1189 B.5

PWB/UNIX

Shell Tutorial

J. R. Mashey

September 1977

Bell Telephone Laboratories, Incorporated

PWB/UNIX Shell Tutorial

CONTENTS

PWB/UNIX Shell Tutorial

J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

The command language for PWB/UNIX* is a high-level programming language that is an extended version of the UNIX Shell. By utilizing the Shell as a programming language, one can eliminate much of the programming drudgery that often accompanies a large project. Many manual procedures can be quickly, cheaply, and conveniently automated. Because it is so easy to create and use Shell procedures, individual users and entire projects can customize the general PWB/UNIX environment into one tailored to their own respective requirements, organizational structure, and terminology.

This paper is actually a combination of several tutorials, as explained in {1}.[1] Some sections provide a basic tutorial for relatively new users. Other sections are intended for more experienced users and introduce them to Shell programming. Finally, some hints on programming techniques and efficiency are offered for those who make especially heavy use of Shell programming.

The accuracy of this tutorial is guaranteed *only* for the Shell of PWB/UNIX—Edition 1.0. Other versions of UNIX have other Shells. Although many of the basic concepts are similar, there exist many differences in features, especially those used to support Shell programming.

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 the end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool for such projects. 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 [1,2,6].

When users log into a PWB/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 later use by saving them in a file, called a *Shell procedure*, a *command file*, or a *runcom*, according to local preferences.

Some users need little knowledge of the Shell to do their work; others choose to 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 PWB/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 procedures is given in {6}. Finally, a brief discussion of efficiency is offered in {7}. This is found in its proper place (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 {7} for an appropriate plan of study. All the *commands* mentioned below are described in Section I of the *PWB/UNIX User's Manual* [3], while *system calls* are described in Section II and *subroutines* in Section III thereof.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of some later discussions depends on familiarity with PWB/UNIX; [9] is most useful for that, and it would be helpful to read at least one of [4,5,10]. For completeness, a short overview of the most relevant concepts is given below.

---

* UNIX is a Trademark/Service Mark of the Bell System.

1. The notation {*n*} refers to Section *n* of this tutorial.

2.1 File System

The PWB/UNIX file system's overall structure is that of a rooted tree composed of *directories* and other files. A *file name* is a sequence of characters. A *pathname* is a sequence of directory names followed by a file name, each separated from the previous one by a slash (/). If a pathname 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 pathname* because it is invariant with regard to the user's current directory.) The user may change the current directory at any time by using the *cd* or *chdir* command. In most cases, a file name and its corresponding pathname may be used interchangeably. Some sample names are:

.	name of the current directory.
..	name of the parent directory of the current directory.
/	root directory of the entire file structure.
/bin	directory containing most of the frequently-used public commands.
/al/tf/jtb/bin	a full pathname typical of multi-person programming projects. This one happens to be a private directory of commands belonging to person "jtb" in project "tf"; "al" is the name of a *file system.*
bin/umail	a name depending on the current directory: it names file "umail" in subdirectory "bin" of the current directory. If the current directory is "/", it names "/bin/umail". If the current directory is "/al/tf/jtb", it names "/al/tf/jtb/bin/umail".
memox	name of a file in the current directory.

2.2 Processes

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

An *image* is a computer execution environment, including memory image, register values, current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image; most PWB/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 the program can determine whether it is executing as parent or child. The 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 quite commonly issues a *wait* system call to suspend execution until a child exits.

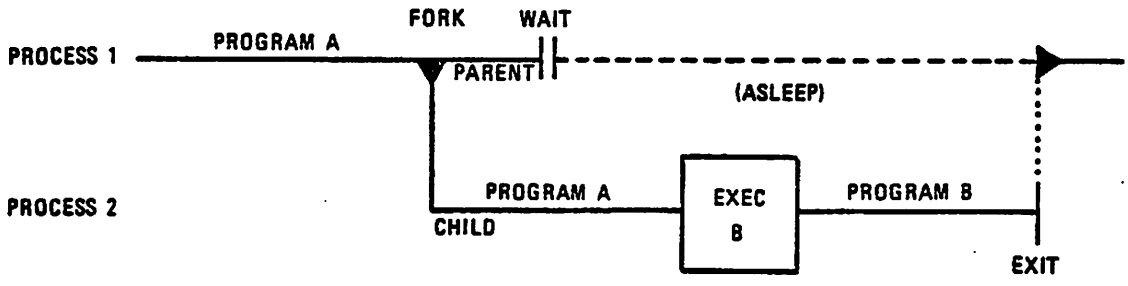

Figure 1

Figure 1 illustrates these ideas. *Program A* is executing (as *process 1*) and wishes to run *program B.* It "forks" and spawns a child (*process 2*) that continues to execute *program A.* The child abandons *A* by executing *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 a common input stream in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations. *Read* and *write* system calls copy a requested number of bytes from or 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. *Seek* 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.

When a process terminates, it can set an eight-bit *return code* (or *exit code*) that is available to its parent. This code is usually used to indicate success or failure.

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 keyboard, or by PWB/UNIX itself. For most types of 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 [4.6]. 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, *pr* 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 terminating completely.

Limiting interprocess communication to a small number of well-defined methods is a great aid to uniformity, understandability, and reliability of programs. It encourages the "packaging" of each function into a small program that is easily connected to other programs, but depends very little on the internal workings of other programs.

3. SHELL BASICS

The *Shell* (i.e., the *sh* command) implements the command language visible to most PWB/UNIX users. It reads input from a terminal or a file and arranges for the execution of the requested commands. It is a small program (about forty pages of C code); many of its functions are actually provided by independent programs that work with it. It is *not* part of the operating system, but is an ordinary user program. The discussion below is adapted from [10,11].

