cat /usr/spool/public/unix_stuff/csh.doc

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

_ A_ B_ S_ T_ R_ A_ C_ T

_ C_ s_ h is a new command language interpreter for UNIX* systems. It incorporates good features of other shells and a _ h_ i_ s_ t_ o_ r_ y mechanism similar to the _ r_ e_ d_ o of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to _ c_ s_ h are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with _ c_ s_ h is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

March 10, 1984

---

*UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

_ I_ n_ t_ r_ o_ d_ u_ c_ t_ i_ o_ n

A _ s_ h_ e_ l_ l is a command language interpreter. _ C_ s_ h is the name of one particular command interpreter on UNIX. The primary purpose of _ c_ s_ h is to translate command lines typed at a terminal into system actions, such as invocation of other programs. _ C_ s_ h is a user program just like any you might write. Hopefully, _ c_ s_ h will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX programmer's manual. The _ c_ s_ h documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in _ i_ t_ a_ l_ i_ c_ s. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

_ A_ c_ k_ n_ o_ w_ l_ e_ d_ g_ e_ m_ e_ n_ t_ s

Numerous people have provided good input about previous versions of _ c_ s_ h and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

_ 1. _ T_ e_ r_ m_ i_ n_ a_ l _ u_ s_ a_ g_ e _ o_ f _ t_ h_ e _ s_ h_ e_ l_ l

_ 1._ 1. _ T_ h_ e _ b_ a_ s_ i_ c _ n_ o_ t_ i_ o_ n _ o_ f _ c_ o_ m_ m_ a_ n_ d_ s

A _ s_ h_ e_ l_ l in UNIX acts mostly as a medium through which other _ p_ r_ o_ g_ r_ a_ m_ s are invoked. While it has a set of _ b_ u_ i_ l_ t_ i_ n functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

_ C_ o_ m_ m_ a_ n_ d_ s in the UNIX system consist of a list of strings or _ w_ o_ r_ d_ s interpreted as a _ c_ o_ m_ m_ a_ n_ d _ n_ a_ m_ e followed by _ a_ r_ g_ u_ m_ e_ n_ t_ s. Thus the command

```
mail bill
```

consists of two words. The first word _ m_ a_ i_ l names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of _ d_ i_ r_ e_ c_ t_ o_ r_ i_ e_ s for a file with the name _ m_ a_ i_ l which is expected to contain the mail program.

The rest of the words of the command are given as _ a_ r_ g_ u_ -

_ m_ e_ n_ t_ s to the command itself when it is executed. In this case we specified also the argument _ b_ i_ l_ l which is interpreted by the _ m_ a_ i_ l program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the _ m_ a_ i_ l command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill
EOT
%
```

Here we typed a message to send to _ b_ i_ l_ l and ended this message with a | ^D which sent an end-of-file to the mail program. (Here and throughout this document, the notation ``| ^_ x'' is to be read ``control-_ x'' and represents the striking of the _ x key while the control key is held down.) The mail program then echoed the characters `EOT' and transmitted our message. The characters `% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the `% ' prompt the shell was reading command input from our terminal. We typed a complete command

`mail bill'. The shell then executed the _ m_ a_ i_ l program with argument _ b_ i_ l_ l and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a | ^D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another `% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is

the _ t_ s_ e_ t command, which sets the default _ e_ r_ a_ s_ e and _ k_ i_ l_ l
characters on your terminal - the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is `#' and the kill character is `@'. Most people who use CRT displays prefer to use the backspace (| ^H) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

```
tset -e
```

which tells the program _ t_ s_ e_ t to set the erase character, and its default setting for this character is a backspace.

_ 1._ 2. _ F_ l_ a_ g _ a_ r_ g_ u_ m_ e_ n_ t_ s

A useful notion in UNIX is that of a _ f_ l_ a_ g argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character `-' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current _ w_ o_ r_ k_ i_ n_ g _ d_ i_ r_ e_ c_ t_ o_ r_ y. The option -_ s is the size option, and

