UNIX Shell Tutorial

G. A. Snyder
J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

In any programming project, some effort is used to build the end product. The remainder is consumed in building the supporting tools and procedures used to manage and maintain that end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool in such situations. If it is a flexible programming language, it can be used to solve many internal support problems without requiring compilable programs to be written, debugged, and maintained; its most important advantage is the ability to get the job done *now*. For a perspective on the motivations for using a command language in this way, see [1,4,5,6].

When users log into a UNIX† system, they communicate with an instance of the shell that reads commands typed at the terminal and arranges for their execution. Thus, the shell's most important function is to provide a good interface for human beings. In addition, a sequence of commands may be preserved for repeated use by saving it in a file, called a *shell procedure*, a *command file*, or a *runcom*, according to local preference.

Some UNIX users need little knowledge of the shell to do their work; others make heavy use of its programming features. This tutorial may be read in several different ways, depending on the reader's interests. A brief discussion of the UNIX environment is found in §2. The discussion in §3 covers aspects of the shell that are important for everyone, while all of §4 and most of §5 are mainly of interest to those who write shell procedures. A group of annotated shell procedure examples is given in §6. Finally, a brief discussion of efficiency is offered in §7; this is found in its proper place (at the end), and is intended for those who write especially time-consuming shell procedures.

Complete beginners should *not* be reading this tutorial, but should work their way through other available tutorials first. See [14] for an appropriate plan of study. All the *commands* mentioned below are described in Section 1 of the *UNIX User's Manual* [7], while *system calls* are described in Section 2 and *subroutines* in Section 3 thereof; references of the form *name*(*N*) point to entry *name* in Section *N* of that manual.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of what follows depends on familiarity with UNIX; [13] is useful for that, and it would be helpful to read [8] and at least one of [9,10]. For completeness, a short overview of the most relevant concepts is given below.

2.1 File System

The UNIX file system's overall structure is that of a rooted tree composed of *directories* and other files. A simple *file name* is a sequence of characters other than a slash (/). A *path name* is a sequence of directory names followed by a simple file name, each separated from the previous one by a /. If a path name begins with a /, the search for the file begins at the *root* of the entire tree; otherwise, it begins at the user's *current directory* (also known as the *working directory*). The first kind of name is often called a *full* (or *absolute*) *path name* because it is invariant

† UNIX is a trademark of Bell Laboratories.

with regard to the user's current directory. The latter is often called a *relative path name*, because it specifies a path relative to the current directory. The user may change the current directory at any time by using the `cd` command. In most cases, a file name and its corresponding path name may be used interchangeably. Some sample names are:

`/`	absolute path name of the root directory of the entire file structure.
`/bin`	directory containing most of the frequently used public commands.
`/a1/tf/jtb/bin`	a full path name typical of multi-person programming projects. This one happens to be a private directory of commands belonging to person `jtb` in project `tf`; `a1` is the name of a *file system*.
`bin/x`	a relative path name; it names file `x` in subdirectory `bin` of the current directory. If the current directory is `/`, it names `/bin/x`. If, on the other hand, the current directory is `/a1/tf/jtb`, it names `/a1/tf/jtb/bin/x`.
`memox`	name of a file in the current directory.

The UNIX file system provides special shorthand notations for the current directory and the *parent* directory of the current directory:

`.` is the generic name of the current directory; `./memox` names the same file as `memox` if such a file exists in the current directory.

`..` is the generic name of the parent directory of the current directory; if you type:

```
cd ..
```

then the parent directory of your current working directory will become your new current directory.

2.2 UNIX Processes

☞ *Beginners should skip this section on first reading.*

An *image* is a computer execution environment, including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image; most UNIX commands execute as separate processes. One process may spawn another using the `fork` system call, which duplicates the image of the original (*parent*) process. The new (*child*) process continues to execute the same program as the parent. The two images are identical, except that each program can determine whether it is executing as parent or child. Each program may continue execution of the image or may abandon it by issuing an `exec` system call, thus initiating execution of another program. In any case, each process is free to proceed in parallel with the other, although the parent most commonly issues a `wait` system call to suspend execution until a child terminates (`exits`).

Figure 1 illustrates these ideas. *Program A* is executing (as *process 1*) and wishes to run *program B*. It `fork`s and spawns a child (*process 2*) that continues to run *program A*. The child abandons *A* by `exec`ing *B*, while the parent goes to sleep until the child `exits`.

A child inherits its parent's *open files*. This mechanism permits processes to share common input streams in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations on the file; `read` and `write` system calls copy a requested number of bytes from and to a file, beginning at the position given by the current value of the pointer. As a side effect, the pointer is incremented by the number of bytes transferred, yielding the effect of sequential I/O; `lseek` can be used to obtain random-access I/O; it sets the pointer to an absolute position within the file, or to a position offset either from the end of the file or from the current pointer position.

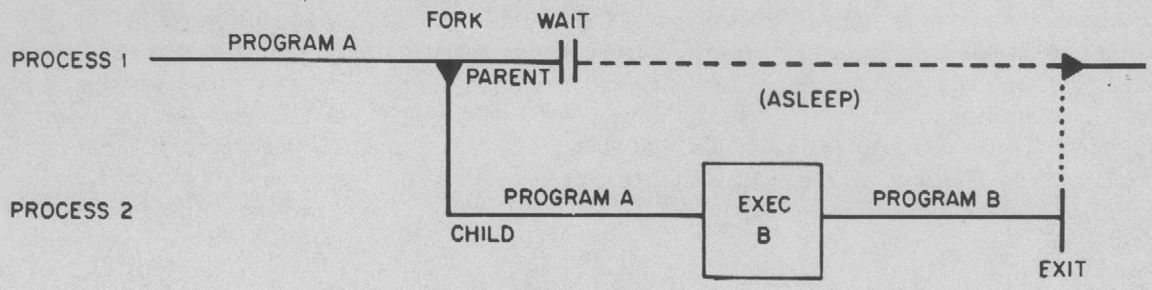

Figure 1

When a process terminates, it can set an eight-bit *exit status* (see `$?` in §3.4.4) that is available to its parent. This code is *usually* used to indicate success (zero) or failure (non-zero).

Signals indicate the occurrence of events that may have some impact on a process. A signal may be sent to a process by another process, from the terminal, or by UNIX itself. A child process inherits its parent's signals. For most signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to *catch* it and take appropriate action, as described in §4.4.11. For example, an `INTERRUPT` signal may be sent by depressing an appropriate key (*del*, *break*, or *rubout*). The action taken depends on the requirements of the specific program being executed:

- The shell invokes most commands in such a way that they immediately die when an interrupt is received. For example, the `pr` (print) command normally dies, allowing the user to terminate unwanted output.
- The shell *itself* ignores interrupts when reading from the terminal, because it should continue execution even when the user terminates a command like `pr`.
- The editor `ed` chooses to *catch* interrupts so that it can halt its current action (especially printing) without allowing itself to be terminated.

3. SHELL BASICS

The shell (i.e., the `sh` command) implements the command language visible to most UNIX users. It reads input from a terminal or a file and arranges for the execution of the requested commands. It is a program written in the C language [11]; it is *not* part of the operating system, but is an ordinary user program. The discussion below is adapted from [2,3,7,12].

3.1 Commands

A *simple command* is a sequence of non-blank arguments separated by blanks or tabs. The first argument (numbered *zero*) usually specifies the name of the command to be executed; any remaining arguments, with a few exceptions, are passed as arguments to that command. A command may be as simple as:

```
who
```

which prints the login names of users who are currently logged into the system. The following line requests the `pr` command to print files `a`, `b`, and `c`:

```
pr a b c
```

If the first argument of a command names a file that is *executable*[1] and is actually a compiled program, the shell (as parent) spawns a new (child) process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines, as well as possibly lines meant to be read by other programs. In this case, the shell spawns another instance of itself (a *sub-shell*) to read the file and execute the commands included in it. The shell `fork`s to do this, but no *exec* call is made. The following command requests that the on-line *UNIX User's Manual* [7] entries that describe the `who` and `pr` commands be printed on the terminal:

```
    man who pr
```

(Incidentally, the `man` command itself is actually implemented as a shell procedure.) From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This preserves the uniformity of invocation and the ease of changing the choice of implementation for a given command. The actions of the shell in executing any of these commands are illustrated in Figure 1 above.

3.2 How the Shell Finds Commands

The shell normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. The shell first attempts to find the command (as given on the command line) in the current directory; if this fails, it prepends the string `/bin` to the name, and, finally, `/usr/bin`. The effect is to search, in order, the current directory, then the directory `/bin`, and finally, `/usr/bin`. For example, the `pr` and `man` commands are actually the files `/bin/pr` and `/usr/bin/man`, respectively. A more complex path name may be given, either to locate a file relative to the user's current directory, or to access a command via an absolute path name. If a command name *as given* begins with a `/`, `./`, or `../` (e.g., `/bin/sort` or `../cmd`), the prepending is *not* performed. Instead, a single attempt is made to execute the command as given.