3.1 Commands

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

```
who
```

which prints the login names of logged-in users. The following line requests the *pr* command to print files a, b, and c:

```
pr a b c
```

If the first argument names a file that is *executable*[2] and is actually a load module, the Shell (as parent) spawns a new (child) process that immediately executes that program. If the file is marked executable, but is neither a load module nor a directory, it is assumed to be a *Shell procedure,* i.e., a file of ordinary text containing Shell command lines and possibly lines to be read by other programs. In this case, the Shell spawns a new instance of itself to read the file and execute the commands included in it. The following command requests that the on-line *PWB/UNIX User's Manual* [3] pages for the *who* and *pr* commands be printed on the terminal (the *man* command is actually implemented as a Shell procedure):

```
man who pr
```

2. As evidenced by an appropriate set of permission bits associated with that file.

From the user's viewpoint, executable 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 implementation choice for a given command. The actions of the Shell in executing any of these commands are illustrated in Figure 1 {2.2}.

3.2 Redirection of Standard Input and Output

When a command begins execution, it usually expects that three files are already open, a "standard input", a "standard output", and a "diagnostic output". When the user's original Shell is started, all three have already been opened to the user's terminal. A child process normally inherits these files from its parent. 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 standard input or output, respectively. An argument of the form ">>file" opens the standard output to the end of the file, thus providing a way to append data to it. In either output case, the Shell creates the file if it did not already exist. The following appends to file "log" the list of users who are logged in:

```
who >>log
```

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or file. Thus, commands can be used conveniently in many different contexts. 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).

Redirection of the diagnostic output is discussed in {4.7.3}.

3.3 Command Lines

A sequence of commands separated by "|" (or "^") make up a *pipeline.* 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 input, transforms it in some way, then writes it as 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, 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; *reform* is used here to speed printing by converting the (tab-less) output of *col* to an equivalent one containing horizontal tab characters. The flag "–mm" indicates one of the more-commonly used formatting options, and "text" is the name of the file to be formatted:

```
nroff –mm text | col | reform
```

Figure 2 shows the sequence of actions that set up this pipeline. Not shown are actions by the Shell that create pipes and manipulate open files, causing the commands to be tied together correctly.

A *command line* consists of zero or more pipelines separated by semicolons or ampersands. If the last command in a pipeline is terminated by a semicolon (;) or a new-line character, the Shell waits for the command to finish before continuing to read command lines. It does *not* wait if the pipeline is terminated by an ampersand (&); both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily, or until its processes are *kill*ed. The first example below executes *who,* waits for it to terminate, and then executes *date;* the second invokes both commands in order, but does not wait for either one to finish. Figure 3 shows the actions of the Shell involved in executing these examples:

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

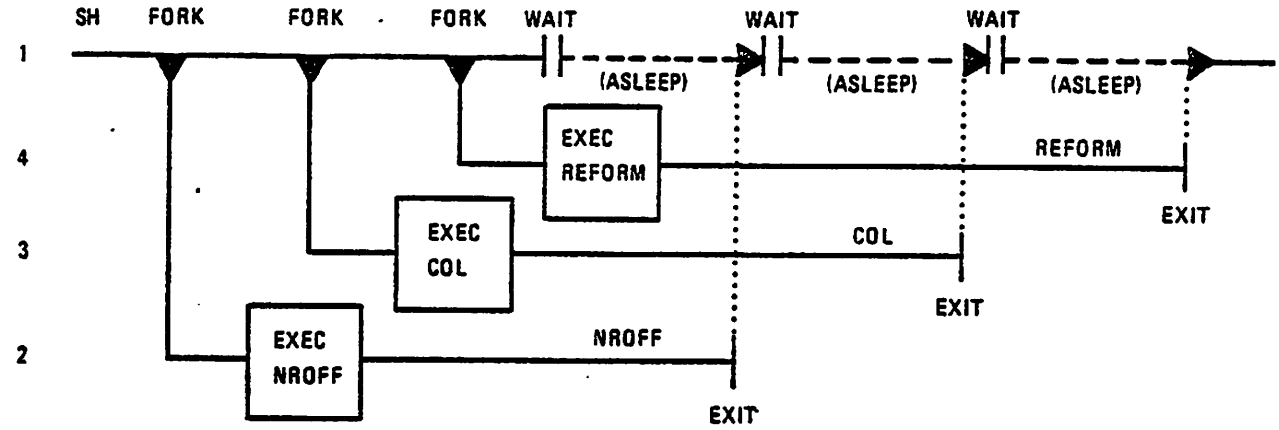

Figure 2

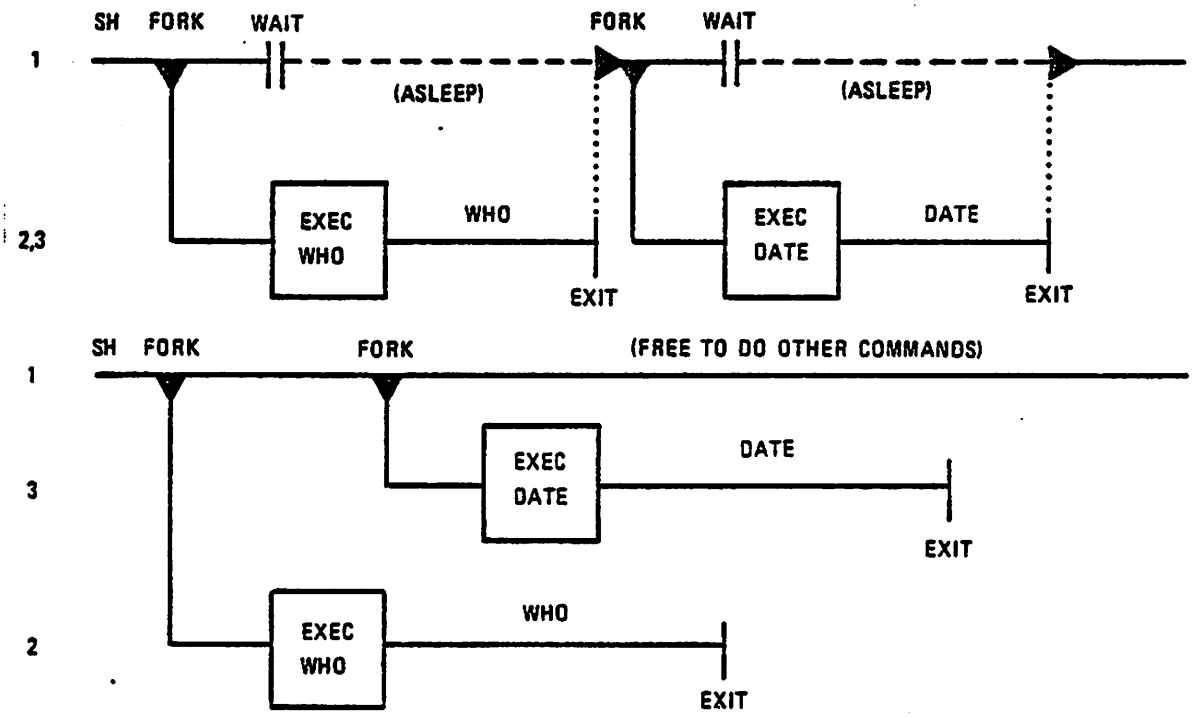

Figure 3

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

nohup cc prog.c&
You continue working while the C compiler runs in background.

A command terminated by "&" is immune to interrupts, 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* (the C compiler) 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.*

A simple command in a pipeline may be replaced by a command line enclosed in parentheses "()"; in this case, another instance of the Shell is spawned to execute the commands so enclosed. This action is

helpful in combining the output of several sequentially executed commands into a stream to be processed by a pipeline. The following line prints two separate documents in a way similar to that shown in a previous example:

```
(nroff -mm text1; nroff -mm text2) | col | reform
```

3.4 Generation of Argument Lists

Many command arguments are names of files. When certain characters are found in an argument, they cause replacement of that argument by a sorted list of zero or more file names obtained by pattern-matching on the contents of a directory. Most characters match themselves. The "?" matches *any one character;* the "*" matches *any string* of any characters (other than "/"), including the null string. Enclosing a set of characters within square brackets "[...]" causes the construct to match *any one character* in that set.[3] Inside square brackets, a pair of characters separated by "-" includes in the set all characters lexically within the inclusive range of that pair.

For example, "*" matches all files in the current directory, "*tmp*" matches all names containing "tmp", "[a-f]*" matches all files whose names begin with "a" through "f", "*.c" matches all files ending in ".c", and "/a1/tf/bin/?" matches all single-character names found in "/a1/tf/bin". This capability saves much typing, and more importantly, makes it possible to organize information as large collections of small files that are named in disciplined ways.

Pattern-matching has several restrictions. If the first character of a file name is ".", it can be matched only by an argument that begins with ".". Pattern-matching is currently restricted to the last component in a pathname—the string "/a1/tf/*" is legal, but the string "/a1/*/bin" is not. Pattern-matching does not apply to the name of the invoked command (i.e., argument number 0).

3.5 Quoting Mechanisms

If a character has a special meaning to the Shell, that meaning may be removed by preceding the character with a back-slash (\); the "\" acts as an escape and disappears. A "\" followed by a new-line character is treated as a blank, permitting continuation of commands on additional input lines. A sequence of characters enclosed in single quotes ('...') is taken literally—"what you see is what you get". The beginner should use single quotes in most instances. Double quotes ("...") are required in a few cases, primarily inside Shell procedures. Double quotes hide the significance of most special characters, but allow substitution of Shell arguments and variables; see {4.8} for further details.

3.6 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 "pw".
- who | grep pw | sort | pr
  Print an alphabetized, paginated list of logged-in users whose names contain "pw".

3. Be warned that square brackets are also used below in an entirely different sense: in descriptions of commands, they indicate that the enclosed argument is optional.

- (date; who | wc -l) >>log
  Append (to "log") the current date followed by the count of logged-in users.
- 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 it with other commands. The kinds of operations illustrated above may be used in other circumstances; *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.

3.7 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 use the command name as given; 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, "/bin", and "/usr/bin". For example, the *pr* and *man* commands are actually located in files "/bin/pr" and "/usr/bin/man", respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command via an absolute pathname. If a command name *as given* contains a "/" (e.g., "/bin/sort" or "../cmd"), the prepending is *not* performed. Instead, a single attempt is made to execute the unmodified command name.

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 interfering with 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. This mechanism may be overridden {4.4}.

3.8 Changing the State of the Shell and the .profile File

The state of a given instance of the Shell may be altered in various ways. The following commands are used more often at the terminal than in Shell procedures.