```
ls -s
```

causes _ l_ s to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The _ l_ s command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to

remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

_ 1._ 3. _ O_ u_ t_ p_ u_ t _ t_ o _ f_ i_ l_ e_ s

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called `now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default _ s_ t_ a_ n_ d_ a_ r_ d _ o_ u_ t_ p_ u_ t for the date command and the date command prints the date on its standard output. The shell lets us _ r_ e_ d_ i_ r_ e_ c_ t the _ s_ t_ a_ n_ d_ a_ r_ d _ o_ u_ t_ p_ u_ t of a command through a notation using the _ m_ e_ t_ a_ c_ h_ a_ r_ a_ c_ - _ t_ e_ r `>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the _ d_ a_ t_ e command such that its standard output is the file `now' rather than the terminal. Thus this command places the current date and time into the file `now'. It is important to know that the _ d_ a_ t_ e command was unaware that its output was going to a file rather than to the terminal. The shell performed this _ r_ e_ d_ i_ r_ e_ c_ t_ i_ o_ n before the command began executing.

One other thing to note here is that the file `now' need not have existed before the _ d_ a_ t_ e command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option _ n_ o_ c_ l_ o_ b_ b_ e_ r exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with `>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a `#' character, this `scratch' character denotes the fact that the file will be a scratch file.* The system will remove

---

*Note that if your erase character is a `#', you will have to precede the `#' with a `\'. The fact that the `#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files.

such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the _ d_ a_ t_ e command above, we don't really want to save the output forever, so we would more likely do

```
date > #now
```

_ 1._ 4. _ M_ e_ t_ a_ c_ h_ a_ r_ a_ c_ t_ e_ r_ s _ i_ n _ t_ h_ e _ s_ h_ e_ l_ l

The shell has a large number of special characters (like `>') which indicate special functions. We say that these notations have _ s_ y_ n_ t_ a_ c_ t_ i_ c and _ s_ e_ m_ a_ n_ t_ i_ c meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of _ q_ u_ o_ t_ a_ t_ i_ o_ n which allows us to use _ m_ e_ t_ a_ c_ h_ a_ r_ a_ c_ t_ e_ r_ s without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via _ m_ a_ i_ l, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with `% '.

_ 1._ 5. _ I_ n_ p_ u_ t _ f_ r_ o_ m _ f_ i_ l_ e_ s; _ p_ i_ p_ e_ l_ i_ n_ e_ s

We learned above how to _ r_ e_ d_ i_ r_ e_ c_ t the _ s_ t_ a_ n_ d_ a_ r_ d _ o_ u_ t_ p_ u_ t of a command to a file. It is also possible to redirect the _ s_ t_ a_ n_ d_ a_ r_ d _ i_ n_ p_ u_ t of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the _ s_ o_ r_ t command with standard input, where the command normally reads its input, from the file `data'. We would more likely say

```
sort data
```

letting the _ s_ o_ r_ t command open the file `data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

If you are using a CRT, your erase character should be a | ^H, as we demonstrated in section 1.1 how this could be set up.

then the sort program would sort lines from its _ s_ t_ a_ n_ d_ a_ r_ d _ i_ n_ p_ u_ t. Since we did not _ r_ e_ d_ i_ r_ e_ c_ t the standard input, it would sort lines as we typed them on the terminal until we typed a | ^D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a _ p_ i_ p_ e_ l_ i_ n_ e. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which _ l_ s sorts. We could look at the many options of _ l_ s to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the _ s_ o_ r_ t command, combining it with _ l_ s to get what we want.

The -_ n option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the _ l_ s command run with the option -_ s is to be _ p_ i_ p_ e_ d to the command _ s_ o_ r_ t run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the -_ r reverse sort option and the _ h_ e_ a_ d command in combination with the previous command doing

```
ls -s | sort -n -r | head -5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the _ s_ o_ r_ t command asking it to sort numerically in reverse order (largest first). This output has then been run into the command _ h_ e_ a_ d which gives us the first few lines. In this case we have asked _ h_ e_ a_ d for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the _ p_ i_ p_ e mechanism. Commands separated by `|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the

routing of information to the line printer.