This mechanism gives the user a convenient way to execute public commands and commands in or *near* the current directory, as well as the ability to execute *any* accessible command regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command will not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the `PATH` variable, as described in §3.4.2.

3.3 Generation of Argument Lists

Command arguments are very often file names. A list of file names can be automatically generated as arguments on a command line, by specifying a pattern that the shell matches against the file names in a directory.

Most characters in such a pattern match themselves, but there are also special *meta-characters* that may be included in a pattern. These special characters are: `*`, which matches any string *including* the null string; `?`, which matches *any one* character; any sequence of characters enclosed within square brackets[2] (`[...]`), which matches *any one* of the enclosed characters; and any sequence of characters preceded by a `!` *and* enclosed within `[...]`, which matches

1. As indicated by an appropriate set of permission bits associated with that file.

2. Be warned that square brackets are also used below for another purpose: in descriptions of commands, they indicate that the enclosed argument is optional. See also §5.1 below.

any one character *other* than one of the enclosed characters. Inside square brackets, a pair of characters separated by a - includes in the set all characters lexically within the inclusive range of that pair, so that `[a-de]` is equivalent to `[abcde]`.

For example, `*` matches all file names in the current directory, `*temp*` matches all file names containing `temp`, `[a-f]*` matches all file names that begin with `a` through `f`, `[!0-9]` matches all single-character names other than the digits, and `*.c` matches all file names ending in `.c`, while `/a1/tf/bin/?` matches all single-character file names found in `/a1/tf/bin`. This capability saves much typing and, more importantly, makes it possible to organize information in large collections of small files that are named in disciplined ways.

Pattern-matching has some restrictions. If the first character of a file name is a period (`.`), it can be matched only by an argument that literally begins with a period. If a pattern does not match any file names, then the pattern itself is returned as the result of the match, for example:

```
echo *.c
```

will print:

```
*.c
```

if the current directory contains no files ending in `.c`.

Directory names should not contain the characters `*`, `?`, `[`, or `]`, because this may cause infinite recursion during pattern matching attempts.[3]

3.4 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*; these are the variables which are normally set only on a command line; there are *positional parameters* (§3.4.1) and *keyword parameters* (§4.1). Other variables are simply names to which the user or the shell itself may assign string values.

3.4.1 Positional Parameters. When a shell procedure is invoked, the shell implicitly creates *positional parameters*: the argument in position zero on the command line (the name of the shell procedure itself) is called `$0`, the first argument is called `$1`, and so on. The `shift` command (§4.3) may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the `set` command:

```
set abc def ghi
```

assigns the string `abc` to the first positional parameter (`$1`), `def` to the second (`$2`), and `ghi` to the third (`$3`); it also *unsets* `$4`, `$5`, etc., even if they were previously set. `$0` may not be assigned a value in this way—it always refers to the name of the shell procedure, or, in the login shell, to the name of the shell.

3.4.2 User-defined Variables. The shell also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement; they can only be set as described above. A simple assignment is of the form:

name=string

3. This is a bug that may be fixed in the future.

Thereafter, **$***name* will yield the value *string*. A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left;* the following command line results in the variable `a` acquiring the value `abc`:

```
a=$b b=abc
```

The following are examples of simple assignments. *Double* quotes around *the right-hand side* allow blanks, tabs, semi-colons, and new-lines to be included in *string*, while also allowing *variable substitution* (also known as *parameter substitution*) to occur; that is, references to positional parameters and other variable names that are prefaced by **$** are replaced by the corresponding values, if any; *single* quotes inhibit variable substitution:

```
MAIL=/usr/mail/gas
var="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable `var` has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to `stars` because pattern matching (expansion of `*`, `?`, `[...]`) does *not* apply in this context. Note that the value of `$asterisks` is the literal string `$stars`, *not* the string `*****`, because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in `$first` and `$second` having the same value:

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces `{}` to delimit the variable name from any following string.[4] In particular, if the character immediately following the name is a letter, digit, or underscore (digit only for positional parameters), then the braces are *required:*

```
a='This is a string'
echo "${a}ent test"
```

The following variables are used by the shell. Some of them are set by the shell, and all of them can be set and reset by the user:

`HOME` is initialized by the `login` program to the name of the user's *login directory*, i.e., the directory that becomes the current directory upon completion of a login; `cd` without arguments uses `$HOME` as the directory to switch to. Using this variable helps one to keep full path names out of shell procedures. This is a big help when the path name of your login directory is changed (e.g., to balance disk loads).

`MAIL` is the path name of a file where your mail is deposited. If `MAIL` is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail every time you return to command level (e.g., by leaving the editor). `MAIL` must be set by the user. (The presence of mail in the standard mail file is also announced at login, regardless of whether `MAIL` is set.)

4. See §4.4.7 and § 5.7 for other meanings of braces in the shell.

`PATH` is the variable that specifies where the shell is to look when it is searching for commands. Its value is an ordered list of directory path names separated by colons. A null character anywhere in that list represents the current directory. The shell initializes `PATH` to the list `:/bin:/usr/bin` where, by convention, a null character appears in front of the first colon. Thus if you wish to search your current directory last, rather than first, you would type:

```
PATH=/bin:/usr/bin::
```

where the two colons together represent a colon followed by a null followed by a colon, thus naming the current directory. A user often has a personal directory of commands (say, `$HOME/bin`) and causes it to be searched *before* the `/bin` and `/usr/bin` directories by using:

```
PATH=:$HOME/bin:/bin:/usr/bin
```

The setting of `PATH` to other than the default value is normally done in a user's `.profile` file (§3.9.2).

`CDPATH` is the variable that specifies where the shell is to look when searching for the argument of the `cd` command whenever that argument is not null and does not begin with `/`, `./`, or `../` (see *cd*(1), §2.1, and §4.5). The value of `CDPATH` is an ordered list of directory path names separated by colons. A null character anywhere in that list represents the current directory. By convention, if the list begins with a colon, a null character is assumed to precede that colon. Initially, `CDPATH` is *unset*, resulting in only the current directory being searched. Thus if you wish the `cd` command to first search your current directory and then your home directory, you would type:

```
CDPATH=:$HOME
```

The setting of `CDPATH` to other than the default value is normally done in a user's `.profile` file (§3.9.2).

Note that if the `cd` command changes to a directory that is *not* a descendent of the current directory, it writes the full name of the new directory on the diagnostic output(§3.6.1, §3.6.2).

`PS1` is the variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of `PS1` when it expects input. The default value of `PS1` is "`$ `" (a `$` followed by a blank).

`PS2` is the variable that specifies the secondary prompt string. If the shell expects more input when it encounters a new-line in its input, it will prompt with the value of `PS2`. The default value of `PS2` is "`> `" (a > followed by a blank).

`IFS` is the variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set `IFS` to include that delimiter.) The shell initially sets `IFS` to include the blank, tab, and new-line characters.

3.4.3 Command Substitution. Any command line can be placed within grave accents (`` `...` ``) to capture the output of the command. This concept is known as *command substitution*. The command or commands enclosed between grave accents are first executed by the shell and then their output replaces the whole expression, grave accents and all. This feature is often combined with shell variables:

```
today=`date`
```

assigns the string representing the current date to the variable `today` (e.g., `Tue Nov 27 16:01:09 EST 1979`).

```
users=`who | wc -l`
```

saves the number of logged-in users in the variable `users`. Any command that writes to the standard output can be enclosed in grave accents. Grave accents (§3.5) may be nested; the inside sets must be escaped with \. For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

Shell variables can also be given values indirectly by using the `read` command. The `read` command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named:

```
read first init last
```

will take an input line of the form:

```
G. A. Snyder
```

and have the same effect as if you had typed:

```
first=G.   init=A.   last=Snyder
```

The `read` command assigns any excess "words" to the last variable.

3.4.4 Predefined Special Variables.

☞ *Beginners should skip this section on first reading.*

Several variables have special meanings; the following are set *only* by the shell:

$# records the number of *positional* arguments passed to the shell, not counting the name of the shell procedure itself; **$#** thus yields the number of the highest-numbered positional parameter that is set. Thus, `sh x a b c` sets **$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$?** is the exit status (also referred to as *return code, exit code,* or *value*) of the last command executed. Its value is a decimal string. Most UNIX commands return 0 to indicate successful completion. The shell itself returns the current value of **$? as *its* exit status.

$$ is the process number of the current process; because process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names for temporary files. UNIX provides no mechanism for the automatic creation and deletion of temporary files: a file exists until it is explicitly removed. Temporary files are generally undesirable objects: the UNIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
temp=$HOME/temp/$$          # use current process number
ls > $temp                  # to form unique temp file
    commands, some of which use $temp, go here
rm $temp                    # clean up at end
```

$! is the process number of the last process run in the background (using &—see §4.4). Again, this is a string of up to five digits.