The *cd* command (or its synonym *chdir*) changes the current directory of the Shell to the one specified. 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 some 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)
```

The *opt* command sets various flags in the Shell. For example, "opt -p prompt-str" changes the Shell's interactive prompt sequence from "% " to *prompt-str*.[4] Typing "opt -v" causes the Shell to enter verbose mode, in which it prints each command line before executing it {4.1}. Try this at the terminal to see how the Shell scans arguments. The output can be turned off by typing "opt +v".

The *login* command causes the Shell to execute the *login* program directly, permitting a new login without re-dialing. A related command is *su*, which permits you to act with someone else's access permissions *without* making you login again.

Wait causes the Shell to suspend execution until all of its child processes have terminated. It is used to assure termination of asynchronous processes.

When you login or use *su*, the Shell is invoked to read your commands, but if your current directory contains a file named ".profile", the Shell reads it *before* reading commands from your terminal; ".profile" often contains commands that set tab stops and terminal delays, read mail, etc. See ".profile" in {6}.

4. The default prompt string "% " is inconvenient for certain display (CRT) terminals.

4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES

4.1 Invoking the Shell

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

sh file [args]	A new instance of the Shell is explicitly invoked to read *file*. Arguments, if any, can be manipulated as described in {4.2}.
sh −v file [args]	This is equivalent to putting "opt −v" at the beginning of *file*. Each command line in *file* is printed before it is executed, thus tracing the progress of execution. *This is an important debugging aid.*
file [args]	If *file* is marked executable, and is neither a directory nor a load module, the effect is that of "sh file [args]", except that *file* may be found by the search procedure described in {3.7}.

4.2 Passing Arguments to the Shell

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 file being read as $0. This notation permits direct reference to the file name and up to 9 arguments. Additional arguments can be processed using the *shift* command. It shifts arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, etc.; the rightmost argument becomes null. For example, consider the (executable) file "ripple" below. *Echo* writes its arguments to the standard output; *if, exit,* and *goto* are discussed later, but perform fairly obvious functions.[5] The form "$1" is used rather than '$1' because it is the value of the first argument that is desired, rather than the literal two-character string "$1":

```
: loop
if "$1" = "" exit
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
shift
goto loop
```

If the file were invoked by "ripple a b c", it would print:

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

The "shift *n*" form of *shift* has no effect on the arguments to the left of the *n*th argument; the *n*th argument is discarded, and the higher-numbered ones shifted. Thus, "shift" is equivalent to "shift 1" (as is "shift 0").

The notation $* causes substitution of *all* current arguments except $0. Thus, the *echo* line in the "ripple" example above could be written in a better way as:

```
echo $*
```

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

```
nroff −h −rT1 −T450 −mm $*
```

It is important to understand the sequence of actions used by the Shell in substituting arguments. First, the Shell reads one line of input, making all substitutions in a single pass; no rescanning is performed. Second, the Shell parses the resulting line. Third, the Shell executes all of the commands in that line. Thus, it is impossible for a command in a line to affect the argument values substituted into that same line. For example, the following sequence prints the same value twice, because the *shift* has no effect on the line in which it appears:

```
echo $1; shift; echo $1
```

5. Much better ways of coding this procedure are shown later. Lines that begin with ":" are labels and/or comments {4.5.1}.

On the other hand, the next sequence prints the first argument, followed by the second:

```
echo $1
shift
echo $1
```

4.3 Shell Variables

The Shell provides 26 string variables, $a through $z. Those in the range $a through $m are initialized to null strings at the beginning of execution and are never modified except by explicit user request. Some variables in the range $n through $z have specific initial values and may possibly be changed implicitly by the Shell during execution. A variable is assigned a value as follows:

```
= letter [ arg1 [ arg2 ] ]
```

If *arg1* is given, its value is assigned to the variable corresponding to *letter*. If two arguments are given, and if *arg1* is a null string, the value of *arg2* is assigned to the variable, permitting a convenient default mechanism. If neither *arg1* nor *arg2* are given, a single line is read from the standard input, and the resulting string (with the new-line character, if any, removed) is assigned to the variable.

The following are examples of simple assignments. You may omit quotes around the arguments if you are sure that they contain no special characters:

```
= a "$1"
= b '*****'
= c /usr/news/.mail
```

The procedure below illustrates the use of a default argument. If an argument is given, mail is read from it. Otherwise, mail is read from "/usr/news/.mail":

```
= a "$1" /usr/news/.mail
mail -f $a
```

The "=" command is often used to capture the output of a program. For example, *date* writes the current time and date to its standard output. The following line saves this value in $d:

```
date | = d
```

This works just as well in longer pipelines. The following saves in $a the number of logged-in users:

```
who | wc -l | = a
```

Another use is in the writing of *interactive* Shell procedures. The following example is part of a procedure to ask the user what kind of terminal is being used, so that tabs and delays can be set and other useful actions taken. The "</dev/tty" indicates a redirection of the standard input to the user terminal; it is *not* seen as an argument to "=", but rather causes the variable to be set to the next line typed by the user:

```
echo 'terminal?'
= a </dev/tty
```

Several variables are currently assigned special meanings:

$n records the number of arguments passed to the Shell, not counting the name of the Shell procedure itself. Thus, "sh file arg1 arg2 arg3" sets $n to 3. Its primary use is in checking for the required number of arguments:

```
if $n -lt 2 then
        echo 'two or more args required'; exit
endif
```

Shift never changes the value of $n.

$p permits alteration of the ordered list of directory pathnames used when searching for commands. It contains a sequence of directory names (separated by colons) that are to be used as search prefixes, ordered from left to right. The current directory is indicated by a null string.

By default, Sp is initialized to a value producing the effect described in {3.7}: ":/bin:/usr/bin". A user could possess a personal directory of commands (say, /a1/tf/jtb/bin) and cause it to be searched *before* the other three directories by using:

```
= p /a1/tf/jtb/bin::/bin:/usr/bin
```

$r gives the value of the return code of the command most recently executed by the Shell. It is a string of digits; most commands return "0" to indicate successful completion. For example, the "=" command returns "0" if two arguments are given and the first is not null, or if a line is actually read from the input. When the Shell terminates, it returns the current value of $r as its own return code.

$s is initialized to the name of the user's *login directory*, i.e., the directory that becomes the current directory upon completion of a login (e.g., "/a1/tf/jtb"). Using this variable helps one to keep full pathnames out of Shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

$t is initialized to the user's terminal identification, a single letter or digit. The terminal can be manipulated using the file name "/dev/tty$t" or just "/dev/tty" alone. The latter is a generic name for the user's terminal.

$w is initialized to the first component of $s, i.e., it is the name of the file system (such as "/a1") in which the login directory is located. Like $s, it is used to avoid pathname dependencies, but is more useful than $s for projects involving many users.

$z is initialized to "/bin/sh". The command named by $z is the one that actually reads the Shell procedures invoked implicitly. The user can alter the choice of the Shell by overriding this value {4.4}. This facility is very useful when there are several different Shells in a system. This may occur because different groups of users want different Shells, or when a new Shell is being tested.

In addition to the above variables, the following read-only variable is provided:

$$ contains a 5-digit number that is the unique process number of the current Shell. Its most common use is in generating unique names for temporary files. Unlike many other systems, PWB/UNIX provides no separate mechanism for the automatic creation and deletion of temporary files: a file exists until it is explicitly removed. Temporary files are generally undesirable objects: the PWB/UNIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occur, especially for multi-user database applications. The following example of $$ usage also illustrates the helpful practice of creating temporary files in a directory used only for that purpose:

```
ls >$s/tmp/$$
... commands    (some of which use $s/tmp/$$)
: 'clean up at end'
rm $s/tmp/$$
```

4.4 Initialization of $p and $z by the .path File

The user may request automatic initialization of each Shell's $p (and $z) by creating a file named ".path" in the login directory. The first (or only) line should be of the form shown for $p {4.3}. If present, the second line should be the full pathname of a Shell. Every instance of the Shell looks for that ".path" file and initializes its own $p (and $z) from it, if ".path" exists. Otherwise, ":/bin:/usr/bin" and "/bin/sh" are the values used, respectively. Thus, the ".path" information is available to all of the user's Shells, but changing $p or $z in one Shell does *not* affect these variables in other Shells. In addition, ".path" is used in a consistent way by commands that must search for other commands, such as *nohup*, *nice*, and *time*.[6] This facility is heavily used in large projects, because it simplifies the sharing of procedures, and can be quickly altered to adapt to changes in organizational requirements.

6. If you plan to write such a command, investigate the *pexec* subroutine, which combines the search and execution code.

4.5 Control Structures

The Shell provides several commands that implement a variety of control structures. These commands are presented here in order of increasing complexity. See {6} for examples of these commands in the context of complete Shell procedures.

☛ *Several of the control commands must not be "hidden" on command lines (e.g., behind semi-colons ";"):*

else end endif endsw if switch while

Other control commands may be "hidden":

break breaksw continue exit goto next

4.5.1 Labels and Goto. The command ":" is recognized by the Shell, but is then treated as a null operation. One use of ":" is to define a label to act as a target for *goto*. Another use is to begin a comment line. However, it is a good idea to place comments in quotes {3.5} if they contain any characters that have a special meaning to the Shell, because the line *is* actually parsed, not just ignored. Using "goto label" causes the following actions:

- A *seek* is performed to move the read pointer to the beginning of the command file.
- The file is scanned from the beginning, searching for ": label", either alone on a line, or followed by a blank or tab.
- The read pointer is made to point at the line *after* the labeled line.

Thus, the only effect of *goto* is the adjustment of the Shell's file read pointer to cause the Shell to resume interpreting commands starting at the line following the labeled line. Invoking *goto* with an undefined label causes termination of the procedure {4.5.5}.

☛ *Avoid the "goto"—future versions of the Shell are not expected to allow it.*

4.5.2 If: Simple Conditional.

if conditional-expression command [args]

Whenever the conditional-expression is found to be *true, if* executes the command (via the *exec* system call), passing the arguments to it. Whenever the conditional-expression is *false, if* merely exits.

The following primaries can be used to construct the 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.
−s file	*true* if the named file exists and has a size greater than zero.
−d file	*true* if the named file is a directory.
−f file	*true* if the named file is an ordinary file.
s1 = s2	*true* if strings *s1* and *s2* are identical.
s1 != s2	*true* if strings *s1* and *s2* are *not* identical.
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".
{ command }	the command is executed; a return code of 0 (yes, zero!) is considered *true*, any other value is considered *false*. Most commands return 0 to indicate successful completion.

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.

All of the operators, flags, and values are *separate* arguments to *if*, and must be separated by blanks. You must be careful to make sure that an argument actually appears and can be parsed correctly:

```
if "$1" = "" echo missing argument
if 0$1 = 0 echo missing argument
if 0"$1" = 0 echo missing argument
```

The first example guards against the possibility that $1 is omitted, null, or has embedded blanks; the second guards against the possibility that $1 has a value that causes parsing problems (such as "-r"), or that it is omitted or null; the third guards against all these problems. The following is dangerous:

```
if $1 = "" echo missing argument
```

because it would cause a syntax error in any of the above cases. Substitution of variables and arguments occurs effectively *before* parsing; thus, for example, if $1 were null, then after substitution the line would read:

```
if = "" echo missing argument
```

In this case, $1 without quotes yields no argument at all (on the other hand, "$1" would have yielded an argument whose value is the null string). It is generally desirable to quote arguments (with double quotes—see 3.5]), especially when they might possibly contain blanks or other characters that have a special meaning to the Shell. Examples of the use of *if* can be found in [6].

4.5.3 If—then—else—endif: Structured Conditional. A more general (and much more readable) form of *if* can be used:

```
if conditional-expression then
        ... commands
else
        ... commands
endif
```

The *else* and the commands following it may be omitted. It is legal to nest *if* commands, but there must be an *endif* to match every *then.*

When *if* is called with a command, using the form of [4.5.2], it acts as described there, deciding whether or not to execute the supplied command. When called with *then* instead of another command, *if* simply exits on a *true*, allowing the Shell to read and interpret the immediately following lines. On a *false, if* reads the file until it finds the next unmatched *else* or *endif*, thus skipping it and any other intervening lines. *Else* reads to the next unmatched *endif. Endif* is a null command.

These commands work together in a way that produces the appearance of a familiar control structure, although they do little but adjust the Shell's read pointer. Be warned that this implementation technique does *not* do a good job of diagnosing extra, missing, or hidden *if, else,* or *endif* commands [4.5]; if you suspect that there are such extra or missing commands, "opt -v" often helps [3.8,4.1].

4.5.4 Switch—breaksw—endsw: Multi-way Branch. The *switch* command manipulates the input file in a way quite similar to *if.* It is modeled on the "switch" statement of the C language [8], and like it, provides an efficient multi-way branch:

```
switch value
: label1
        ... commands
: label2
        ... commands
    :
    :
: default
        ... commands
endsw
```

Switch reads the input until it finds:

- a statement label that matches *value.* The label may contain special characters as described in {3.4}; the method of matching is identical. A few of the many possible labels that could be used to match the value "thing.c" are:

```
thing.c   *.c   t*   *   ???????
```

- *default* used as a statement label (optional).
- the next unmatched *endsw* command.

Again, from the Shell's viewpoint, the only effect of *switch* is to adjust the read pointer so that the Shell effectively skips over part of the procedure, and then continues executing commands following the chosen label or *endsw.* For examples, see ".profile" and "fsplit" in {6}.

Value is obtained from an argument or from a variable; if the label *default* is present, it must be the last label in the list; it indicates a default action to be taken if *value* matches none of the preceding labels. The *switch* construct may be nested; labels enclosed by interior *switch-endsw* pairs are ignored during the execution of *switch. Breaksw* reads the input until the next unmatched *endsw* and is used to end the sequence of commands associated with a label. *Endsw* is a null command like *endif.*

4.5.5 End-of-file and Exit. When the Shell reaches the end-of-file, it terminates execution, returning to its parent the return code found in $r. The *exit* command simply seeks to the end-of-file and returns, setting the return code to the value of its argument, if any. Thus, a procedure can be terminated "normally" by using *exit 0.*

4.5.6 While—break—continue—end: Looping. A *while-end* pair delimits a loop. *Break* can be used to terminate execution of such a loop. *Continue* requests the execution of the next iteration of the loop:

```
while conditional-expression
        ... commands
end
```

While evaluates the conditional-expression, which is similar to that of *if* {4.5.2}. If the conditional-expression is *true, while* does nothing, permitting the following lines to be read and interpreted. If the conditional-expression is *false,* the input file is searched for a matching *end,* and command interpretation resumes with the next line. *While-end* groupings may be nested to a depth of three.

While treats a single, non-null argument as *true* and a single null argument or lack of arguments as *false.* This is convenient for the simple case that handles one argument per iteration:

```
while "$1"
        Do something with $1.
        shift
end
```

Break terminates execution of the smallest enclosing *while-end* group, causing execution to resume after the nearest following unmatched *end.* Exit from *n* levels is obtained by writing *n break* commands on the same line:

```
break; break; ...
```

Continue causes execution to resume at the preceding *while,* i.e., the one that begins the smallest loop containing the *continue.*

4.5.7 Conditional Operators || and &&. These operators enforce left-to-right execution of commands. In the line "cmd1 || cmd2", *cmd1* is executed and its return code examined. Only if it failed (exit code non-zero) is *cmd2* executed. It is thus a more terse notation for:

```
cmd1
if $r -ne 0 then
        cmd2
endif
```

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

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

See "fsplit" and "writemail" in {6} for examples.

4.5.8 Next: Transfer to Another File. The command "next *name*" causes the Shell to abandon the current input and begin reading file *name*. *Next* with no arguments causes the Shell to read from the terminal. By creating a file that initializes Shell variables, then typing "next file" at the terminal, anyone can have a simple shorthand for setting a number of Shell variables with little typing. See "nx" in {6}.

4.6 Onintr: Interrupt Handling

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 *onintr* to obtain the same effects:

```
onintr [ label ]
```

Onintr takes several forms: "onintr label" yields the effect of "goto label" on receipt of an interrupt; "onintr" alone causes normal action to be restored, so that the process terminates on the next interrupt; "onintr –" causes interrupts to be ignored completely, not only by the Shell, but also by any commands invoked by it.

The most frequent use of *onintr* is to make sure that temporary files are removed at the end of a procedure. The example at the end of {4.3} typically would be written as:

```
onintr clean
ls >$s/tmp/$$
        ... commands
: clean
rm $s/tmp/$$
```

When "onintr label" is used, interrupts are effective at the time when the label is reached; it is often desirable to insert another *onintr* following the label. Even so, there may be a short "window" when the user can accidentally kill the procedure by causing repeated interrupts in quick succession.

4.7 Special I/O Redirections

As noted in {3.2}, when the Shell is invoked it expects to inherit from its parent an open standard input (file descriptor 0), standard output (file descriptor 1), and diagnostic output (file descriptor 2). Each of these is initially connected to the terminal.

4.7.1 Standard Input. When the Shell is invoked to read a command file, it saves the old standard input (in an invisible place), then opens the command file as the new standard input. The fact that commands inherit the *new* standard input is convenient for commands that read in-line data (editor scripts, etc.) not read by the Shell. However, this mechanism prevents a Shell procedure from acting as a filter or from reading the *old* standard input in the way that most C programs do. The Shell solves this problem by permitting the notation "<––" to allow a command to take its input from the old standard input, which the Shell has previously saved.[7]

Note that "</dev/tty" and "<––" usually have equivalent effects in a procedure invoked directly from the terminal. The effects differ in a procedure invoked from within another procedure, unless the first procedure takes care to invoke the second with "<––". In any case, "<––" is to be preferred because it can be used to read from a file or a pipe and is thus more general. See "fsplit" and "lower" in {6}.

---

7. The notation "––" arises from the concept of "standard input once removed", because many PWB/UNIX commands accept "–" in place of a file name to indicate that the *current* standard input should be read. This choice makes it impossible to redirect input from a file named "––". Fortunately, file names almost never begin with "–", because many commands expect "–" to signal a flag of some sort.

4.7.2 Standard Output. The use of ">/dev/tty" redirects output to the terminal, even if used in the middle of a pipeline. Shell procedures that act as filters sometimes need to do this. The redirection ">/dev/null" causes the standard output of a command to be thrown into a bottomless pit (presumably to feed the wumpus—see *wump*(VI)). This is used when you want to execute a command for its side-effects, but do not want to be bothered by its output.

4.7.3 Diagnostic Output. Most commands direct diagnostics to file descriptor 2 to make sure that they do not get lost down pipelines. Some situations require that this output go to some place other than the terminal. For example, a long-running procedure may be started, and then the terminal is hung up. In this case, it is helpful to save diagnostics in a file. A deficiency of the current Shell is the lack of syntax for redirecting the diagnostic output. The separate command *fd2* performs the required services:

fd2 [+] [−file] [−−file] command arguments ...

The "+" flag causes diagnostic output to be merged into the standard output. The second option writes that output to *file;* the third appends it to *file.* If the file name is omitted in the second or third cases, "msg.out" is used. If no flag is given, "−msg.out" is assumed.

4.8 Quoting Revisited

The main problem with quoting conventions is the need to treat "$" and "\" in ways flexible enough for convenient use with arguments and variables, but simple enough to be understandable, easy to implement, and unobtrusive in simple cases. In this respect, the current version of the Shell is far from elegant, but is reasonable in practice. The rules are:

- *Inside single* quotes, every character stands for itself without exception. A single quote is *not,* itself, allowed within single quotes.
- *Inside double* quotes, "\$" and "\"" stand for the characters "$" and """, respectively, but with all special meaning removed. All other characters, other than a pair of characters the first of which is an unescaped "$", behave exactly as they do within single quotes, including a "\" *not* followed by a "$" or a """.
- *Inside double* quotes and *outside* either kind of quotes, any two-character sequence whose first character is an unescaped "$" is replaced by the value of the corresponding Shell argument or variable; any variable that has no value (such as "$:") is replaced by a null string.
- *Outside* either kind of quotes, any two-character sequence whose first character is a "\" is replaced by the second character of that sequence, but with any special meaning removed.

4.9 Creation and Organization of Shell Procedures

A Shell procedure can be created in two simple steps. The first is that of building an ordinary text file. The second is that of changing the *mode* of the file to make it *executable,* thus permitting it to be invoked by "name args", rather than "sh name 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 −mm S*
.
w draft
q
chmod 755 draft
```

It may then be invoked as "draft file1 file2". If the Shell procedure "draft" were thus created in a directory whose name appears in the user's ".path" file, 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, then remove it. An alternate approach is that of using *next* to make the current Shell execute the new file, allowing use of existing Shell variables and avoiding the spawning of an additional process for another Shell. In some cases, the need for a temporary file may be eliminated by using the Shell in a pipeline.

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 an ordinary file of 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, 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 of 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 ".path" files that 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}.

5. MISCELLANEOUS SUPPORTING COMMANDS

Shell procedures can make use of almost any command. The commands described in this section are either used especially frequently in Shell procedures, or are explicitly designed for such use.

5.1 Echo: Simple Output

The *echo* command, invoked as "echo [args]", copies its arguments to the standard output, each followed by a single space, except the last argument, which is 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 (as in {3.4}) 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.

Echo recognizes several escape sequences. A "\n" yields a new-line character. *Echo* normally appends a new-line character to its last argument; a "\c" is used to *suppress* that new-line character. The following prompts the user for input and allows input to be typed on the same line as the prompt:

```
echo 'enter name: \c'
= a </dev/tty
```

Echo also recognizes an octal escape sequence for any character, whether printable or not.