_ 1._ 6. _ F_ i_ l_ e_ n_ a_ m_ e_ s

Many commands to be executed will need the names of files as arguments. UNIX _ p_ a_ t_ h_ n_ a_ m_ e_ s consist of a number of _ c_ o_ m_ p_ o_ n_ e_ n_ t_ s separated by `/'. Each component except the last names a directory in which the next component resides, in effect specifying the _ p_ a_ t_ h of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory `etc' which is a subdirectory of the _ r_ o_ o_ t directory `/'. Within this directory the file named is `motd' which stands for `message of the day'.
A _ p_ a_ t_ h_ n_ a_ m_ e that begins with a slash is said to be an _ a_ b_ s_ o_ -
_ l_ u_ t_ e pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the _ r_ o_ o_ t).
_ P_ a_ t_ h_ n_ a_ m_ e_ s which do not begin with `/' are interpreted as starting in the current _ w_ o_ r_ k_ i_ n_ g _ d_ i_ r_ e_ c_ t_ o_ r_ y, which is, by
default, your _ h_ o_ m_ e directory and can be changed dynamically by the _ c_ d change directory command. Such pathnames are said to be _ r_ e_ l_ a_ t_ i_ v_ e to the working directory since they are found

by starting in the working directory and descending to lower levels of directories for each _ c_ o_ m_ p_ o_ n_ e_ n_ t of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and `.'s (periods). In fact, all printing characters except `/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character `.' (period) is not a shell-metacharacter and is often used to separate the _ e_ x_ t_ e_ n_ s_ i_ o_ n of a file name from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a _ b_ a_ s_ e portion of a name (a base portion being that part of the name that is left when a trailing `.' and following characters which are not `.' are stripped off). The file `prog.c' might be the source for a C program, the file `prog.o' the corresponding object file, the file `prog.errs' the errors resulting from a compilation of the program and the file `prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation


```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with `prog.'. The character `*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the _ a_ r_ g_ u_ m_ e_ n_ t _ l_ i_ s_ t of the command. Thus
the command

```
echo prog.*
```

will echo the names

```
        prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we listed them above. The _ e_ c_ h_ o command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by _ f_ i_ l_ e_ n_ a_ m_ e _ e_ x_ p_ a_ n_ s_ i_ o_ n of the one input word.

Other notations for _ f_ i_ l_ e_ n_ a_ m_ e _ e_ x_ p_ a_ n_ s_ i_ o_ n are also available. The character `?' matches any single character in acan't open file tabs

filename. Thus

```
        echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between `[' and `]'. This metasequence matches any single character from the enclosed set. Thus

```
        prog.[co]
```

will match

```
        prog.c prog.o
```

in the example above. We can also place two characters around a `-' in this notation to denote a range. Thus

```
        chap.[1-5]
```

might match files

```
        chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for


```
        chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an _ a_ r_ g_ u_ m_ e_ n_ t _ l_ i_ s_ t) contains filename
expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character `.' at the beginning are treated specially. Neither `*' or `?' or the `[' `]' mechanism will match it. This prevents accidental matching of the filenames `.' and `..' in the working directory which have special meaning to the system, as well as other files such as ._ c_ s_ h_ r_ c which are not normally visible. We will discuss the special role of the file ._ c_ s_ h_ r_ c later.

Another filename expansion mechanism gives access to the pathname of the _ h_ o_ m_ e directory of other users. This notation consists of the character `~' (tilde) followed by another users' login name. For instance the word `~bill' would map to the pathname `/usr/bill' if the home directory for `bill' was `/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a `~' alone, e.g. `~/mbox'. This notation is expanded by the shell into the file `mbox' in your _ h_ o_ m_ e directory, i.e. into `/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used _ c_ d to change to another directory and have found a file you wish to copy using _ c_ p. If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters `{' and `}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms

because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character `*'. It will either echo an sorted list of filenames in the current _working directory_, or print the message `No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, `/', `.' or `-' in an argument word to a command is to enclose it with single quotation characters `', i.e.

