C shell script tutorial pdf

A shellscript is an executable file with shell commands. The script acts as a program by running each command in the file sequentials. Each of the 5 common shells has its own scripting language. There are similarities between the languages, but also differences. A scripting language consists of control structures, shell commands, expressions, and variables. If a shellscript written in a particular scripting language is to run under the correct shell, the first line of the script must specify the shell to run under. For example, a C shellscript must have the first line: #!/bin/csh Script files must be allowed to file with the chmod u+x myscript command A simple shell script that appears under one of the 5 shells below. This script simply displays a greeting and the date/time. Comments are preceded with a pound sign (#): #Simple Script #echo 'Welcome to the world of script files' date Start the Shell Scripts Exercises Expressions are statements, composed of constants, variables and operators, that are evaluated to determine a result. Expressions can be mathematical or logical. Example 1: Mathematical expression, where a and t are variables, are 3 and 4 are constants and are = and + operators. a = 3 + 4t Example 2: Logical expression, which evaluates to where or false. The string exit is a constant, var is a variable, and (,) and == are operators. ($var == exit) Shell scripts often use expressions. Each shell has its own rules for writing expressions, however. The C Shell recognizes the following operators, in order of priority. () - hook - change order of evaluation - - unary minus / denial ~ - a supplement ! - logical denial % - rest / - dividing lines * - multiply - - subtract + - addition >> - shift right << - shift left > - larger than < - less than > = - larger than or equal - less than or equal != - not equal != to (strings) == - equal to (strings) & - bitwise AND ^ - bitwise exclusive OR | - bitwise inclusive OR && - logical EN || - logical OR continue with the Shell Scripts Exercises Script languages use programming control structures, such as statements and loops. Those for C Shell are described below. if used to test an expression and then execute a command conditionally. If the specified expression evaluates true, the command runs with arguments. The command should be a simple command, not a pipeline, a command list, or a list of parentheses. Syntax: as (expr) command [arguments] Example: #!/bin/csh as ($#argv == 0) echo There are no arguments In addition to the logical expressions of the C Shell you use expressions that value based on the status of a file. For example: if (-e myfile) echo myfile already exists The possible file status expressions are: d- file is a directory e - file f - file is an ordinary file o - user owns the file r - user has read access to the file w - user has write access to the file x - user has access to the file z run - file is zero bytes long Continue with the Shell Scripts Exercises used to test multiple conditions and to perform more than a single command per condition. If the specified expr is true, the commands are executed to the first other; otherwise if expr2 is true, the commands are executed on the second other, etc. Any number of other-like pairs are possible; Only one endif is needed. The other part is also optional. The words else and endif must appear at the beginning of the command lines; if only appear on the command line or immediately after another. Syntax: if (expr) then else commands like (expr2) than commands other commands endif Example: #!/bin/csh as ($#argv == 0) then echo No number to classify differently as ($#argv > 0) then set number = $argv[1] as ($number < 0) then @ class = 0 different if (0 Continue with the Shell Scripts Exercises The foreach instruction is a kind of loop statement. The variable name is set sequentially to each member of the wordlist and the order of the commands until the corresponding final statement is executed. Both for and before and out should appear only on separate lines. Syntax: front name (wordlist) commands end Sample: #!/bin/csh foreach color (red orange yellow green blue) echo $color end The while statement is a different type of loop instruction. Statements within the while/end loop are conditionally executed based on the evaluation of the expression. Both while and end should appear only on separate lines. Syntax: While (expression) commands end Example: #!/bin/csh set word = something while ($word !=) echo -n Enter a word to check (Back to exit): set word = $< if ($word !=) grep $word /usr/share/dict/words end Continue with the Shell Scripts Exercises used to interrupt the execution of an anterior or loop while. Transfers control to the statement after the end statement, causing the loop to end. If other commands are on the same line as a statement of interruption, they run before the interruption occurs. Multi-level breaks are therefore possible by writing them all on one line. Syntax: end example: #!/bin/csh proposal (one two three exit four) if ($number == exit) then reaches an exit end $number end remains Used to interrupt the execution of a front or while loop. Transfers control to the final statement, continuing the loop. If there are other commands on the same if a follow-up statement is in place, they will be executed before the follow-up line takes place. Syntax: continue Example: #!/bin/csh front number (a two three exit four) if ($number == exit) then echo reached an exit remain endif echo $number end goto De goto statement transfers control to the statement starting with label: Syntax: goto label Example: #!