## What is a shell?

The shell is interface for user to computer OS.

The name is misleading as an animal's shell is hard protection and computer shell is for interactive (and non-interactive) communication.

The original shell is sh, the Bourne shell.  Bourne was one of the authors of Unix.

The login shell starts at login and sometimes when an interactive shell is started with no specification of a shell as with creation of a window or a remote operation which isn't necessarily a login (commands can be executed remotely without logging in).

Even this can't be guaranteed as it depends on the programming of whatever creates the window or runs the remote operation.

The shell parses commands which means identifying executable elements, arguments, and special characters which may affect command execution in various ways.  It also includes evaluation of symbols and can include repetition of parsing operations.  Once parsing is completed, execution of commands begins.

## What are other shells?

Other shells (not an exhaustive list) are:

csh, the C-shell; introduced with Berkeley Unix in the mid-1970's and was one of the seminal events in the evolution of Unix; the name is a play on words as its characteristics resemble the C language

tcsh, a derivative of the C-shell with additional features primarily useful for interactive use as opposed to shell programming; "C-shell" is now ambiguous as it might refer to the set of shells deriving from csh including tcsh or only csh

ksh, the Kron shell; Korn was the programmer; an attempt to offer the advantages of csh for interactive use and those of sh for programming; (ksh was introduced before tcsh)

bash, the Bourne-again shell (another play on words); like ksh, an attempt to amalgamate csh's and sh's advantages in one shell; many of the enhancements of ksh are also in bash; bash has more and is in increasing use and is therefore to be preferred to ksh.

bash and ksh are called "Bourne-type shells" because they include the chacteristics of the Bourne shell and are incompatible with some features of csh.

There are other shells, some intended for restricted use.

## More about the various shells

bash has become associated with the various forms of Linux (I'm not sure that this is true of absolutely all forms as there are now many); so far as I know, there is no fundamental compatibility between bash and Linux and this association reflects programmers' proclivities; this is another reason to prefer bash to ksh; ksh still has to be maintained under some OS's because some have OS features written in ksh

Bourne shell scripts should run under ksh or bash.  ksh scripts will probably run under bash.

csh scripts should run under tcsh.  While there is an extensive compendium of sh scripts which were written long ago and are still in use and might now be run under bash, the C-shell was never

extensively used for scripting.

In addition to its advantages for scripting and compatibility with sh and ksh, bash is becoming a kind of lingua franca for these purposes. There are implementations of bash for Windows and VMS (a proprietary operating system of DEC which is still in use). Differences in the form of file pathnames and libraries mean that it isn't possible to simply move a bash script between these various operating systems and use it without modification or at least some consideration of the need for modification. The usefulness of interactive bash shells under Windows or VMS isn't clear to me.

There is no guarantee that all these shells will be available under all forms of Unix, Linux, or MacOS or that shells with the same name on different OS's will be the same.

## Scripts and pecifying which shell will run a script

Scripts are programs consisting of shell commands. They can be written in each of the various shells. Arguments, called "parameters" or "positional parameters", on the command line are referenced in the script by dollar signs and numerals corresponding to their positions on the command line, e.g. $1, $2. There are further possibilities such as "$*" which refers to the entire argument list or "$#" which refers to the number of arguments.

Shell-specifier lines are at the head of scripts and permit running the scripts as commands with indicating a shell in the execution. An example would be

```
#!/bin/bash
```

The "#" is the comment-line character in all shells. The exclamation point is a unique feature to set the executing shell. Just a "#" as the first line can indicate a C-shell but this isn't reliable.

The location of a shell as in the above pathname with /bin in one OS may not be the same as in another. So a specifier line might have to be changed for a different OS.

If the shell-specifier line gets the path wrong, you typically get the error message that the script file you called doesn't exist.

Specifying a shell on the command line, e.g.

```
bash myscript
```

takes precedence over a specifier line in a script. This can be a way of providing for varying pathnames: Rather than using a specifier line, call the script explicitly and the use the particular OS's way of locating commands to locate the shell.

Specifier lines can be used for things other than shells like sed or awk or perl. In the case of sed or awk, this can be difficult as there are things that one would ordinarily do with the shell or other commands even if most of what you are doing is with the utility. In addition, passing arguments (parameters) to such a script may be impossible or confusing. sed, while it can be used to do more than its stated function as a stream editor, can do those things only with difficulty and can't do some ordinary things. awk is more capable and can probably do anything that a shell can do. However even here it may be preferable to call awk from a shell script. perl, on the other hand, is a self-contained language with all the capabilities needed to do what a shell script can do. The drawback to perl is that it can't be run interactively and therefore perl scripts can't be developed with interactive testing and parts of scripts can't be executed independently without creating new scripts. Those considerations haven't prevented perl from becoming a widely used, highly diversified language.