```
echo '*'
```

There is one special character `!' which is used by the _history_ mechanism of the shell and which cannot be _escaped_ by placing it within `' characters. It and the character `' itself can be preceded by a single `\' to prevent their special meaning. Thus

```
echo \'\!
```

prints

```
'!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

```
echo \''*'
```

which prints

```
'*
```

since the first `\' escaped the first `'' and the `*' was enclosed between `'' characters.

_ 1._ 8. _ T_ e_ r_ m_ i_ n_ a_ t_ i_ n_ g _ c_ o_ m_ m_ a_ n_ d_ s

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT _ s_ i_ g_ n_ a_ l to the _ c_ a_ t command by typing the DEL or RUBOUT key on your terminal.* Since _ c_ a_ t does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that _ c_ a_ t has terminated and prompts you again with `% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like _ c_ a_ t did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the _ m_ a_ i_ l program in the first example above was terminated when we typed a | ^D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing `logout'; UNIX then logs you off the system. Since this means that typing too many | ^D's can accidentally log us off, the shell has a mechanism for preventing this. This _ i_ g_ n_ o_ r_ e_ e_ o_ f option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a | ^D. This is because it read to the end-of-file of our file `prepared.text' in which we placed a message for `bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the _ c_ a_ t command would then have written the text through the pipe to the standard input of the mail command. When the _ c_ a_ t command completed it would have terminated, closing down the pipeline and the _ m_ a_ i_ l command would have received an end-of-file from it and terminated. Using a

---

*Many users use _ s_ t_ t_ y(1) to change the interrupt character to | ^C.

pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a | ^Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types `Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the _ f_ g command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

_ 2. _ D_ e_ t_ a_ i_ l_ s _ o_ n _ t_ h_ e _ s_ h_ e_ l_ l _ f_ o_ r _ t_ e_ r_ m_ i_ n_ a_ l _ u_ s_ e_ r_ s

2. 1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset -s -m dialup:c100rv4pna -m
plugboard:?hp2621nl *`';
ts; stty intr | ^C kill | ^U crt
set time=15 history=10
msgs -f
if (-e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit | ^D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set*; this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable `time' to `15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable `history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history* list, (described later).

I create an _ a_ l_ i_ a_ s ``ts'' which executes a _ t_ s_ e_ t(1) command setting up the modes of the terminal. The parameters to _ t_ s_ e_ t indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute ``ts'' and also use the _ s_ t_ t_ y command to change the interrupt character to | ^C and the line kill character to | ^U.

I then run the `msgs' program, which provides me with any system messages which I have not seen before; the `-f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the `mail' program to process my mail.

When the `mail' and `msgs' programs finish, the shell will finish processing my ._ l_ o_ g_ i_ n file and begin reading commands from the terminal, prompting for each with `% '. When I log off (by giving the _ l_ o_ g_ o_ u_ t command) the shell will print `logout' and execute commands from the file `.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the `logout' message the shell is committed to terminating and will take no further input from my terminal.

_ 2._ 2. _ S_ h_ e_ l_ l _ v_ a_ r_ i_ a_ b_ l_ e_ s

The shell maintains a set of _ v_ a_ r_ i_ a_ b_ l_ e_ s. We saw above the variables _ h_ i_ s_ t_ o_ r_ y and _ t_ i_ m_ e which had values `10' and `15'. In fact, each shell variable has as value an array of zero or more _ s_ t_ r_ i_ n_ g_ s. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable _ p_ a_ t_ h. This variable contains a sequence of directory names

where the shell searches for commands. The _ s_ e_ t command with no arguments shows the value of all variables currently defined (we usually say _ s_ e_ t) in the shell. The default value for path will be shown by _ s_ e_ t to be

_ 3. _ S_ h_ e_ l_ l _ c_ o_ n_ t_ r_ o_ l _ s_ t_ r_ u_ c_ t_ u_ r_ e_ s _ a_ n_ d _ c_ o_ m_ m_ a_ n_ d _ s_ c_ r_ i_ p_ t_ s

_ 3._ 1. _ I_ n_ t_ r_ o_ d_ u_ c_ t_ i_ o_ n

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called _ s_ h_ e_ l_ l _ s_ c_ r_ i_ p_ t_ s. We here detail those features of the shell useful to the writers of such scripts.