/bin/csh as ($#argv ($#argv 1) goto error1 as ($argv[1] < 6) goto error2 goto OK error1: echo Invalid - wrong number or no arguments echo Stop exit 1 error2: echo Invalid argument - must be greater than 15 echo Quitting exit 1 OK: echo Argument = $argv[1] exit 1 switch / case / breaksw / endsw The switching structure allows you to set up a series of tests and conditionally executed commands based on the value of a string. If none of the labels match before a default label is found, the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The breaksw command ensures that the execution is continued after the final. Otherwise, the check may fall through case labels and standard labels. If no label matches and there is no standard, the execution continues after the final. Syntax: switch (string) case str1: commands breaksw case str2: commands breaksw ... standard: commands breaksw endsw Example: #!/bin/csh if ($#argv == 0) then echo No arguments delivered... exiting exit 1 else switch ($argv[1]) case [yY][eE][sS]: echo Argument men is yes. breakw case [nN][oO]: echo Argument one is no. Breaksw Default: Echo Argument one is neither yes nor no. breaksw endif Continue the Shell Scripts Exercises The onintr statement transfers control when you interrupt (CTRL-C) the shell script. Control is transferred to the instruction starting with label: Can be useful for gracefully cleaning up temporary files and leaving a program should it be interrupted. Syntax: onintr label Sample: #!/bin/csh onintr close while (1 == 1) echo Program is running sleep 2 end close: echo End of program Various tasks for C Shell programming are described below. Use quotation marks The shell uses both single (') quotation marks and double () quotation marks. They have different effects. Single quotation marks: Allow spaces to prevent variable substitution permit file name generation Double quotation marks: Allow spaces to allow variable substitution license file name generation Sample 1: Variable substitution #!/bin/csh set on T=-l set x1='ls $opt' echo $x 1 set x2=ls $opt echo $x 2 The output produces: ls $opt ls -l Example 2: Filel generation #!/bin/bin/csh set ls1='some files: [a-z]*' echo $ls 1 set ls2=some files: [a-z]* echo $ls 2 Sample output (identical): some files: csh .html images man misc other.materials some files: csh.html images man misc other.materials Saving the output of an assignment The shell uses backquotes to obtain the output of the command embedded in the backquotes. This output can be stored within a matrix variable. Each element can then be indexed and processed as necessary. Syntax: set variable = command Example: #!/bin/csh set date_fields='date' echo $date_fields echo $date_fields[1] echo $date_fields[2] for field ('date') echo $field end Monster output: 9 Mar 22:25:45 HST 1995 Do Mar 9 22:25:45 HRT Read User Input Depending on Your System You $ Note: Be careful using $Syntax: variable set = $< - or - set variable = 'head -1' Example: #!/bin/csh echo -n Enter your value: input = $< echo You entered: $input - or - #!/bin/csh echo -n Enter your value: set input = 'head -1' echo You have entered: $input Concludes the tutorial. Back to the Introduction to C Shell Programming contents statement

---

However, the syntax is completely different. This tutorial focuses exclusively on the Cshell, not the Bourne shell.

You do this, in fact, creating a brand new assignment. Other operating systems allow this convenience, especially MS-DOS, which calls such files BAT or batch files. In UNIX, such a file is called a shellscript. First, make sure you are aware of the different UNIX shells (Bourne and C-shell). There is information in the dictionary menu. Both the Bourne shell and the C shell you make shell scripts and use, but because the syntax of the commands that use these two shells is slightly different, your shell script should match the shell that is interpreted, or you get errors. A shell script is just a readable file that you create and in which you have shell commands. The first rule determines which shell program interprets or executes a shellscript. * If the first line starts with a C-shell comment (starting with #in position 1) then this will be interpreted by the C-shell and should use C-shell syntax. * Otherwise, the file like a Bourne shell script. You have comments in either type of shellscript, although the syntax differs. Bourne shell comments start with a colon, while C-shell comments start with the pound sign (#). For the rest of this tutorial, we will focus on the C-shell. Simple scripts ---------- Most shell scripts that writing will be very simple. They will consist of a number of UNIX commands that you would have typed on the prompt, possibly replaced by another file name. These replacements are called positional parameters. To create a shellscript that has no parameters and does the same thing every time it is called, place the commands in a file. Change the permissions for the file to be executable, and then use it. The name of the file should be something that you can easily remember and make sense given the operation you're doing. Let's create one that deletes the screen, the date, time, host name, current workbook, current username, and number of people signed in. The name of the script is status. So edit a file called status and set the following lines: (Don't type the frame of dashes and vertical bars - these are meant to show you what the file looks like.) +---------- | # | clearly | echo -n It is currently: ;d ate | echo -n I'm notified as ;who am i | echo -n This computer is called ;hostname | echo -n I am currently in the directory ;p wd | echo -n The number of people currently logged in is: | which | tolet -l +---------- How to make a file doable and put it in your path ---------- make sure you get the #in line 1. Now set the permissions: % chmod 0700 status This makes it executable and readable, both of which are needed. To use, type