## Calling shells from Linux and MacOS

In all or most forms of Linux and in MacOS, if you type csh you get tcsh and if you type sh you get bash. This can lead to confusion if you write a script using tcsh or bash features while calling csh or sh, respectively, and then try to run it on a computer which doesn't have tcsh or bash, respectively, or doesn't run one of those shells when csh or sh, respectively, is called. This mapping is effected by links or by independently coded files. Do you all know about hard and symbolic links? Even in the case of a link in which the same file is called by more than one name, you might get different functionality with different calling names. A program can determine the name by which it was called and act on that basis. This probably won't affect the execution of tcsh if called as "csh". It might affect bash if called as "sh". There are variations which affect compliance with standards or the capacity to execute things which have already been written. These variations can sometimes be determined by variable definitions or command-line arguments which will take precedence over the way a shell is called.

The current Bourne shell sh even when it isn't bash has enhancements over the original and now resembles the C language more than csh.

The Bourne shell is just sh with no distinction for the type of shell. Why? It was the original shell, there was no other at the time, and there was no reason to distinguish it from another shell. It was "the shell".

## Declaration of login shell

Here is a line from the passwd database from another computer system which defines usernames and their login characteristics for me:

kline:Obwbjm1uTfBTg:360:1024:Doug Kline, CfA Collaborator:/home/kline:/bin/tcsh

Fields in the line are delimited by colons. The login shell is in the last field. I don't know why the originators of Unix included that field since there was no choice of shell at the point. Fortunately they did as recoding the format of the passwd database would be prohibitively difficult as all sorts of things refer to it and depend on its format (changes that could be effected by changes of format are effected in other ways like introducing auxiliary files). The above sort of line is used on most if not all Unix and Linux systems. The Mac has its own system.

## Using interactive shells to develop scripts

Shells can be called as interactive sub-shells. Just "bash" will bring up a bash shell under whatever shell you're running this from. This could be used to try out commands you intend to use in a bash script. Even control structures like loops can be used interactively for such testing or sometimes as interactive operations.

As alluded to earlier, the Bourne-type shells are preferred for programming. They have features like the control structures and conditional statements which are useful primarily or exclusively in scripts and they execute faster, partly because they lack features intended for interactive use which slow down execution and aren't useful in scripting. (More on this later.)

## C-shell aliases including history-type references; shell history mechanism

```
alias l 'ls'
alias lf 'ls -F'
alias la 'ls -a'
```

```
alias ll 'ls -l'
alias lls 'ls -ls'
```

The quotes while not necessary in these cases are recommended in all alias definition except for those cases when it's necessary to omit them or limit them to part of the definition. They would be necessary if meta-characters are in the definition and intended for evaluation on use, e.g.

```
alias lgs 'ls g*'
```

Without the quotes,

```
alias lgs ls g*
```

will expand the "g*" to the files with names beginning with "g" when the above command defining the alias is executed and the resulting alias will be ls followed by a list of those file names in the directory in which the above command is executed regardless of what files are in the directory in which one runs lgs.

Here's another alias:

```
alias gr 'grep'
```

An alias can include command elements other than a command, e.g.

```
alias grabc 'grep abc'
```

To insert an argument into the middle of an alias definition, use history-type references, e.g.

```
alias grstrings_file 'grep \!^ strings'
```

The exclamation point is part of the history mechanism of csh which can be used to recall earlier commands or parts thereof. The emacs- (or vi-) type history operations are available with tcsh. They are easier to use interactively. The exclamation-point syntax was present in the orignal csh. The quotes and the back-slash constitute two levels of escaping. Only a back-slash can escape a history reference. The exclamation point can be followed by a specification of a line from the history. When no line is specified which is what happens here, then the immediately previous line is used. In an alias definition, what would be a reference to the immediately previous command in an actual history operation now refers to the command in which the alias is used. Although what would be references to earlier lines in a history list can be used in an alias definition, they aren't as their results would depend on earlier commands of the shell in which the alias was called which could be any number of things. (One might try to use such a reference to recall commands for purposes other than executing them or parts of them. I've never seen that and find it unlikely.) The escaping of the history-type reference is necessary to forestall its being evaluated in the command in which the alias is defined. The definition won't have the back-slash and so when the alias is used the history-type reference will be evaluated.

