loading the .cshrc file for a shell script. On the first line of the file containing the script put

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

The comment symbol (#) is actually a special UNIX symbol that means "the name of the program to interpret this file follows me". Thus, /bin/csh appears because it is the interpreter for this file. The bang symbol (!) means not to load the preamble file for this interpreter.

You can even put options to the interpreter on this line. For instance if you wanted the shell script lines echoed for purposes of debugging, you could use instead:

```
#!/bin/csh -vx
```

Generally shell scripts are either short or are used because it is too clumsy to write a C program to make all the file decisions that need to be made. Shell programming is a convenience, and it has a clearly defined niche, but that niche is not general purpose problem solving such as you might use C or Ada for.

Many programmers still use the Bourne shell for shell programming, partly because it is faster and partly because there are more books and examples out there. Since the Bourne shell is simpler and has fewer features, it is a smaller interpreter so starting it up takes less time.

Learning more about shell programming

We have only just skimmed the surface! Cshell programming is about as deep and as complex as any other kind of programming. Indeed you could write all sorts of programs in Cshell, but they would be terribly slow compared to their C or Pascal counterparts.

Many important topics have been omitted from this tutorial, such as the role of environment variables and how variables' values are either inherited or lost. But with this tutorial almost 920 lines long, some cuts had to be made!

Whole books have been written about shell programming, although most of them focus on the Bourne shell, which is still widely used. The best book that teaches about Cshell programming is

"An Introduction to Berkeley UNIX" by Paul Wang.
Wadsworth Pub. Co., 1988, 512 pages, paperback.

Bouwhuis CALL NUMBER: QA76.76.063 W36 1988

He also shows several complete non-trivial scripts, and their Bourne shell equivalents.

One of the best ways to learn about any programming language is just to read other people's programs and try to discover how the elements of the language are being used. If you stumble across an unfamiliar item, look it up in a reference book or the man page. (Most of the Csh man page is devoted to the minutiae of Cshell programming.) You can look at some of the scripts in the public directory /usr/local/bin, for starters.