_ 3._ 2. _ M_ a_ k_ e

It is important to first note what shell scripts are _ n_ o_ t useful for. There is a program called _ m_ a_ k_ e which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a _ m_ a_ k_ e_ f_ i_ l_ e which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this _ m_ a_ k_ e_ f_ i_ l_ e. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a _ m_ a_ k_ e_ f_ i_ l_ e may be created which defines how different versions of the document are to be created and which options of _ n_ r_ o_ f_ f or _ t_ r_ o_ f_ f are appropriate.

_ 3._ 3. _ I_ n_ v_ o_ c_ a_ t_ i_ o_ n _ a_ n_ d _ t_ h_ e _ a_ r_ g_ v _ v_ a_ r_ i_ a_ b_ l_ e

A _ c_ s_ h command script may be interpreted by saying

```
% csh script ...
```

where _ s_ c_ r_ i_ p_ t is the name of the file containing a group of _ c_ s_ h commands and `...' is replaced by a sequence of arguments. The shell places these arguments in the variable _ a_ r_ g_ v and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file `script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a `#' character) then a `/bin/csh' will automatically be invoked to execute `script' when you type

```
script
```


If the file does not begin with a `#' then the standard shell `/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use _ c_ s_ h at your convenience.

_ 3._ 4. _ V_ a_ r_ i_ a_ b_ l_ e _ s_ u_ b_ s_ t_ i_ t_ u_ t_ i_ o_ n

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as _ v_ a_ r_ i_ a_ b_ l_ e _ s_ u_ b_ s_ t_ i_ t_ u_ t_ i_ o_ n is done on these words. Keyed by the character `$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable _ a_ r_ g_ v to be echoed to the output of the shell script. It is an error for _ a_ r_ g_ v to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
        $?name
```

expands to `1' if name is _ s_ e_ t or to `0' if name is not _ s_ e_ t. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
        $#name
```

expands to the number of elements in the variable _ n_ a_ m_ e. Thus

```
        % set argv=(a b c)
        % echo $?argv
        1
        % echo $#argv
        3
        % unset argv
        % echo $?argv
        0
        % echo $argv
        Undefined variable: argv.
        %
```

It is also possible to access the components of a variable which has several values. Thus


```
        $argv[1]
```

gives the first component of _ a_ r_ g_ v or in the example above `a'. Similarly

```
        $argv[$#argv]
```

would give `c', and

```
        $argv[1-2]
```

would give `a b'. Other notations useful in shell scripts are

```
        $_ n
```

where _ n is an integer as a shorthand for

```
        $argv[_ n]
```

the _ n_ t_ h parameter and

```
        $*
```

which is a shorthand for

```
        $argv
```

The form

```
        $$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
        $<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
        echo 'yes or no?\c'
        set a=($<)
```

would write out the prompt `yes or no?' without a newline and then read the answer into the variable `a'. In this case `$#a' would be `0' if either a blank line or end-of-file (| ^D) was typed.

One minor difference between `$_ n' and `$argv[_ n]' should

be noted here. The form `$argv[_ n]' will yield an error if _ n is not in the range `1-$#argv' while `$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form `n-'; if there are less than _ n components of the given variable then no words are substituted. A range of the form `m-n' likewise returns an empty vector without giving an error when _ m exceeds the number of elements of the given variable, provided the subscript _ n is in range.

_ 3._ 5. _ E_ x_ p_ r_ e_ s_ s_ i_ o_ n_ s

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations `==' and `!=' compare strings and the operators `&&' and `||' implement the boolean and/or operations. The special operators `=~' and `!~' are similar to `==' and `!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
-? filename
```

where `?' is replace by a number of single characters. For instance the expression primitive