The caret ("^") is a symbol for the first argument in a command which here would mean the first argument to the use of the alias. When that argument specifier occurs right after the exclamation point, then the reference is to the previous command. The numeral 1 could be used for the first argument also. However then we would have to write "!:1", using the colon to indicate that what follows refers to arguments as "!1" would refer to command no. 1 in the history list, not the first argument to the previous command. Other arguments can be specified by numerals or other symbols which are described in the sections on history references in the man pages on C-shells.

If there is a history-type reference, then only arguments to which there are such references are used. So in the above example, if grstrings_file were called with multiple arguments, only the first would be used.

tcsh, ksh, and bash offer history operations using the commands of the editors emacs and vi.  Whichever editor you use for editing, I think the emacs-type history mechanism is much easier to use. Control-p or the up arrow recalls the previous command and this can be repeated.  Control-n or the down arrow gets the next command and is again repeatable.  Control-a goes to the beginning of the current line and Control-e to the end.  Control-b or the left arrow goes one character back and Control-f or the right arrow one character forward.  Escape-B and Escape-F move by one word.  Control-number multiplies the following motion by that number.  There are other operations documented in the man pages.

C-shell variables

What are variables in shells?  I'm glad you asked.

Variables, as in programming languages (you know what those are, don't you?), assign symbols.  In C-shells, this done with the "set" command, e.g.

set a=b

a perfectly useless definition which could be used with the reference "$a" if anyone had any use for it.  If a variable doesn't already exist, the set command will create it.

The positional parameters which are evaluated with dollar signs can be considered a kind of variable although the rules for evaluation which can be more than simple replacement of a dollar-sign expression with a value may be different in some respects.

These variables can be used for the same sorts of purposes as in programming languages.  If you type just "set" with no arguments, you get a list of all currently defined variables.

One odd aspect of the variable assignment is that there must either be no spaces around the equals sign or spaces on both sides, i.e.

set a=b

and

set a = b

will work but

set a =b

and

set a= b

won't.

set a= b

will run but the effect will be to define both a and b as variables without values.  A variable assignment with no equals sign or an equals sign with nothing after it assigns a null value.  It's possible to make multiple assignments with one set command.

set a =b

won't run because "=b" will be interpreted as an assignment expression and the equals sign isn't a valid character in a variable name.

As alluded to earlier, there are further possibilties here such as qualifiers beginning with colons, e.g.

$a:r which will eliminate an extension to a filename, e.g. if a is set to /a/b/c.o, $a:r will be evaluated to /a/b/c; it will have no effect is there is no extension and applies only to the pathname

element after the last /;

$a:h which will replace a pathname with the part before the last element if the last element doesn't end with a slash, i.e. a variable value of /a/b/c will be replaced by /a/b while /a/b/c/ will be passed on unchanged;

these qualifiers don't include determining whether there is such a pathname, i.e. the above example will still work even if /a/b doesn't exist.

In Bourne-type shells, variables are handled differently.

The meaning of the dollar sign can be escaped with either the back-slash or single quotes. In all shells, while double quotes will escape some things, they will force the evaluation of a dollar-sign expression as a variable even if it would otherwise not be evaluated. This is an unusual effect of an escape character. If a dollar-sign expression is to be within nested single and double quotes, you should test whether it will be evaluated as a variable. Double quotes inside of single quotes won't necessarily give the same result as the reverse.

## Environment variables and bound C-shell variables

C-shell variables come in two varieties: environment variables and shell variables.

The variables discussed before are shell variables. Environment variables are automatically passed on to shells which are called from a shell in which they are defined which includes being passed on to commands called from a shell; when a command is called which is not built into the shell (more about that later), the shell calls a new shell and then that new shell is over-written by the command without over-writing the environment variable definitions such that they are available to the command. In the case of built-in commands, the variables are available just as they are to all other shell functions. Here we see the meaning of "environment": The environment variables are passed on and thus constitute part of a broad environment encompassing the shell and operations originating therefrom. (The term is used ambiguously as it can also refer to all the conditions including shell variables, aliases, and other things. In that usage, its meaning is broader than that of environment variables in including other things and narrower in not extending past the shell. You might find other things in a called shell that seem to be passed on from the calling shell like aliases and shell variables. That can occur because they are defined in initialization files which are executed when sub-shells start up. They aren't passed on automatically; rather the initialization files are executed automatically. If a sub-shell is called without such execution which is possible or the initialization files are changed after a shell is started and before a sub-shell is started, then such initialization commands may be omitted.)