simply % state If you see a cryptic command that says command not found, it's probably because your path doesn't contain the current folder. To remedy this, set the dot and slash the name: % ./status or change your path: % set path=($path .) Note the space for the period. Let's explain just a few things in the shell script above. Note that echo -n is widely used. The echo command presses lines to the screen, and normally puts a newline after the thing that prints it. -n inhibits this, so that the output looks better. String together more than one command using a semicolon. So, clear;date;whoami;pwd could be put all on a par 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 allow us to become more complicated by adding positional parameters. Parameters are given after the name of the file when you start the shellscript. Each parameter has a position, first, second, etc. When the shell interpreter reads and executes every line of the shell script file, it searches for symbols like $1, $2, etc and it replaces for these symbols the Parameters. Let's do a very simple example. Our shell script will try to find the word (regardless of the case) in a file we specify 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.), we just need to vary the file name. Suppose we called this file funix for unix find, and we made it doable with 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.), print out every line it found. You have any number of parameters. The second is $2, the third is $3, etc. Another common variation is to refer to all parameters at once using $*. Our little shell script only looks at one file at a time. If we typed funix myjunk yourjunk hunjunk ourjunk it would only search in the first file myjunk. To get it all looking, we could +---------- | # | forachach i ($*) | 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 need a $ at the front when they are used. The final word says this is the end of the foreach construct, not the end of the shell script. In many situations, UNIX commands are set up to accept multiple file names, and grep is one of them. So you could have done + ---------- ---------- | # | grep -i unix $* +---------- place. But not all cases work so easily. All you have to do is know your UNIX commands. Let's look at the parameters syntax. Each parameter is identified by $1, $2, $3 and so on. The name of the command is $0. A short hand for all parameters is $*. To find out how many parameters there are, $#argv is used. Here's an example of the beginning of a shellscript that checks that the user has entered enough parameters, because some scripts require a certain number. For example, Grep needs at least one parameter, the string to be searched for. +---------- | # | if ($#argv < 2) then | echo Sorry, but you have entered too few parameters | echo use: slop file searchstring | exit | endif This example gives you a taste of the if statement syntax, use the echo command to act as a shell script output, and the exit command that immediately ends the shell script. The general syntax of if and if-then-different is: if (expression) then if (( ) then statements where statements endif otherwise false statements endif We will discuss what expressions may go in the brackets 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, the above as-if-instructions patterns do not show the two keywords then and endif in C. The curly brackets of C are not used in the C shell for the same thing, but for some- thing completely different, which can be quite confusing. So it is wrong to suggest that the knowledge of C grants you the ability to write C shell scripts! We start with something that is widely used in as statements, and is not in C: file ques. These are expressions that are used to determine characterists of files so that appropriate action can be taken. Suppose you want to see if there is a particular file: if (-e somefile) then grep $1 somefile else echo Grievous error! Database file does not exist. The name of the file does not need to be hard encoded in the if statement, but can be a parameter: if (-e $2) then here is a full list of the Cshell file ques. -e-file file exists only (can be protected from the user) -r file exists and is readable by the user -w file is described by the user -x file is executable by the user -o file is owned by user -z file has size 0 -f file is an ordinary file -d file is a directory All queries except -e automatic testing for the existence of files. That is, if the file does not exist, then it cannot be described. But -r will fail for one of two reasons: 1.) the file exists, but is not readable by the owner of the process that runs 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 above file ques. They're: -- deny && -- logical and || -- logically 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 -e because not doing ox 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 about priority, use parentheses, but you may need to use spaces before and after the brackets. The c-shell's parser isn't as robust as the C compiler's, so it can easily get confused when things are running together without intermediate spaces. Variables --------- C-shell scripting language variables. The variables can have long, mnemonic names and may be character or numeric, although floating point variables are not allowed. You also create arrays, which will be discussed in a later section. When you refer to the value of a variable in a C-shell statement, you must advance variable name with a dollar sign. The only time you don't use a dollar sign is in the set statement that assigned a value to a variable or changes the value of an existing variable. The layout of the set is set name = expression C shell variables are dynamic. They are not declared, but arise when