$-** is a string consisting of names of execution flags (§3.9.3, §4.7) currently turned on in the shell; **$- might have the value `xv` if you are tracing your output.

3.5 Quoting Mechanisms

Many characters have a special meaning to the shell which is sometimes necessary to conceal. Single quotes (′′) and double quotes ("") surrounding a string, or backslash (\) before a single character, provide this function in somewhat different ways. (Grave accents (\`\`) are sometimes called *back quotes*, but are used only for command substitution (§3.4.3) in the shell and do not hide special meanings of any characters.)

Within single quotes, all characters (except ′ itself) are taken literally, with any special meaning removed. Thus:

```
   stuff='echo $? $*; ls * | wc'
```

results only in the string `echo $? $*; ls * | wc` being assigned to the variable `stuff`, but *not* in any other commands being executed.

Within double quotes, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are **$**, \`, and " itself. Thus, within double quotes, variables are expanded and command substitution takes place; however, any commands in a command substitution are not affected by double quotes outside of the grave accents, so that characters such as * retain their special meaning.

To hide the special meaning of **$**, \`, and " within double quotes, you can precede these characters with a backslash (\). Outside of double quotes, preceding a character with \ is equivalent to placing single quotes around that character. A \ followed by a new-line causes that new-line to be ignored, thus allowing continuation of long command lines.

3.6 Redirection of Input and Output

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline (see §3.7). A few commands vary their actions depending on the nature of their input or output, either for efficiency's sake, or to avoid useless actions (such as attempting random-access I/O on a terminal).

3.6.1 Standard Input and Standard Output. When a command begins execution, it usually expects that three files are already open: a *standard input*, a *standard output*, and a *diagnostic (error) output*. A number called a *file descriptor* is associated with each of these files; by convention, file descriptor 0 is associated with standard input, file descriptor 1 with standard output, and file descriptor 2 with diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the printer or screen). The shell permits them to be redirected elsewhere before control is passed to an invoked command. An argument to the shell of the form < *file* or > *file* opens the specified file as the standard input or output, respectively (in the case of output, destroying the previous contents of *file*, if any). An argument of the form >> *file* directs the standard output to the end of *file*, thus providing a way to *append* data to it without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist (thus `> output` alone on a line creates a zero-length file). The following appends to file `log` the list of users who are currently logged on:

```
   who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of file names occurs after these substitutions. Thus:

```
   echo 'this is a test' > *.ggg
```

and:

```
   cat < ?
```

will produce, respectively, a one-line file named `*.ggg` (a rather disastrous name for a file) and an error message (unless you have a file named `?`, which is also *not* a wise choice for a file name—see end of §3.3).

3.6.2 Diagnostic and Other Outputs. Diagnostic output from UNIX commands is traditionally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines—see §3.7.) One can redirect this error output to a file by immediately prepending the number of the file descriptor (i.e., 2 in this case) to either output redirection symbol (`>` or `>>`). The following line will append error messages from the `cc` command to file `ERRORS`:

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening blanks or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow one to redirect output associated with any of the first ten file descriptors (numbered `0-9`) so that, for instance, if `cmd` puts output on file descriptor `9`, the following line will capture that output in file `savedata`:

```
cmd 9> savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose that `cmd` directs its standard output to file descriptor `1`, its error output to file descriptor `2`, and builds a data file on file descriptor `9`. The following would direct each of these three outputs to a different file:

```
cmd > standard   2> error   9> data
```

Other forms of input/output redirection are described in §4.4.8, §4.4.9, and §5.6.

3.7 Command Lines and Pipelines

A sequence of one or more commands separated by `|` (or `^`) make up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbor(s) by *pipes*, i.e., the *output* of each command (except the last one) becomes the *input* of the next command in line. A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read larger amounts of data before producing output; `sort` is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline: `nroff` is a text formatter whose output may contain reverse line motions; `col` converts these motions to a form that can be printed on a terminal lacking reverse-motion capability; `greek` is used to adapt the output to a specific terminal, here specified by `-Thp`. The flag `-cm` indicates one of the commonly used formatting options, and `text` is the name of the file to be formatted:

```
nroff -cm text | col | greek -Thp
```

3.8 Examples

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

- `who`
  Print (on the terminal) the list of logged-in users.
- `who >> log`
  Append the list of logged-in users to the end of file `log`.
- `who | wc -l`
  Print the number of logged-in users. (The argument to `wc` is *minus ell*.)
- `who | pr`
  Print a paginated list of logged-in users.
- `who | sort`
  Print an alphabetized list of logged-in users.
- `who | grep pw`
  Print the list of logged-in users whose login names contain the string `pw`.
- `who | grep pw | sort | pr`
  Print an alphabetized, paginated list of logged-in users whose login names contain the string `pw`.
- `{ date; who | wc -l; } >> log`
  Append (to file `log`) the current date followed by the count of logged-in users (see §4.4.7 for the meaning of `{ . . . }` in this context).
- `who | sed 's/ .*//' | sort | uniq -d`
  Print only the login names of all users who are logged in more than once.

The `who` command does not *by itself* provide options to yield all these results—they are obtained by combining `who` with other commands. Note that `who` just serves as the data source in these examples. As an exercise, replace `who |` by `< /etc/passwd` in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line.

3.9 Changing the State of the Shell and the `.profile` File

The state of a given instance of the shell includes the values of positional parameters (§3.4.1), user-defined variables (§3.4.2), environment variables (§4.1), modes of execution (§4.7), and the current working directory.

The state of a shell may be altered in various ways. These include the `cd` command, several flags that can be set by the user, and a file in one's login directory called `.profile` that is treated specially by the shell.

3.9.1 `Cd`. The `cd` command changes the current directory to the one specified as its argument. This can (and should) be used to change to a convenient place in the directory structure; `cd` is often combined with `( )` to cause a sub-shell to change to a different directory and execute a group of commands without affecting the original shell. The first sequence below extracts the component files of the archive file `/a1/tf/q.a` and places them in whatever directory is the current one; the second places them in directory `/a1/tf`:

```
ar x /a1/tf/q.a
(cd /a1/tf; ar x q.a)
```

3.9.2 The `.profile` File. When you log in, the shell is invoked to read your commands. First, however, the shell checks to see if a file named `/etc/profile` exists on your UNIX system, and if it does, commands are read from it; `/etc/profile` is used by system administrators to set up variables needed by *all* users. Type:

```
cat /etc/profile
```

to see what your system administrator has already done for you. After this, the shell proceeds to see if you have a file named `.profile` in your login directory. If so, commands are read and executed from it. For a sample `.profile`, see *profile*(5). Finally, the shell is ready to read commands from your standard input—usually the terminal.

3.9.3 Execution Flags: `set`. The `set` command provides the capability of altering several aspects of the behavior of the shell by setting certain *shell flags*. In particular, the `x` and `v` flags may be useful from the terminal. Flags may be `set` by typing, for example:

```
set -xv
```

(to turn on flags `x` and `v`). The same flags may be turned *off* by typing:

```
set +xv
```

These two flags have the following meaning:

`-v` Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

`-x` Commands and their arguments are printed as they are executed. (Shell control commands, such as `for`, `while`, etc., are not printed, however.) Note that `-x` causes a trace of *only* those commands that are actually executed, whereas `-v` prints each line of input until a syntax error is detected.

The `set` command is also used to set these and other flags within shell procedures (see §4.7).

4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES

4.1 A Command's Environment

All the variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the `export` command. The `export` command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line (but see also `-k` flag in §4.7). Such variables are placed in the environment of the procedure being invoked. For example:

```
#       key_command
echo $a $b
```

is a simple procedure that `echo`es the values of two variables; if it is invoked as:

```
a=key1 b=key2 key_command
```

then the output is:

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count `$#` (§3.4.4).

A procedure may access the value of any variable in its environment; however, if changes are made to the value of a variable, these changes are *not* reflected in the environment—they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the `export` command within that procedure (but see §4.2 below). To obtain a list of variables that have been made `export`able from the current shell, type:

```
export
```

(You will also get a list of variables that have been made `readonly`—see §4.5 below.) To get a list of name-value pairs in the current environment, type:

```
env
```

4.2 Invoking the Shell

The shell is an ordinary command and may be invoked in the same way as other commands:

`sh` *proc* [*arg* ...]	A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated as described in §4.3.
`sh -v` *proc* [*arg*...]	This is equivalent to putting `set -v` at the beginning of *proc*. Similarly for the **x**, **e**, **u**, and **n** flags (§3.9.3, §4.7).
proc [*arg* ...]	If *proc* is marked executable, and is not a compiled, executable program, the effect is similar to that of `sh` *proc* [*args* ...]. An advantage of this form is that *proc* may be found by the search procedure described in §3.2 and §3.4.2. Also, variables that have been `export`ed in the shell will still be `export`ed from *proc* when this form is used (because the shell only `fork`s to read commands from *proc*). Thus any changes made within *proc* to the values of `export`ed variables will be passed on to subsequent commands invoked from within *proc*.