Two environment variables are PRINTER and LPDEST. Some commands which print will seek one or the other of these variables in the environment to determine which printer to use as there can be more than one printer available. If a printer is specified on the command line, that one will be used. Otherwise, if the appertinent environment variable is defined, its value will be used. Otherwise a system default will be used. The environment variable thus defines a personal default. Various commands have their own associated environment variables. Note that there is no guarantee that a command will use an environment variable which may seem to be associated with it. The PRINTER and LPDEST variables are used because a command is programmed to refer to one of the other. A command could be written to send things to printers which doesn't read either of those variables in which case the variables' values will be irrelevant.

Environment variables are defined with the command "setenv" as in

setenv PRINTER someprinter

with no arguments, "setenv" will display all the currenty defined environment variables and their values.  With one argument, it will define that variable to a null value.  "printenv" with no arguments will do that too.  "printenv" with one argument will display that argument's value if there is such an environment variable.  It can't take more than one argument.  (There are other commands too.)

The modifiers like ":h" which can be applied to shell variables didn't work for environment variables in the standard csh but work with tcsh.

In the C-shells, with major exceptions shell variables and environment variables are independent entities.  There can be environment and shell variables with the same names.  In such a case, the value of one won't affect the other.  Evaluation with the dollar sign can't reliably be expected to be one or the other.  In my tests, I got the shell variable but this isn't documented and therefore isn't guaranteed.  Is creating shell and environment variables with the same names a good idea?

What are the major exceptions?

Environment variable TERM and shell variable term which declare the type of terminal or emulation which can determine the sequences for rearranging the screen as with editors among other things

Environment variable USER and shell variable user which declare the username

Environment variable HOME and shell variable home which declare the user's home directory

Environment variable PATH and shell variable path (described later)

One just about never has occasion to re-define HOME or USER.  Doing so can have all sorts of undesirable effects.  One occasionally might want to re-define TERM.  If it's set to the wrong value, things won't work right.  The most likely case now is that it will be set to a value which is not in the operating system's database of terminal types.

These bindings cannot be undone nor can such bindings be established for other variables.

There could be shell variables with the capital-letter names or environment variables with the small-letter names.  Such variables will be independent of the above pairs.

The pattern of a name in capital letters for the environment variable and small letters for the shell variable in these cases doesn't imply that other such pairs of variable names will be bound.  There can be such variables with the names "shell" and "SHELL" which are automatically defined and not bound to each other (and also not reliable).

The path variable is a list of pathnames in which the shell looks for commands (the "where" command reports the locations).  This is very important and has various ramifications.  The shell variable is an array in which the list of elements is delimited by parentheses and the elements are separated from each other by spaces.  The environment variable is a colon-separated list and is not an array.  Arrays are available for C-shell shell variables and not for environment variables.  Because array-handling structures are available, one ordinarily works with the array-type shell variable.  It would be more complicated to select individual pathnames or make sustitutions with the environment variable with commands which work on strings

Changing the value of either variable in a bound pair should change the bound variable.  I found that changing the value of the environment variable TERM can sometimes not change the value of the shell variable term while changing the shell variable's value changes the environment variable's value.  This is a bug.

Variables in Bourne-type shells:

In Bourne-type shells, variables are assigned by equation-like expressions, e.g.

```
a=b
```

There must be no spaces.

All variables are shell variables. Environment variables are a sub-set of the shell variables. They are distinguished from the non-environment variables by exporting. The command to do that, not surprisingly, is "export", viz.

```
export PRINTER
```

You may see the expression "export to the environment". This expression is used in documentation of the C-shell where there is no command "export" which is confusing. With no arguments, "export" will report a list of exported variables.

In bash, the export command can set a variable's value and in doing so create it if it doesn't already exist, e.g.

```
export a=value
```

In the Bourne shell these must be separate commands.

"set" shows list of variables but assigns to argument variables (positional parameters). This function while often used can't readily be explicated today. "set" also can change shell behavior and with options can work with shell variables.

The original Bourne shell didn't have arrays. bash does. bash also has operations on arrays which go beyond the C-shells'.

The "typeset" command can be used to set variable types, e.g. integer, character string. Use of this command isn't mandatory but is fairly common.

where command and command substitution

The "where" command in C-shells shows the pathnames (location) of commands.

What is command substition? A command within a pair of back-quotes ("`") is part of a longer command. The back-quoted command is executed and the output is then substituted for the back-quoted string before the longer command is executed.

This command uses "where" and back-quote syntax for command substitution:

```
ls -l `where sh bash csh tcsh ksh`
```

and here are alias definitions for such commands which use alias history-style references:

```
alias lwh 'ls -l `where \!* `'
alias fwh 'file `where \!* `'
```

This kind of command might not work. Try "where where" and "lwh where"

bash has an alternative to the back-quote syntax for command substitution:

$(command)

which has advantages like permitting nesting.  This is an example
of the use of dollar-sign expressions.  The use of the dollar sign
derives from variable evaluation.