they are first set. Therefore, you remove them in a shell using unset. non-specified name There is a special value, the NULL value, and it is assigned to a variable by doing set name = without expression. Note that such a variable is still defined, i.e. it still exists, even though it has special NULL value. To actually get rid of the variable, use non-in. To give a drawing value to a variable, use double quotes or forear it. If the string contains special characters, such as an empty one, use double quotes. Here are some examples: set name = Mark echo $name as ($name == Mark) then ... set name = Mark Meyer echo $name set dirname = /usr/local/doc/HELP Is $dirname You find out whether or not a variable is defined using the special form $?var. This can be used in an if statement. For example: if ($? dirname) then $dirname else ... To change the value of a variable, use the set again, but don't use $. set dirname = /mnt1/dept/glorp To add to an existing character variable, you do something like the following: set sentence = Hi set sentence = $sentence there everyone Now $sentence, if echoed, would have Hi there everyone in. The following also works: set sentence = Hi set sentence = $sentence there is everyone There is a special variable called $$ which has the process id number of the process that runs this shell script. Many programmers use this to create unique file names, often in the /tmp folder. Here is an example of copying the first parameter (which is obviously a file name) in a temporary file whose name uses the pid number: cp $1/tmp/tempfile.$$ This will be a file whose name is something like/tmp/tempfile.14506, if the pid number is 14506. In fact, the computer cycles through the pid numbers eventually, but usually the same pid doesn't occur for several days, so there's rarely a need to worry. Using variables in the shell ---------- One of the nice features about Cshell programming is that there is no clear line between what you do in a shell script and what you type in from the prompt. So you set and unplug variables, use them for loops, and do all sorts of things on the command prompt. Some things won't work, like using the parameters $1, $2, etc because there aren't any. But functions work, and using setting variables is quite convenient, especially when you want to use a long, complex pathname repeatedly: % set X = /usr /local/doc/HELP %ls $X % ls $X /TUTORIALS You even embed the shell variables variables other strings, as shown above in $X/TUTORIALS. Of course, you don't follow the shell variable with a string that starts with an alphabetical or numeric character, because the C-shell doesn't know what variable you're talking about, like $XHITHERE. Arithmetic variables ---------- ---------- Variables whose values are integer use a slightly different setup command. The commercial-at character is used instead of set to indicate assignment. Otherwise, the Cshell would use a character string 123 instead of the integer 123. Here are a few examples: @ i = 2 @ k = ($x - 2) * 4 @ k = $k + 1 There is also a – and a + + operator to decrement and increment numerical variables. Be careful not to omen the blank that follows the at-sign! @!- @ i+ + Expressions and true and false ---------- The Cshell works very much like C in that it treats 0 as false and something else as true. The expressions used in if and when instructions use the numeric values. Here's a tellus that uses a numerical variable: @n= 5 while ($n) # doing something @n-- end There are also numerical 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 whole shell script to end! The following statement prompts the user to type in something. If 0 is entered, the while loop ends. Note the use of $< as the input device in the Cshell language, and a shortened as statement that forgoes the use of then and endif: while (1) echo -n Gimme something: set x = $< if (! $x) end break If a variable contains the NULL value, its use in an expression is the same as when it is 0. Boolean conditions ---------- To complete the discussion about operators and conditions, here are the Boolean comparison operators. Keep in mind that some of them are only used for strings, while some are used for numbers only. A string doesn't necessarily have to be surrounded by double quotes (unless it contains special characters, such as spaces or other things.) Expressions and operators == equal (strings or numbers) != not equal (strings or numbers) =~ string match !~ string mismatch <= numerically less than or equal to >= numerically greater than or equal to > numerically larger than < numerically 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 quotation marks, it will say they are the same. Oddly enough, if you omit the double quotation marks when you type in mark, the no longer thinks the variables are the same! Apparently, the double quotes are saved as part of the string when you enter the value through $<. Strings must match exactly, and 0005 and 5 are two totally different strings. However, however have the same numerical value. The following shell script would say that 0005 and 5 are the same: # @x = 5 @y =$< echo $x, $y if ($x== $y) then echo they are the same endif But if you replace the @'s with sets, they would no longer be the same. Input and output ---------- Output is fairly simple. Use echo to display literal and variable values. If you don't want a new line to print, use -n. This is particularly valuable in prompts, as in the while loop in the last part. echo Hi there world echo -n Type in your name: echo The current directory is $cwd $cwd is the current workbook, and is a built-in variable (discussed below). To get something from the user, use $<. This interrupts the shell until the user types a return of the transport. What the user typed for