There are several shell invocation flags that are sometimes useful for more advanced shell programming. They are described in §5.8.

4.3 Passing Arguments to the Shell; `shift`

When a command line is scanned, any character sequence of the form **$***n* is replaced by the *n*th argument to the shell, counting the name of the shell procedure itself as **$0**. This notation permits direct reference to the procedure name and to as many as nine positional parameters (§3.4.1). Additional arguments can be processed using the `shift` command or by using a `for` loop (§4.4.4).

The `shift` command shifts arguments to the left; i.e., the value of **$1** is thrown away, **$2** replaces **$1**, **$3** replaces **$2**, etc.; the highest-numbered positional parameter becomes *unset*. (**$0** is *never* `shift`ed.) The command `shift` *n* is a shorthand notation for *n* consecutive `shift`s; `shift 0` does nothing. For example, consider the shell procedure `ripple` below: `echo` writes its arguments to the standard output; `while` is discussed in §4.4.3 (it is a looping command); lines that begin with **#** are comments.

```
#       ripple command
while test $# != 0
do
        echo $1 $2 $3 $4 $5 $6 $7 $8 $9
        shift
done
```

If the procedure were invoked by:

```
ripple a b c
```

it would print:

```
a b c
b c
c
```

The notation `$*` causes substitution of *all* positional parameters except `$0`. Thus, the `echo` line in the `ripple` example above could be written more compactly as:

```
echo $*
```

These two `echo` commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The `$*` notation is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command such as the `nroff` text formatter:

```
nroff -h -rW120 -T450 -cm $*
```

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a new-line or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via *grave accents*) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next the shell scans the resulting command line for *internal field separators*, that is, for any characters specified by `IFS` to break the command line into distinct arguments; *explicit* null arguments (specified by `""` or `''`) are retained, while *implicit* null arguments resulting from evaluation of variables that are null or not set are removed. Then file name generation occurs, with all meta-characters being expanded. The resulting command line is executed by the shell.

Sometimes, one builds command lines inside a shell procedure. In this case, one might want to have the shell rescan the command line after all the initial substitutions and expansions are done. The special command `eval` is available for this purpose; `eval` takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=' | wc -l'
eval $command $output
```

This segment of code results in the pipeline `who | wc -l` being executed.

The output of `eval` cannot be redirected; uses of `eval` can, however, be nested.

4.4 Control Commands

The shell provides several flow-of-control commands that are useful in creating shell procedures. To explain them, we first need a few definitions.

A *simple command* is as defined in §3.1. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by `|`. (For historical reasons, `^` is a synonym for `|` in this context.) The standard output of each command but the last in a pipeline is connected (by a *pipe*(2)) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is non-zero if the exit status of either the first or last process in the pipeline is non-zero. (This is a bit weird, and may be changed in the future.)

A *command list* is a sequence of one or more pipelines separated by `;`, `&`, `&&`, or `||`, and optionally terminated by `;` or `&`. A semicolon (`;`) causes sequential execution of the previous pipeline (i.e., the shell waits for the pipeline to finish before reading the next pipeline), while `&` causes asynchronous execution of the preceding pipeline; both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily, or until its processes are `killed`. In the first example below, the shell executes `who`, waits for it to terminate, then executes `date` and waits for it to terminate; in the second example, the shell invokes both commands in order, but does not wait for either one to finish.

Figure 2 shows the actions of the shell involved in executing these two command lists:

```
who >log; date
who >log& date&
```

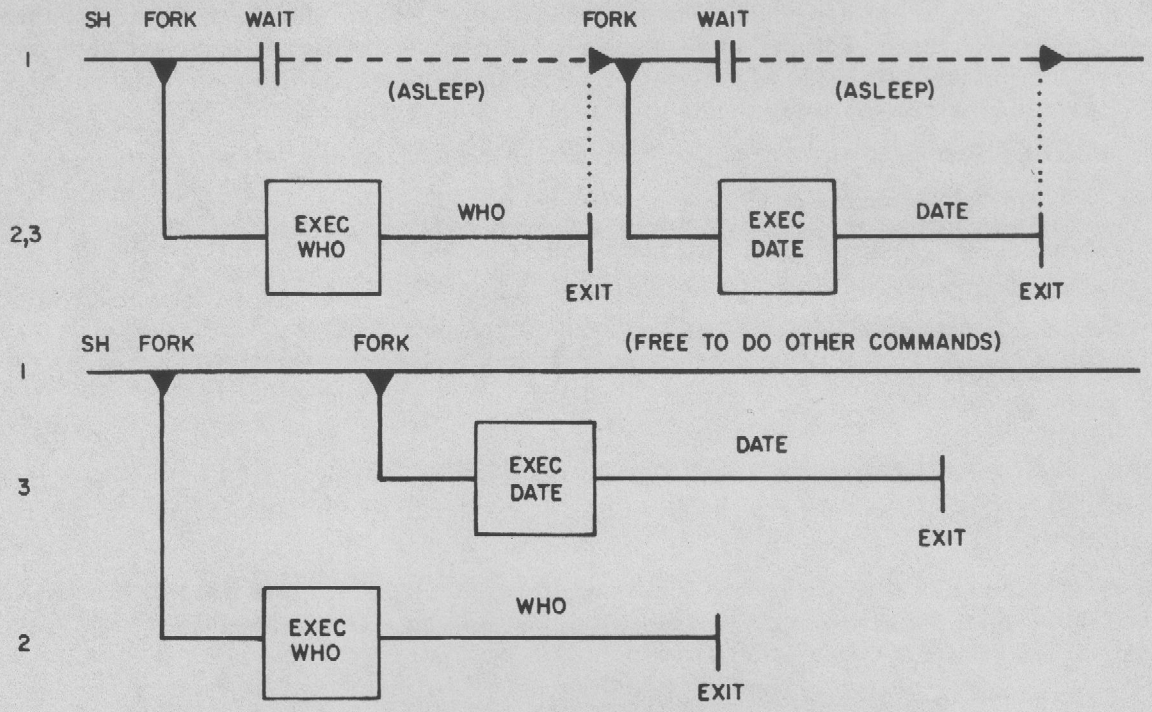

Figure 2

More typical uses of & include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type:

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with & is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The `nohup` command is used for this purpose. Without `nohup`, if you hang up while `cc` in the above example is still executing, `cc` will be killed and your output will disappear.

☞ *The & operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of simultaneous, asynchronous processes without a compelling reason for doing so.*

The `&&` and `||` operators, which are of equal precedence (but lower than `&` and `|`), cause conditional execution of pipelines. In `cmd1 || cmd2`, `cmd1` is executed and its exit status examined. Only if `cmd1` fails (i.e., has a non-zero exit status) is `cmd2` executed. This is thus a more terse notation for:

```
if      cmd1
        test $? != 0
then
        cmd2
fi
```

See `writemail` in §6 for an example of use of `||`.

The `&&` operator yields the complementary test: in `cmd1 && cmd2`, the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints *two* separate documents in a way similar to that shown in a previous example (§3.7):

```
{ nroff -cm text1; nroff -cm text2; } | col | greek -Thp
```

See §4.4.7 for further details on command grouping.

All of the following commands are formally described in *sh*(1).

4.4.1 Structured Conditional: `if`. The shell provides an `if` command. The simplest form of the `if` command is:

```
if command list
then command list
fi
```

The *command list* following `if` is executed and if the last command in the list has a *zero* exit status, then the *command list* that follows `then` is executed; `fi` indicates the end of the `if` command.

In order to cause an alternative set of commands to be executed in the case where the *command list* following `if` has a *non-zero* exit status, one may add an `else`-clause to the form given above. This results in the following structure:

```
if command list
then command list
else command list
fi
```

Multiple tests can be achieved in an `if` command by using the `elif` clause. For example:

```
if      test -f "$1" # is $1 a file?
then    pr $1
elif    test -d "$1" # else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a file name, then print that file; if not, then check to see if it is the name of a directory. If so, change to that directory and print all the files there. Otherwise, `echo` the error message.

The `if` command may be nested (but be sure to end each one with a `fi`). The new-lines in the above examples of `if` may be replaced by semicolons.

The exit status of the `if` command is the exit status of the last command executed in any `then` clause or `else` clause. If no such command was executed, `if` returns a zero exit status.