"type" is used in Bourne-type shells for locating commands.

built-in commands

Like the "where" command, some commands are part of shell.  Execution
of these built-in commands doesn't mean reading a file into memory
and then transfering execution to that file.  Rather, the execution
shifts to part of the shell which is already in memory.  This can
be faster.  Built-in commands always take precedence over other
commands with the same names.  So changing the PATH won't affect
whether one gets a built-in command.  To run a command of the same
name, you must call it by the full pathname.  You can define an
alias to get such a pathname command and the alias can have the
same name as the built-in command.  The alias definition is substituted
before execution of a command begins (this isn't obvious for a
built-in command as both are shell operations).

There are command names like "echo" which apply both to built-in
commands and pathname commands and don't necessarily have the same
behavior in both.  The original echo command was in the PATH.  Now
it's there and in the shells.  They don't have the same functionality.
To confuse things further, since the pathname command predates the
built-in commands and had more than one type of functionality
depending on the environment, the C-shell built-in command has
functionality which is intended to emulate the pathname command
which includes that variability.  The built-in command wouldn't
have been written that way if it had always been in the shell.  The
bash version is yet another.  In tcsh, the functionality can be set
by a shell varible echo_style.  This wouldn't apply in the standard
csh but remember that if you execute csh in Linux or MacOS you get
tcsh.

This illustrates that shell variables can be used to affect the
execution of built-in commands just as environment variables can
be used for pathname commands.  Only the shell has access to shell
variables and so they can't influence the execution of pathname
commands

Parsing of built-in commands in C-shells is different from that of
non-built-in commands (I'm not sure about bash and so you shouldn't
make assumptions about it).  The sequence of evaluations is different.
This matters if the results of evaluation of expressions like command
substitutions is intended to precede another parsing operaion.  This
means that commands might have unanticipated effects.  If you write
a command string for a pathname command and then try to use the
same structure for a built-in command it might not work.

An interesting command:

What does this command do?

```
set noglob;eval `tset -Qs -m 'ethernet:vt100' -m 'network:vt100' \
-m 'dialup:?vt100'`;unset noglob
```

Its purpose is not so relevant now.  It illustrates characteristics
of shell parsing and implications for built-in commands.

umask Command, Unix Origins:

umask:

umask 022

Why is there a single command for this?  Why not a variable which seems more appropriate to me?

Origins of Unix

Unix was constructed as a programming project at Bell Labs in the 1970's.  It wasn't intended to be used extensively.  People wrote different parts of it with only a sketchy overall plan and as a result there are inconsistencies.

C-shell not advantageous for programming; script execution speed

It executes more slowly.  There is a trade-off between the features which are intended for interactive use and speed as those features require processing.  The features added to the Bourne shell to make the Korn and Bourne-again shells which are also intended to enhance interactive use don't slow them down to the level of the C-shell.  I don't know why.  I think it has to do with the progamming itself rather than the results for application.

Speed isn't really such a big deal in general.  Although interpreted shell scripts run more slowly than equivalent compiled programs, my experience is that even long scripts of hundreds of lines execute fairly quickly.  Calculational programs take more processing and shell scripts aren't ordinarily used for such purposes.  Scripts thousands of lines in length are used for OS patching and can take some time to execute but those are run only once for each set of patches and one usually gets patches to run at longer intervals.

The Bourne-type shells have more advanced structures like the loops and conditional statements.  The Korn and Bourne-again shells also have features of array processing, a shell operation intended for calculations, a superior form of command substitution, $(command) (mentioned earlier), which, unlike the back-quote syntax (which is also available) allows for nesting and easier-to-read commands, and other enhancements.

An illustration of a drawback of the C-shell:

Another interesting command:

set a=`printenv variablename`

What does this do?  Why use it?

man pages

The man pages are hard to read.  Searching for pertinent strings is a possible approach as with other files but that doesn't work well here because strings are repeated so many times.  A possible, not-terribly fast approach is to keep trying different search strings or something you remember even if it isn't a subject.

There may be different sets of man pages on different operating systems.  There could be distinct pages for csh and tcsh as these might be distinct commands.  Similarly, if sh doesn't really call bash, there could and probably would be a separate man page.  There are systems with man pages for C-shell built-in commands.  Man pages for commands like "echo" can be relevant for, as mentioned earlier, the built-in commands can be written to emulate previously written non-built-in commands.