```
-e filename
```

tell whether the file `filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form `{ command }' which returns true, i.e. `1' if the command succeeds exiting normally with exit status 0, or `0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable `$status' examined in the next command. Since `$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see

the manual section for the shell.

_ 3._ 6. _ S_ a_ m_ p_ l_ e _ s_ h_ e_ l_ l _ s_ c_ r_ i_ p_ t

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the _ f_ o_ r_ e_ a_ c_ h command, which causes the shell to execute the commands between the _ f_ o_ r_ e_ a_ c_ h and the matching _ e_ n_ d for each of the values given between `(' and `)' with the named variable, in this case `i' set to successive values in the list. Within this loop we may use the command _ b_ r_ e_ a_ k to stop executing the loop and _ c_ o_ n_ t_ i_ n_ u_ e to prematurely terminate one iteration and begin the next. After the _ f_ o_ r_ e_ a_ c_ h loop the iteration variable (_ i in this case) has the value at the last iteration.

We set the variable _ n_ o_ g_ l_ o_ b here to prevent filename expansion of the members of _ a_ r_ g_ v. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a `$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of

the form


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

The placement of the keywords here is _ n_ o_ t flexible due to the current implementation of the shell.

The shell does have another form of the if statement of the form

```
        if ( expression ) command
```

which can be written

```
        if ( expression ) \
                command
```

Here we have escaped the newline for the sake of appearance. The command must not involve `|', `&' or `;' and must not be another control command. The second form requires the final `\' to _ i_ m_ m_ e_ d_ i_ a_ t_ e_ l_ y precede the end-of-line.

The more general _ i_ f statements above also admit a sequence of _ e_ l_ s_ e-_ i_ f pairs followed by a single _ e_ l_ s_ e and an _ e_ n_ d_ i_ f, e.g.:

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

        else
                commands
        endif
```

Another important mechanism used in shell scripts is the `:' modifier. We can use the modifier `:r' here to

---

The following two formats are not currently acceptable to the shell:

```
     if ( expression )              # Won't work!
     then
             command
             ...
     endif
```

and

```
     if ( expression ) then command endif            # Won't work
```


extract a root of a filename or `:e' to extract the _ e_ x_ t_ e_ n_ - _ s_ i_ o_ n. Thus if the variable _ i has the value `/mnt/foo.bar' then

```
     % echo $i $i:r $i:e
     /mnt/foo.bar /mnt/foo bar
     %
```

shows how the `:r' modifier strips off the trailing `.bar' and the the `:e' modifier leaves only the `bar'. Other modifiers will take off the last component of a pathname leaving the head `:h' or all but the last component of a pathname leaving the tail `:t'. These modifiers are fully described in the _ c_ s_ h manual pages in the programmers manual. It is also possible to use the _ c_ o_ m_ m_ a_ n_ d _ s_ u_ b_ s_ t_ i_ t_ u_ t_ i_ o_ n mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the `:' modification mechanism.# Finally, we note that the character `#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a `#' are discarded by the shell. This character can be quoted using `'' or `\' to place it in an argument word.

_ 3._ 7. _ O_ t_ h_ e_ r _ c_ o_ n_ t_ r_ o_ l _ s_ t_ r_ u_ c_ t_ u_ r_ e_ s

The shell also has control structures _ w_ h_ i_ l_ e and _ s_ w_ i_ t_ c_ h similar to those of C. These take the forms

```
        while ( expression )
```

```
                commands
        end
```

and

#It is also important to note that the current imple-
mentation of the shell limits the number of `:' modif-
iers on a `$' substitution to 1. Thus

```
     % echo $i $i:h:t
     /a/b/c /a/b:t
     %
```

does not do what one would expect.


```
        switch ( word )

        case str1:
                commands
                breaksw

         ...

        case strn:
                commands
                breaksw

        default:
                commands
                breaksw

        endsw
```

For details see the manual section for _ c_ s_ h. C programmers
should note that we use _ b_ r_ e_ a_ k_ s_ w to exit from a _ s_ w_ i_ t_ c_ h
while
_ b_ r_ e_ a_ k exits a _ w_ h_ i_ l_ e or _ f_ o_ r_ e_ a_ c_ h loop. A
common mistake to

make in _ c_ s_ h scripts is to use _ b_ r_ e_ a_ k rather than _ b_ r_ e_ a_ k_ s_ w in switches.