4.4.2 Multi-way Branch: `case`. A multiple way branch is provided by the `case` command. The basic format of `case` is:

```
case string in
pattern )  command list ; ;
   :
   :
pattern )  command list ; ;
esac
```

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in file-name generation (§3.3). If a match is found, the *command list* following the matched pattern is executed; the `;;` serves as a break out of the `case` and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if `*` is the first pattern in as `case`, no other patterns will ever be looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by `|`. For example:

```
case $i in
        *.c)     cc $i
                 ;;
        *.h|*.sh)          # do nothing
                 ;;
        *)       echo "$i of unknown type"
                 ;;
esac
```

In the above example, no action is taken for the second set of patterns because the *null* command is specified; `*` is used as a default pattern, because it matches any word.

The exit status of `case` is the exit status of the last command executed in the `case` command. If no commands were executed, then `case` has a zero exit status.

4.4.3 Conditional Looping: `while` and `until`. A `while` command has the general form:

```
while command list
do
        command list
done
```

The commands in the first *command list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second list are executed. This sequence is repeated as long as the exit status of the first *command list* is zero. A loop can be executed as long as the first *command list* returns a non-zero exit status by replacing `while` with `until`.

Any new-line in the above example may be replaced by a semicolon. The exit status of a `while` (`until`) command is the exit status of the last command executed in the *second* command list. If no such command is executed, `while` (`until`) has exit status zero.

4.4.4 Looping over a List: `for`. Often, one wishes to perform some set of operations for each in a set of files, or execute some command once for each of several arguments. The `for` command can be used to accomplish this. The `for` command has the format:

```
for variable in word list
do
        command list
done
```

where *word list* is a list of strings separated by blanks. The commands in the *command list* are executed once for each word in *word list*. *Variable* takes on as its value each word from *word list*, in turn; *word list* is fixed after it is evaluated the first time. For example, the following `for` loop will cause each of the C source files `xec.c`, `cmd.c`, and `word.c` in the current directory to be `diff`ed with a file of the same name in the directory `/usr/src/cmd/sh`:

```
for cfile in xec cmd word
do      diff $cfile.c /usr/src/cmd/sh/$cfile.c
done
```

One can omit the "`in` *word list*" part of a `for` command; this will cause the current set of positional parameters to be used in place of *word list*. This is very convenient when one wishes to write a command that performs the same set of commands for each of an unknown number of arguments. See `null` in §6 for an example of this feature.

4.4.5 Loop Control: `break` and `continue`. The `break` command can be used to terminate execution of a `while`, `until`, or a `for` loop; `continue` requests the execution of the next iteration of the loop. These commands are effective only when they appear between `do` and `done`.

The `break` command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched `done`. Exit from *n* levels is obtained by `break` *n*.

The `continue` command causes execution to resume at the nearest enclosing `while`, `until`, or `for`, i.e., the one that begins the innermost loop containing the `continue`; one can also specify an argument *n* to `continue` and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive; 'break' and 'continue'
# commands are used to allow the user to control data entry.
while true
do      echo "Please enter data"
        read response
        case "$response" in
                "done") break   # no more data
                        ;;
                "")     continue
                        ;;
                *)
                        process the data here
                        ;;
        esac
done
```

4.4.6 End-of-file and `exit`. When the shell reaches the end-of-file, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The `exit` command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated "normally" by using `exit 0`.

4.4.7 Command Grouping: Parentheses and Braces. There are two methods for grouping commands in the shell. As mentioned in §3.9.1, parentheses `( )` cause the shell to spawn a *sub-shell* that reads the enclosed commands. Both the right and left parentheses are recognized *wherever* they appear in a command line—they can appear as literal parentheses *only* by being quoted. For example, if you type `garble(stuff)` the shell interprets this as four separate words: `garble`, `(`, `stuff`, and `)`.

This sub-shell capability is useful if one wishes to perform some operations without affecting the values of variables in the current shell, or to temporarily change directory and execute some commands in the new directory without having to explicitly return to the current directory. The current environment is passed to the sub-shell and variables that are `exported` in the current shell are also `exported` in the sub-shell. Thus:

```
current=`pwd`; cd /usr/docs/sh_tut;
nohup mm -Tlp sc_? | lpr& cd $current
```

and:

```
(cd /usr/docs/sh_tut; nohup mm -Tlp sc_? | lpr&)
```

accomplish the same result: a copy of this tutorial is printed on the line printer; however, the second example automatically puts you back in your original working directory. In the second example above, blanks or new-lines surrounding the parentheses are allowed but not necessary. The shell will prompt with **$PS2** if a) is expected. See also the example in §3.9.1.

Braces { } may also be used to group commands together.[5] Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace { may be followed by a new-line (in which case the shell will prompt for more input). Unlike in the case of parentheses, no sub-shell is spawned for braces; the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command; see the last example in §4.4 above.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

4.4.8 Input/Output Redirection and Control Commands. The shell normally does not *fork* when it recognizes the *control* commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input (output) to (from) each command. Also, when redirection of input/output is specified explicitly for a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, or **for** is used in a pipeline consisting of more than one command, the shell *forks* and a sub-shell runs the control command. This has certain implications; the most noticeable one is that *any changes made to variables within the control command are not effective once that control command finishes* (similar to the effect of using parentheses to group commands). The control commands run slightly slower when redirection is specified.

☞ *Beginners should skip to Section 4.5 on first reading.*

4.4.9 In-line Input Documents. Upon seeing a command line of the form:

command << *eofstring*

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring* (possibly preceded by one or more tab characters). By appending a minus (-) to <<, leading tab characters are deleted from each line of the input document before the shell passes the line to *command.*

The shell creates a temporary file containing the input document and performs variable and command substitution (§3.4.3) on its contents before passing it to the command. Pattern matching on file names is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, one may quote any character of *eofstring*:[6]

command << \eofstring

The in-line input document feature is especially useful for small amounts of input data (e.g., an editor "script"), where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, one could type:

5. See §3.4.2 and §5.7 for other meanings of braces in the shell.

6. Typically, *eofstring* consists of a single character; ! is often used for this purpose.

```
cat <<- xyz
        This message will be printed on the
        terminal with leading tabs removed.
xyz
```

This in-line input document feature is most useful in shell procedures. See `edfind`, `edlast`, and `mmt` in §6. Note that in-line input documents may *not* appear within grave accents.[7]

4.4.10 Transfer to Another File and Back: the Dot (.) Command. A command line of the form:

. *proc*

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the *dot* command finishes. This is thus a good way to gather a number of shell variable initializations into one file. Note that an `exit` command in a file executed in this manner will cause an exit from your current shell; if you are at login level, you will be logged out.

4.4.11 Interrupt Handling: `trap`. As noted in §2.2, a program may choose to *catch* an interrupt from the terminal, *ignore* it completely, or be terminated by it. Shell procedures can use the `trap` command to obtain the same effects.

`trap` *arg signal-list*

is the form of the `trap` command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers (as described in *signal*(2)). The commands in *arg* are scanned at least once, when the shell first encounters the `trap` command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The former inhibit immediate command and variable substitution; this becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory on the file `errdirect` when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
           commands to be executed in directory $i here
done
```

while the same procedure with double (rather than single) quotes (`trap "echo `pwd` >errdirect" 2 3 15`) will, instead, print the name of the directory from which the procedure was executed.

Signal 11 (`SEGMENTATION VIOLATION`) may never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is not a UNIX signal, but is effectively interpreted by the `trap` command as a signal generated by exiting from a shell (either via an `exit` command, or by "falling through" the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string (`''` or `""`), then the signals in *signal-list* are *ignored* by the shell.

7. This is a implementation bug that should (and may) be fixed eventually.

The most frequent use of `trap` is to assure removal of temporary files upon termination of a procedure. The second example of §3.4.4 would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
        commands, some of which use  $temp, go here
```

In this example, whenever signals 1 (`HANGUP`), 2 (`INTERRUPT`), 3 (`QUIT`), or 15 (`SOFTWARE TERMINATION`) are received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotes will be executed. The `exit` command must be included, or else the shell continues reading commands where it left off when the signal was received. The `trap` 0 turns off the original trap on exits from the shell, so that the `exit` command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the shell continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the `read` command to return a non-zero exit status, thus terminating the `while` loop and restarting the cycle for the next directory; the entire procedure is terminated if interrupted when waiting for input, but during the execution of a command, an interrupt terminates *only* that command:

```
dir=`pwd`
for i in *
do      if test -d $dir/$i
        then    cd $dir/$i
                while   echo "$i:"
                        trap exit 2
                        read x
                do      trap : 2          # ignore interrupts
                        eval $x
                done
        fi
done
```

Several `traps` may be in effect at the same time; if multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the shell implements the `trap` command in order not to be surprised. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the shell *then* polls any traps that happen to be set and executes the appropriate `trap` commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level; in this case, it is possible that no child process is running, so the shell waits for the termination of the first process spawned *after* the signal is received before it polls the traps.

For internal commands, the shell normally polls traps on completion of the command; an exception to this rule is made for the `read` command, for which traps are serviced immediately, so that `read` can be interrupted while waiting for input.

4.5 Special Shell Commands

There are several special commands that are *internal* to the shell (some of which have already been mentioned). These commands should be used in preference to other UNIX commands whenever possible, because they are, in general, faster and more efficient. The shell does not fork to execute these commands, so no additional processes are spawned; the trade-off for this efficiency is that redirection of input/output is not allowed for most of these special commands.

Several of the special commands have already been described in §4.4 because they affect the flow of control. They are `break`, `continue`, `exit`, dot (`.`), and `trap`. The `set` command described in §3.4.1 and §3.9.3 is also a special command. Descriptions of the remaining special commands are given here:

`:`	The *null* command; this command does nothing; the exit status is zero (*true*). *Beware:* any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
`cd` *arg*	Make *arg* the current directory. If *arg* does not begin with `/`, `./`, or `../`, `cd` uses the `CDPATH` shell variable (§3.4.2) to locate a parent directory that contains the directory *arg*. If *arg* is not a directory, or the user is not authorized to access it, a non-zero exit status is returned. Specifying `cd` with no *arg* is equivalent to typing `cd $HOME`.
`exec` *arg* ...	If *arg* is a command, then the shell executes it without forking. No new process is created. Input/output redirection arguments *are* allowed on the command line. If *only* input/output redirection arguments appear, then the input/output of the shell itself is modified accordingly. See `merge` in §6 for an example of this use of `exec`.
`newgrp` *arg* ...	The *newgrp*(1) command is executed, replacing the shell; *newgrp* in turn spawns a new shell; see *newgrp*(1). *Beware:* Only variables in the environment will be known in the shell that is spawned by the *newgrp* command. Any variables that were `exported` will no longer be marked as such.
`read` *var* ...	One line (up to a new-line) is read from standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All left-over words are assigned to the *last* variable. The exit status of `read` is zero unless an end-of-file is read.
`readonly` *var* ...	The specified variables are made `readonly` so that no subsequent assignments may be made to them. If no arguments are given, a list of all `readonly` and of all `exported` variables is given.
`test`	A conditional expression is evaluated. More details are given in §5.1 below.
`times`	The accumulated user and system times for processes run from the current shell are printed.
`umask` *nnn*	The user file creation mask is set to *nnn*; see *umask*(2) for details. If *nnn* is omitted, then the current value of the mask is printed.
`ulimit` *n*	This command imposes a limit of *n* blocks on the size of files written by the shell and its child processes (files of any size may be read). If *n* is omitted, the current value of this limit is printed. The default value for *n* varies from on installation to another.
`wait` *n*	The shell waits for the child process whose process number is *n* to terminate; the exit status of the `wait` command is that of the process waited on. If *n* is omitted or is not a child of the current shell, then *all* currently active processes are waited for and the return code of the `wait` command is zero.

4.6 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps: first, one builds an ordinary text file; then one changes its *mode* to make it *executable,* thus permitting it to be invoked by *proc args*, rather than by `sh` *proc args*. The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for longer-lived ones. Here is the entire input needed to set up a simple procedure (the executable part of `draft` in §6):

```
ed
a
nroff -rC3 -T450-12 -cm $*
.
w draft
q
chmod +x draft
```

It may then be invoked as `draft file1 file2`. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If `draft` were thus created in a directory whose name appears in the user's `PATH` variable, the user could change working directories and still invoke the `draft` command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (`.`) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer to write shell procedures instead of C programs. First, it is easy to create and maintain a shell procedure because it is only a file of ordinary text. Second, it has no corresponding object program that must be generated and maintained. Third, it is easy to create a procedure on the fly, use it a few times, and then remove it. Finally, because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and/or shell procedures are usually named *bin*. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their `PATH` variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard—it may become difficult to keep track of your environment, and efficiency may suffer (§7.3).

4.7 More about Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

`-e` The shell will exit immediately if any command that it executes exits with a non-zero exit status.

`-u` When this flag is set, the shell treats the use of an unset variable as an error. This flag can be used to perform a global check on variables.

`-t` The shell exits after reading and executing the commands on the remainder of the current input line.

`-n` This is a *don't execute* flag. On occasion, one may want to check a procedure for syntax errors, but not to execute the commands in the procedure. Writing `set -nv` at the beginning of the file will accomplish this.

`-k` *All* arguments of the form *variable=value* are treated as keyword parameters. When this flag is *not* set, only such arguments that appear *before* the command name are treated as keyword parameters.

5. MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES

Shell procedures can make use of any UNIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use. More detailed descriptions of each of these commands can be found in Section 1 of the *UNIX User's Manual* [7].

5.1 Conditional Evaluation: `test`

The `test` command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status; otherwise, a non-zero (false) exit status is returned; `test` also returns a non-zero exit status if it has no arguments. Often it is convenient to use the `test` command as the first command in the *command list* following an `if` or a `while`. Shell variables used in `test` expressions should be enclosed in double quotes if there is any chance of their being null or not set.

On some UNIX systems, the square brackets (`[ ]`) may be used as an alias for `test`; e.g., `[` *expression* `]` has the same effect as `test` *expression.*

The following is a partial list of the primaries that can be used to construct a conditional expression:

`-r` *file*	*true* if the named file exists and is readable by the user.
`-w` *file*	*true* if the named file exists and is writable by the user.
`-x` *file*	*true* if the named file exists and is executable by the user.
`-s` *file*	*true* if the named file exists and has a size greater than zero.
`-d` *file*	*true* if the named file exists and is a directory.
`-f` *file*	*true* if the named file exists and is an ordinary file.
`-p` *file*	*true* if the named file exists and is a named pipe (*fifo*).
`-z` *s1*	*true* if the length of string *s1* is zero.
`-n` *s1*	*true* if the length of the string *s1* is non-zero.
`-t` *fildes*	true if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
s1 `=` *s2*	*true* if strings *s1* and *s2* are identical.
s1 `!=` *s2*	*true* if strings *s1* and *s2* are *not* identical.
s1	*true* if *s1* is *not* the null string.
n1 `-eq` *n2*	*true* if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by `-ne`, `-gt`, `-ge`, `-lt`, and `-le`.

These primaries may be combined with the following operators:

`!`	unary negation operator.
`-a`	binary logical *and* operator.
`-o`	binary logical *or* operator; it has lower precedence than `-a`.
`(` *expr* `)`	parentheses for grouping; they must be escaped to remove their significance to the shell; in the absence of parentheses, evaluation proceeds from left to right.

Note that all primaries, operators, file names, etc., are separate arguments to `test`.

5.2 Reading a Line: `line`

The `line` command takes one line from standard input and prints it on standard output. This is useful when you need to read a line from a file, or capture the line in a variable. The functions of `line` and of the `read` command that is internal to the shell differ in that input/output redirection is possible only with `line`. If the user does not require input/output redirection, `read` is faster and more efficient. An example of a usage of `line` for which `read` would not suffice is:

```
firstline=`line < somefile`
```

5.3 Simple Output: `echo`

The `echo` command, invoked as `echo` [*arg* ...] copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a new-line; often, it is used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command `ls` is often replaced by `echo *`, because the latter is faster and prints fewer lines of output.

The `echo` command recognizes several escape sequences. A `\n` yields a new-line character; a `\c` removes the new-line from the end of the `echo`ed line. The following prompts the user, allowing one to type on the same line as the prompt:

```
echo 'enter name:\c'
read name
```

The `echo` command also recognizes octal escape sequences for *all* characters, whether printable or not: `echo "\007"` typed at a terminal will cause the bell on that terminal to ring.

5.4 Expression Evaluation: `expr`

The `expr` command provides arithmetic and logical operations on integers and some pattern matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; `expr` can be used inside grave accents to set a variable. Typical examples are:

```
#       increment $a
a=`expr $a + 1`
#       put third through last characters of
#       $1 into substring
substring=`expr "$1" : '..\(.*\)'`
#       obtain length of $1
c=`expr "$1" : '.*'`
```

The most common uses of `expr` are in counting iterations of a loop and in using its pattern matching capability to pick apart strings; see *expr*(1) for more details.

5.5 `true` and `false`

The `true` and `false` commands perform the obvious functions of exiting with zero and non-zero exit status, respectively. The `true` command is often used to implement an unconditional loop.

5.6 Input/Output Redirection Using File Descriptors.

☞ *Beginners should skip this section on first reading.*

Above (§3.6.2), we mentioned that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with *any* file descriptor by using the *write*(2) system call. The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing:

fd1>&*fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* onto the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at execution time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

```
command 2>&1
```

If one wanted to redirect both standard output and standard error output to the same file, one would type:

```
command 1> file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is *currently* associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard *input*. One could type:

fda<&*fdb*

to cause both file descriptors *fda* and *fdb* to be associated with the same input file; if *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for commands that use two or more input sources. Another use of this notation is for sequential reading and processing of a file; see `merge` in §6 for an example of use of this feature.

