An Introduction to the UNIX Shell

S. R. Bourne

Murray Hill, NJ

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

The *shell*‡ is a command programming language that provides an interface to the UNIX® operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while, if then else, case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, "UNIX for beginners". unix beginn kernigh 1978 Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual. seventh 1978 ritchie thompson

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
        who
```

is a command that prints the names of users logged in. The command

```
        ls −l
```

prints a list of files in the current directory. The argument *−l* tells *ls* to print status information, size and the creation date for each file.

‡ This paper describes sh(1). If it's the c shell (csh) you're interested in, a good place to begin is William Joy's paper "An Introduction to the C shell" (USD:4).

1.2 Background commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing **&** is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls –l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls.* If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls.* Output may be appended to a file using the notation

```
ls –l file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

```
wc –l <file
```

could be used.

1.4 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by , as in,

```
ls –l  wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls –l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep,* selects from its input those lines that contain some specified string. For example,

```
ls  grep old
```

prints those lines, if any, of the output from *ls* that contain the string *old.* Another useful filter is *sort*. For example,

```
who  sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls  grep old  wc –l
```

prints the number of file names in the current directory containing the string *old.*

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls –l main.c
```

prints information relating to the file *main.c* .

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls –l *.c
```

generates, as arguments to *ls,* all file names in the current directory that end in *.c* . The character * is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

*****	Matches any string of characters including the null string.
?	Matches any single character.
[]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a–z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z.*

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of **/usr/fred .** (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of **/usr/fred .**

There is one exception to the general rules given for patterns. The character '**.**' at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with '**.**' .

```
echo .*
```

will echo all those file names that begin with '**.**' . This avoids inadvertent matching of the names '**.**' and '**..**' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '**.**' and '**..**' .)

1.6 Quoting

Characters that have a special meaning to the shell, such as **< > * ? &,** are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \\?
```

will echo a single **? ,** and

```
echo \\\\
```

will echo a single **\\ .** To allow long strings to be continued over more than one line the sequence **\\newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx´****´xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4 .

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '**$** ' . It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '**>** ' . Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
  Print the names of files in the current directory.
- **ls >file**
  Put the output from *ls* into *file.*
- **ls wc –l**
  Print the number of files in the current directory.
- **ls grep old**
  Print those file names containing the string *old.*
- **ls grep old wc –l**
  Print the number of files whose name contains the string *old.*
- **cc pgm.c &**
  Run *cc* in the background.