the return is the value that $< returns. This can be used in many different settings: in as conditions, while looping, or in set instructions. set x = $< Of course, if you expect to get something intelligent from the user, make sure its asking for the type of information you ask for! Built-in variables ---------- There are a few built-in variables, such as $cwd and $HOME. $Cwd is the current workbook, which you see when you use pwd. $HOME s your house guide. Here are others: $user -- who am I? $hostess -- name of the computer I'm logged into $path -- my execution path (list of folders to be searched for executables) $term -- what kind of terminal I use $status -- a numerical variable, usually used to re-tune error codes $prompt - which I currently use for a quick $shell - which shell I use (usu. either /bin/csh or /bin/sh) These variables can be found by typing: % set to the prompt. Array variables ---------- Not all variables are separate values. Some consist of a list of values, which are different ways arrays, multi-word variables or lists (3 names for the same). We'll call them arrays in here, but they're really just lists of values. The lists are dynamic in size, meaning they can shrink or grow. To create an array from a single value, use the brackets. For example, a list of four names is created: set name = (mark sally kathy tony) You still get the value of this variable by doing $name, but you get the whole list. A new syntax is used to figure out how long an array is: $#name, such as: echo $#name that will print 4. The value of $#name is always an integer, and can be used in different settings. To access elements in an array, square brackets surround a such as echo $name[1] echo $name[4] If you give a subscript that is too high, Cshell prints Subscript out of range. There are many useful shortcuts that you use in Cshell subscripts that are not possible in C. For example, specify a range Subscript. The range can even be a one-off, so that you 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 add to an array in different ways, reassigning the variable using parentheses. For example, if you want to add something at the end, specify the current value of the variable followed by the new item, all completed by parentheses: set name = ($name doran) Also add you at the beginning: set name = (doran $name) The size of the array also changes, of course. To add to the center of the array, you must specify two ranges. For example, if your array is 5 elements long, for example (mark kathy sally tony doran) and you want to add alfie between kathy and sally, you also do set name = ($name[1-2] alfie $name[3-]), you remove a middle element by specifying two sets within brackets. @ k = 2 @ j = 4 set name = ($name[1-$k] $name[$j-]) Unfortunately you can't put arithmetic expressions in the brackets, so you need to use additional variables. The shift command removes the first element of an array. For example, if the 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, it shifts the built-in array variable argv. Shift Shift Names In fact, shift is a holdover of Bourne shell programming that has no arrays. When a shellscript examines the arguments, it often notices which options have been requested, and then moves on to the next option. Shift makes this a lot easier. Here is a typical example: while ($#argv > 0) grep $something $argv[1] end Finally, arguments to a shell script are put in the array variable argv, so that the first argument is $argv[1], the second is $argv[2], etc. Like another holdover of the Bourne shell, $1 is a steno for $argv[1], $2 for $argv[2], and so on. But the Bourne shell expression $* that stood for all arguments will not work. You need to use $argv[*] or just $argv. Switching instructions ---------- The switch statement provides a multi-way branch, just like in C. However, different keywords differ from C. Here is the general format: switch (expression) case a: commands breaksw case b: commands breaksw endsw Message that breaksw is used instead of just breaking, as opposed to C. Another big difference is that the commands for a particular case should not be on the same line. So, the following would be wrong: case a: commands The reason for this lies in the fact that the Cshell language is interpreted, not composed. The values in the do not have to be integers or scalar values. They can be words from an array. The following can stop within a while loop: switch ($argv[$i]) case: break # let the switch switch case list: Is breaksw case delete: @ k = $i + 1 rm $argv[$k] breaksw endsw Here document ---------- We know how > and < work= in= i/o= redirection.= there= is= a= use= for=> >, namelijk to append data to end of an existing file. Hoe zit <? 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.= the= only= require-= ment= is= that= it= must= match=></ENDOFDATA> < 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 conditional expres- sions. 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 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= examples= 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= commands= 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= expres-= sions.= 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=></ 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 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 expres- sions. 