5.7 Conditional Substitution

Normally, the shell replaces occurrences of **$***variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in any one of the following ways:

```
A=
bcd=""
Ef_g=''
set '' ""
```

The first three of these examples assign the null string to each of the corresponding *shell variables*. The last example sets the first and second *positional parameters* to the null string, and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null* (note that, in these expressions, *variable* refers to either a digit or a variable name and the meaning of braces differs from that described in §3.4.2 and §4.4.7):

${*variable*:-*string*}	If *variable* is set and is non-null, then substitute the value **$***variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.
${*variable*:=*string*}	If *variable* is set and is non-null, then substitute the value **$***variable* in place of this expression; otherwise, set *variable* to *string*, and then substitute the value **$***variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

`${`*variable*`:?`*string*`}` If *variable* is set and is non-null, then substitute the value of *variable* for the expression; otherwise, print a message of the form:

> *variable*`:` *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message:

> *variable*`:  parameter null or not set`

is printed instead.

`${`*variable*`:+`*string*`}` If *variable* is set and is non-null, then substitute *string* for this expression, otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon (`:`), in which case the shell does *not* check whether *variable* is null or not; it only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If `PATH` has ever been set and is not null, then keep its current value; otherwise, set it to the string `:/bin:/usr/bin`. Note that one needs an explicit assignment to set `PATH` in this form:

   ```
   PATH=${PATH:-':/bin:/usr/bin'}
   ```

2. If `HOME` is set and is not null, then change directory to it, otherwise set it to the given value and change directory to it; note that `HOME` is automatically assigned a value in this case:

   ```
   cd ${HOME:='/usr/gas'}
   ```

5.8 Invocation Flags

There are four flags that may be specified on the command line invoking the shell; these flags may *not* be turned on via the `set` command:

`-i` If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, `INTERRUPT` (signal 2) is caught and ignored, while `QUIT` (signal 3) and `SOFTWARE TERMINATION` (signal 15) are ignored.

`-s` If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system effectively has the `-s` flag turned on.

`-c` When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multi-word string, in order to allow for variable substitution.

`-r` When this flag is specified on invocation, then the *restricted shell* is invoked. This is a version of the shell in which certain actions are disallowed. In particular, the `cd` command produces an error message, and the user cannot set `PATH`. See *sh*(1) for a more detailed description.

6. EXAMPLES OF SHELL PROCEDURES

☞ *Some examples in this section are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.*

copypairs:

```
#       usage: copypairs file1 file2 ...
#       copy file1 to file2, file3 to file4, ...
while test "$2" != ""
do
        cp $1 $2
        shift; shift
done
if test "$1" != ""
then echo "$0: odd number of arguments"
fi
```

Note: This procedure illustrates the use of a `while` loop to process a list of positional parameters that are somehow related to one another. Here a `while` loop is much better than a `for` loop, because you can adjust the positional parameters via `shift` to handle related arguments.

copyto:

```
#       usage: copyto dir file ...
#       copy argument files to 'dir', making sure that at least
#       two arguments exist and that 'dir' is a directory
if test $# -lt 2
then    echo "$0: usage: copyto directory file ..."
elif test ! -d $1
then    echo "$0: $1 is not a directory";
else    dir=$1; shift
        for eachfile
        do
                cp $eachfile $dir
        done
fi
```

Note: This procedure uses an `if` command with two tests in order to screen out improper usage. The `for` loop at the end of the procedure loops over all of the arguments to `copyto` but the first; the original `$1` is shifted off.

distinct:

```
#       usage: distinct
#       reads standard input and reports list of alphanumeric strings
#       that differ only in case, giving lower-case form of each
tr -cs '[A-Z][a-z][0-9]' '[\012*]' | sort -u |
        tr '[A-Z]' '[a-z]' | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. It may not be immediately obvious how this works. (You may wish to consult *tr*(1), *sort*(1), and *uniq*(1) if you are completely unfamiliar with these commands.) The `tr` translates all characters except letters and digits into new-line characters, and then squeezes out repeated new-line characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The `sort` command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next `tr` converts everything to lower case, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The `uniq -d` prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged; the two lines below are equivalent, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3; rm temp[12]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic `distinct` with such a step-by-step process, using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, exercise some restraint, lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

draft:

```
#       usage: draft file(s)
#       prints the draft (-rC3) of a document on a DASI 450
#       terminal in 12-pitch using memorandum macros (MM).
nroff -rC3 -T450-12 -cm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

edfind:

```
#       usage: edfind file arg
#       find the last occurrence in 'file' of a line whose
#       beginning matches 'arg', then print 3 lines (the one
#       before, the line itself, and the one after)
ed - $1 <<!
H
?^$2?;-,+p
!
```

Note: This procedure illustrates the practice of using editor (`ed`) in-line input scripts into which the shell can substitute the values of variables. It is a good idea to turn on the `H` option of `ed` when embedding an `ed` script in a shell procedure (see *ed*(1)).

edlast:

```
#       usage: edlast file
#       prints the last line of file, then deletes that line
ed - $1 <<-\eof          # no variable substitutions in "ed" script
        H
        $p
        $d
        w
        q
eof
echo Done.
```

Note: This procedure contains an in-line input document or script (see §4.4.9); it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, `eof`) of the input redirection. If this had not been done, `$p` and `$d` would have been treated as shell variables.

fsplit:

```
#       usage: fsplit file1 file2
#       read standard input and divide it into three parts:
#       append any line containing at least one letter
#       to file1, any line containing at least one digit
#       but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
        total="`expr $total + 1`"
        case "$next"  in
        *[A-Za-z]*)
                echo  "$next"  >> $1 ;;
        *[0-9]*)
                echo  "$next"  >> $2 ;;
        *)
                lost="`expr $lost + 1`"
        esac
done
echo "$total lines read, $lost thrown away"
```

Note: In this procedure, each iteration of the `while` loop reads a line from the input and analyzes it. The loop terminates only when `read` encounters an end-of-file.

☞ *Don't use the shell to read a line at a time unless you must—it can be grotesquely slow* (§7.2.1).

initvars:

```
#       usage: . initvars
#       use carriage return to indicate "no change"
echo "initializations? \c"
read response
if test "$response" = y
then    echo "PS1=\c"; read temp
                PS1=${temp:-$PS1}
        echo "PS2=\c"; read temp
                PS2=${temp:-$PS2}
        echo "PATH=\c"; read temp
                PATH=${temp:-$PATH}
        echo "TERM=\c"; read temp
                TERM=${temp:-$TERM}
fi
```

Note: This procedure would be invoked by a user at the terminal, or as part of a `.profile` file. The assignments are effective even when the procedure is finished, because the *dot* command is used to invoke it. To better understand the *dot* command, invoke `initvars` as indicated above and check the values of `PS1`, `PS2`, `PATH`, and `TERM`; then make `initvars` executable, type `initvars`, assigning different values to the three variables, and check again the values of these three shell variables after `initvars` terminates. It is assumed that `PS1`, `PS2`, `PATH`, and `TERM` have been exported, presumably by your `.profile` (§3.9.2, §4.1).

merge:

```
#       usage:  merge src1 src2 [ dest ]
#       merge two files, every other line.
#       the first argument starts off the merge,
#       excess lines of the longer file are appended to
#       the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}            # default destination file is named $1.m
while true
do
                          # alternate reading from the files;
                          # 'more' represents the file descriptor
                          # of the longer file
        line <&4 >>$dest || { more=5; break ;}
        line <&5 >>$dest || { more=4; break ;}
done
                          # delete the last line of destination
                          # file, because it is blank.
ed - $dest <<\eof
        H
        $d
        w
        q
eof
while line <&$more >> $dest
do :; done                # read the remainder of the longer
                          # file - the body of the 'while' loop
                          # does nothing; the work of the loop
                          # is done in the command list following
                          # 'while'
```

Note: This procedure illustrates a technique for reading sequential lines from a file or files without creating any sub-shells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used `src1` and `src2` explicitly rather than the associated file descriptors, this procedure would never terminate, because the *first* line of each file would be read over and over again.

mkfiles:

```
#       usage: mkfiles pref [ quantity ]
#       makes 'quantity' (default = 5) files, named pref1, pref2, ...
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
        > $1$i
        i="`expr $i + 1`"
done
```

Note: This procedure uses input/output redirection to create zero-length files. The `expr` command is used for counting iterations of the `while` loop. Compare this procedure with procedure `null` below.