Finally, _ c_ s_ h allows a _ g_ o_ t_ o statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

_ 3._ 8. _ S_ u_ p_ p_ l_ y_ i_ n_ g _ i_ n_ p_ u_ t _ t_ o _ c_ o_ m_ m_ a_ n_ d_ s

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file


```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/| ^[ ]*//
w
q
'EOF'
end
%
```

The notation `<< 'EOF'' means that the standard input for the _ e_ d command is to come from the text in the shell script file up to the next line consisting of exactly `'EOF''. The fact that the `EOF' is enclosed in `'' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the `<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form `1,$' in our editor script we needed to insure that this `$' was not variable substituted. We could also have insured this by preceding the `$' here with a `\', i.e.:

```
1,\$s/| ^[ ]*//
```

but quoting the `EOF' terminator is a more reliable way of achieving the same thing.

_ 3._ 9. _ C_ a_ t_ c_ h_ i_ n_ g _ i_ n_ t_ e_ r_ r_ u_ p_ t_ s

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where _ l_ a_ b_ e_ l is a label in our program. If an interrupt is received the shell will do a `goto label' and we can remove the temporary files and then do an _ e_ x_ i_ t command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status `1'.

_ 3._ 1_ 0. _ W_ h_ a_ t _ e_ l_ s_ e?

There are other features of the shell useful to writers of shell procedures. The _ v_ e_ r_ b_ o_ s_ e and _ e_ c_ h_ o options and the related -_ v and -_ x command line options can be used to help trace the actions of the shell. The -_ n option causes the

shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that _ c_ s_ h will not execute shell scripts which do not begin with the character `#', that is shell scripts that do not begin with a comment. Similarly, the `/bin/sh' on your system may well defer to `csh' to interpret shell scripts which begin with `#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using `"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as `'' does.

_ 4. _ O_ t_ h_ e_ r, _ l_ e_ s_ s _ c_ o_ m_ m_ o_ n_ l_ y _ u_ s_ e_ d, _ s_ h_ e_ l_ l _ f_ e_ a_ t_ u_ r_ e_ s

_ 4._ 1. _ L_ o_ o_ p_ s _ a_ t _ t_ h_ e _ t_ e_ r_ m_ i_ n_ a_ l; _ v_ a_ r_ i_ a_ b_ l_ e_ s _ a_ s _ v_ e_ c_ t_ o_ r_ s

It is occasionally useful to use the _ f_ o_ r_ e_ a_ c_ h control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, `/bin/sh', `/bin/nsh', and `/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use _ f_ o_ r_ e_ a_ c_ h to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with `? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

```
%
```

The _ s_ e_ t command here gave the variable _ a a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within `` ' characters is converted by the shell to a list of words. You can also place the `` ' quoted string within `"' characters to take each

(non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier `:x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

_ 4._ 2. _ B_ r_ a_ c_ e_ s { ... } _ i_ n _ a_ r_ g_ u_ m_ e_ n_ t _ e_ x_ p_ a_ n_ s_ i_ o_ n

Another form of filename expansion, alluded to before involves the characters `{' and `}'. These characters specify that the contained strings, separated by `,' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories `hdrs', `retrofit' and `csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.3. Command substitution

A command enclosed in `` characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable pwd or to do

```
ex `grep -l TRACE *.c`
```

to run the editor ex supplying as arguments those files whose names end in `.c' which have the string `TRACE' in them.*

---

*Command expansion also occurs in input redirected with

4.4. Other differences in command processing

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

`<<' and within `"' quotations. Refer to the shell manual section for full details.

_ A_ p_ p_ e_ n_ d_ i_ x - _ S_ p_ e_ c_ i_ a_ l _ c_ h_ a_ r_ a_ c_ t_ e_ r_ s

The following table lists the special characters of _ c_ s_ h and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the _ c_ s_ h manual section for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
|	1.5	separates commands in a pipeline
( )	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.5	indicates redirected input
>	1.3	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
| ^	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	1.3,3.6	begins scratch file names; indicates shell comments
-	1.2	prefixes option (flag) arguments to commands
%	2.6	prefixes job