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 > </HERE> </HERE> use it if you sur-round the command in curly braces: if ({(grep-s junk $1)} then echo We found junk in file $1 endif Mark on that if ('grep junk $1') wouldn't work because this would cause the output of grep to be replaced in the expression, but in silent mode, there is no output. The return code of a shell script is set by the exit instruction, which can take an integer argument: exit -1 exit 0 exit 12 Good script programmers follow the convention that 0 means everything is ok, while a non-zero value gives some error code. If you use exit without argument, 0 is assumed. The return code of a C program is set by the system call exit() which also includes an integer argument. The same convention is followed that 0 is all ok. Now the strange thing is that 0 usually means false which would lead to the as statement to make the false statements. To avoid this weird mismatch of conventions, the curly brackets reverse the return code. That is, if grep finds the string it normally returns 0. But the curly brackets put this in 1, so that the submission will lead well. You don't necessarily need to be aware of this to program Cshell scripts properly. Just follow the conven-tions. 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 slower because starting processes is slow. So if the same shell script file name appears in itself, UNIX simply blindly starts another process and runs into the Cshell, interpreting the commands in the file. Recursive shell scripts are very common when the script is naturally recursive with respect to the tree structure. Many built-in UNIX commands make it possible to specify the -R option that the command runs recursively on all parts of the folder: % ls -RC / As an example of a recursive shellscript, here is one that prints the head of each file in the current folder and in each submap. Let's assume that this shellscript is a file headers: #foreach i (*) as (-f $i) than echo==========$i $i $i $i==========

The brackets 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 operation of the script, which would have no way to return return to the previous directory when it's done. But isolating the commands in their own shell makes this safe 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 always rely on the good old standard way of debugging: peppering your code with output statements, echo in </whatever> </whatever> to see what's going on. To deactivate some of them without getting rid of them, comment them off by putting a pound sign in column 1. About the only other support for debugging is the use of a number of options to the Cshell. -v is verbose and -x echoes the commands after all replacements have been made. However, to use it, you do not run your shellscript by simply typing in the name, followed by arguments. Instead, you should give the name of the file as an argument to the csh command itself, followed by the arguments for your own script: % csh -vx somescript args Both options are necessary because the Cshell does many replacements (history, alias, and variable) after it has read each line. -vx ensures that both the original line is printed from the script file and the revised form after the replacements are created. Another convenient option is -n, which dissects the script commands without execution to check for Cshell syntax errors. % csh -n somescript Performance considerations ---------- Shell scripts are interpreted in UNIX. That is, there is no compiler to translate the code into machine language, as the C or Ada compiler does. As you may have heard, interpreted languages tend to be very slow in execution speed, so don't 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 to run commands from a file as opposed to reading them from the user sitting on a terminal, but it's still the same interpreting program. When you run a command in a shell, a new copy of the shell is started (or forked, to use the correct UNIX lingo). The new copy is actually another process that includes running the CSH interpreter. When it starts, it reads the CSHRC file from your home folder to learn about aliases and paths you may have customized. So, if your .cshrc file is long, the 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 scripted file put #!/bin/csh The comment symbol (#) is actually a special UNIX symbol that means that the name of the program to interpret this file follows me. So, /bin/csh seems to be- because it is the interpreter for this file. The bang symbol (!) does not mean to load the preamble file for this interpreter. You even put options to the interpreter on this line. For example, if you wanted the shell script lines echoed for purposes of debugging, you could use it instead: #!/bin/csh -vx Generally shell are either short or used because it is too clunky 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 a general goal as you could use C or Ada for. Many programmers still use the Bourne shell for shell programming, partly because it's faster and partly because there are more books and examples out there. Since the Bourne shell is simpler and has fewer features, it's a smaller interpreter so booting up takes less time. Learn more about shell programming ---------- We've only just skimmed the surface! Cshell programming is about as deep and as complex as any other form 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 environmental variables and how the values of variables are inherited or lost. But with this tutorial almost 920 lines long, some cuts had to be made! Entire 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. Construction house CALL NUMBER: QA76.76.063 W36 1988 He also shows some complete non-trivial scripts, and their Bourne shell equivalents. One of the best ways to learn about a programming language is simply to read other people's programs and try to discover how the elements of the language are used. If you come across an unknown item, look it up in a reference book or the men's page. (Most of the Csh man page is dedicated to the minutiae of Cshell programming.) You look at some of the scripts in the public directory/usr/local/bin, for starters. Starters.