mmt:

```
if test "$#" = 0; then cat <<\!
Usage: "mmt [ options ] files" where "options" are:
-a       => output to terminal
-e       => preprocess input with eqn
-t       => preprocess input with tbl
-Tst     => output to STARE
-T4014 => output to Tektronix 4014
-Tvp     => output to Versatec printer
-        => use instead of "files" when mmt used inside a pipeline.
Other options as required by TROFF and the MM macros.
!
         exit 1
fi
PATH='/bin:/usr/bin'; O='-g'; o='|gcat -ph';
#                 Assumes typesetter is accessed via gcat(1)
#                 If typesetter is on-line, use O=''; o=''
while test -n "$1" -a ! -r "$1"
do case "$1" in
         -a)              O='-a';          o='' ;;
         -Tst)            O='-g';          o='|gcat -st';;
#                 Above line for STARE only
         -T4014)          O='-t';          o='|tc';;
         -Tvp)            O='-t';          o='|vpr -t';;
         -e)              e='eqn';;
         -t)              f='tbl';;
         -)               break;;
         *)               a="$a $1";;
   esac
   shift
done
if test -z "$1";          then echo 'mmt: no input file'; exit 1; fi
if test "$O" = '-g';      then x="-f$1";  fi
d="$*"
if test "$d" = '-';       then shift;     x='';    d='';    fi
if test -n "$f";          then f="tbl $*|";        d='';    fi
if test -n "$e"
         then    if test -n "$f"
                          then e='eqn|'
                          else e="eqn $*|";        d=''
                 fi
fi
eval "$f $e troff $O -cm $a $d $o $x";  exit 0
```

Note: This is a slightly simplified version of an actual UNIX command (although this is *not* the version included in UNIX Release 4.0). It uses many of the features available in the shell; if you can follow through it without getting lost, you have a good understanding of shell programming. Pay particular attention to the process of building a command line from shell variables and then using `eval` to execute it.

null:

```
#        usage: null file
#        create each of the named files as an empty file
for eachfile
do
         > $eachfile
done
```

Note: This procedure uses the fact that output redirection creates the (empty) output file if that file does not already exist. Compare this procedure with procedure `mkfiles` above.

phone:

```
#       usage: phone initials
#       prints the phone number(s) of person with given initials
echo 'inits     ext     home'
grep "^$1" <<\!
abc     1234    999-2345
def     2234    583-2245
ghi     3342    988-1010
xyz     4567    555-1234
!
```

Note: This procedure is an example of using an in-line input document or *script* to maintain a *small* data base.

writemail:

```
#       usage: writemail message user
#       if user is logged in, write message on terminal;
#       otherwise, mail it to user
echo "$1" | { write "$2" || mail "$2" ;}
```

Note: This procedure illustrates command grouping. The message specified by `$1` is piped to the `write` command and, if `write` fails, to the `mail` command.

7. EFFECTIVE AND EFFICIENT SHELL PROGRAMMING

7.1 Overall Approach

This section outlines strategies for writing *efficient* shell procedures, i.e., ones that do not waste resources unreasonably in accomplishing their purposes. In the authors' opinion, the primary reason for choosing the shell procedure as the implementation method is to achieve a desired result at a minimum *human* cost. Emphasis should *always* be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, one should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume a lot of resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the `time` command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by, for instance, variations in system load.

7.2 Approximate Measures of Resource Consumption

7.2.1 Number of Processes Generated. When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built-in:

```
break       case        cd          continue    eval        exec
exit        export      for         if          newgrp      read
readonly    set         shift       test        times       trap
ulimit      umask       until       wait        while       .
:           {...}
```

(. . .) executes as a child process, i.e., the shell does a *fork*, but no *exec*. Any command *not* in the above list requires both *fork* and *exec*.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by:

processes = k*n + c

where *k* and *c* are constants, and *n* is the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero. Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of procedure `fsplit` of §6. For each iteration of the loop, there is one `expr` plus either an `echo` or another `expr`. One additional `echo` is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n + 1. On the other hand, the number of processes in the following (equivalent) procedure is 12, regardless of the number of lines of input:

```
#       faster fsplit
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$                # read standard input into temp file
                            # save original lengths of $1, $2
if test -s "$1"; then start1=`wc -l < $1`; fi
if test -s "$2"; then start2=`wc -l < $2`; fi
grep "$b" temp$$ >> $1  # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                            # lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \( $end1 - $start1 \) - \( $end2 - $start2 \)`"
echo "$total lines read, $lost thrown away"
```

This version is often ten times faster than `fsplit`, and it is even faster for larger input files.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C.

☞ *Shell procedures should not be used to scan or build files a character at a time.*

7.2.2 Number of Data Bytes Accessed. It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. Which of the following is likely to be faster?

```
sort file | grep pattern
grep pattern file | sort
```

7.2.3 Directory Searches. Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long path names. Judicious use of `cd` can help shorten long path names and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system):[8]

```
    time sh -c 'ls -l /usr/bin/* >/dev/null'
    time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

7.3 Efficient Organization

7.3.1 Directory-Search Order and the `PATH` Variable. The `PATH` variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead that occurs in a subtle, but avoidable, way.

The process of finding a command involves reading every directory included in every path name that precedes the needed path name in the current `PATH` variable. As an example, consider the effect of invoking `nroff` (i.e., `/usr/bin/nroff`) when `$PATH` is `:/bin:/usr/bin`. The sequence of directories read is: `.`, `/`, `/bin`, `/`, `/usr`, and `/usr/bin`, i.e., a total of six directories. A long path list assigned to `PATH` can increase this number significantly.

The vast majority of command executions are of commands found in `/bin` and, to a somewhat lesser extent, in `/usr/bin`. Careless `PATH` setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (but *only* with respect to the efficiency of command searches):

```
    :/a1/tf/jtb/bin:/usr/lbin:/bin:/usr/bin
    :/bin:/a1/tf/jtb/bin:/usr/lbin:/usr/bin
    :/bin:/usr/bin:/a1/tf/jtb/bin:/usr/lbin
    /bin::/usr/bin:/a1/tf/jtb/bin:/usr/lbin
```

The first one above should be avoided. The others are acceptable, the choice among them is dictated by the rate of change in the set of commands kept in `/bin` and `/usr/bin`.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by `set`ting the `PATH` variable inside the procedure such that the fewest possible directories are searched in an optimum order; the `mmt` example in §6 does this.

7.3.2 Good Ways to Set up Directories. It is wise to avoid directories that are larger than necessary. You should be aware of several *magic* sizes. A directory that contains entries for up to 30 files (plus the required `.` and `..`) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a *small* file; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink.

ACKNOWLEDGEMENTS

The UNIX shell was initially written by S. R. Bourne [2,3]. Its design is based, in part, on the original UNIX shell [15] and on the PWB/UNIX shell [12], some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System and of the MIT Compatible Time-Sharing System. T. E. Fritz and several other colleagues provided helpful comments during the writing of this tutorial; T. A. Dolotta, in addition, provided a great deal of editorial assistance.

---

8. You may have to do some reading in the *UNIX User's Manual* [7] to understand exactly what is going on in these examples.

REFERENCES

[1] Bianchi, M. H., and Wood, J. L. A User's Viewpoint on the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering,* pp. 193-99 (Oct. 13-15, 1976).

[2] Bourne, S. R. The UNIX Shell. *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 1971-90 (July-Aug. 1978).

[3] Bourne, S. R. *An Introduction to the UNIX Shell.* Bell Laboratories (1979).

[4] Dolotta, T. A., Haight, R. C., and Mashey, J. R. The Programmer's Workbench. *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 2177-200 (July-Aug. 1978).

[5] Dolotta, T. A., and Mashey, J. R. An Introduction to the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering,* pp. 164-68 (Oct. 13-15, 1976).

[6] Dolotta, T. A., and Mashey, J. R. Using a Command Language as the Primary Programming Tool. In: Beech, D. (ed.), *Command Language Directions* (Proc. of the Second IFIP Working Conf. on Command Languages), pp. 35-55. Amsterdam: North Holland (1980).

[7] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G., eds. *UNIX User's Manual*—Release 3.0. Bell Laboratories (June 1980).

[8] Kernighan, B. W., and Mashey, J. R. The UNIX Programming Environment. *COMPUTER,* Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in *Software—Practice & Experience,* Vol. 9, No. 1, pp. 1-15 (Jan. 1979).

[9] Kernighan, B. W., and Plauger, P. J. Software Tools. *Proc. First Nat. Conf. on Software Engineering,* pp. 8-13 (Sept. 11-12, 1975).

[10] Kernighan, B. W., and Plauger, P. J. *Software Tools.* Reading, MA: Addison-Wesley (1976).

[11] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall (1978).

[12] Mashey, J. R. *PWB/UNIX Shell Tutorial.* Bell Laboratories (1977).

[13] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, pp. 1905-29 (July-Aug. 1978).

[14] Snyder, G. A., and Mashey, J. R. UNIX Documentation Road Map. Bell Laboratories (January 1981).

[15] Thompson, K. The UNIX Command Language. In: *Structured Programming—Infotech State of the Art Report,* pp. 375-84. Infotech International Limited, Nicholson House, Maidenhead, Berkshire, England (1976).

January 1981

CONTENTS