5.2 Pump: Shell Data Transfer

Pump is a filter that copies its standard input to its standard output with possible substitution of Shell arguments and variables:

```
pump [ -[ subchar ] ] [ + ] [ eofstr ]
```

Pump reads input until an end-of-file, or until it finds *eofstr* alone on a line. The default *eofstr* is "!". Normally, Shell arguments and variables are substituted in the data stream. The flag "-" suppresses all substitution, while the form "-subchar" causes *subchar* to be used as the indicator character for substitution of Shell variables and arguments, instead of "$". Escaping is handled as in strings enclosed by double quotes—the indicator character may be hidden by preceding it with "\". The "+" flag causes all leading tab characters in the input to *pump* to be eliminated; this permits that input to be indented for readability. A common use of *pump* is to get Shell variables into editor scripts—see "edfind" in {6}, for example. Because editor scripts may use "$" for other purposes, readability may be improved by using a *subchar* such as "%":

```
:       'in file $1, change every instance of $2 to $3'
:       'then delete all lines consisting only of $4'
if -r "$1" then
        pump -% + | ed $1
        g/%2/s//%3/g
        g/^%4$/d
        w
        !
else
        echo "$1: cannot open"
endif
```

Pump is often used to copy a few lines to another file:

```
pump >>logfile
here is $1
and here is $2 on a separate line
!
```

5.3 Expr: Expression Evaluation

Expr supports arithmetic and logical operations on integers, and PL/I-like "substr", "length", and "index" operators for string manipulation. It evaluates a single expression and writes the result to the standard output, typically piped into "=" to be assigned to a variable. Typical examples are:

```
:       'increment $a'
expr $a + 1 | = a

:       'put 3rd through last characters of $1 into $b'
:       'expr substr abcde 3 1000 returns cde (1000 is just a big number)'
expr substr "$1" 3 1000 | = b

:       'obtain length of $1'
expr length "$1" | = c
```

The most common uses of *expr* are in counting for loops and in using "substr" to pick apart strings.

5.4 Logname, Logdir, Logtty: Login Data

When a user logs in, he or she supplies a *login name* and a password. The *login* program searches the password file for that *login name* and obtains the name of the program to be executed by the user (normally the Shell), the directory to be made the current directory, and also a *userid,* a value ranging from 0 to 255. Most UNIX protection and identification mechanisms utilize the last item. Limiting the number of distinct users to 256 is no problem for most UNIX systems, but the original PWB/UNIX installation currently supports more than 1,000 users. However, it is not necessary to provide a distinct userid for every user. Project-oriented groups of users often choose to share one or two userids, in order to ease the problems caused by personnel absences, and also to ease the manipulation of shared files.[8] Although the members of such groups do not generally worry about being protected from each other, they need to be identified as distinct individuals by some programs, i.e., those that tag inter-user messages with user names or log the name of the user making a change to a source program. PWB/UNIX records the login name instead of discarding it after login. The *logname* command writes this name to the standard output, allowing it to be captured in a Shell variable. It can then be used to permit only selected users to execute a procedure, or can be included in logging information:

```
logname | = u
(echo "$u updated files on \c"; date) >>projectlog
```

The *logdir* and *logtty* commands are used in the same way as *logname;* they produce the same values as the initial values of Ss and St, respectively {4.3}.

8. Although some groups started by using one userid per person, it was discovered that these users often shared a single password. Thus, the possession of separate userids was considered more of a hindrance than a help.

6. EXAMPLES OF SHELL PROCEDURES

☛ *Some examples in this section may be quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name.*

.profile:

```
:       '.profile (automatically invoked on login) asks for terminal type,'
:       'reads a line from terminal, loops until a known type'
:       '(or empty line) is entered, sets terminal options appropriately,'
:       'asks for new directory name and changes to it, if one is given,'
:       'and then, if file nx exists, transfers to it'
while 1
        echo 'terminal:\c'
        = a </dev/tty
        switch "$a"
        :       'DASI450'
        : 450
                stty cr2; tabs +t450; break
        :       'GSI/DASI300'
        : gsi
        : 300
                stty cr2; tabs; break
        :       'HP264X'
        : hp
                stty cr0 nl0; tabs +thp; break
        :       'TI 700'
        : ti
                stty -tabs nl1 cr1; break
        : default
                if 0"$a" = 0 break
                echo "$a? try 450,gsi,hp,ti"
        endsw
end
echo "cd \c"
= b </dev/tty
if "$b" != "" then
        cd $b
endif
if -r nx then
        next nx
endif
```

Note: *Break* is used instead of *breaksw* in the above example to terminate the *while* loop, not just the *switch* construct.

copypairs:

```
:       'copypairs file1 file2 ...'
:       'copy file1 to file2, file3 to file4, ...'
while "$2"
        cp $1 $2
        shift; shift
end
if 0"$1" != 0 echo 'odd number of arguments'
```

Note: Remember that "shift; shift" is *not* the same as "shift 2". See next example for use of "shift 2".

copyto:

```
:       'copyto dir file ...'
:       'copy argument files to dir, making sure that at least'
:       'two arguments exist, that dir is a directory, and that'
:       'each additional argument is a readable file'
if $n -lt 2 then
        echo 'usage: copyto directory file ...'; exit
endif
if ! -d $1 then
        echo "$1 is not a directory"; exit
endif
while "$2"
        if ! -r $2 then
                echo "$2 not readable"
        else
                cp $2 $1
        endif
        shift 2
end
```

distinct1:

```
:       'distinct1'
:       'reads standard input, reports list of identifiers 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. The *tr* translates all characters except letters and digits into new-line characters, and then "squeezes out" repeated new-line characters. This leaves each identifier (in this case, any contiguous sequence of letters and digits) on a separate line. *Sort* 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 >tmp1; <tmp1 cmd2 >tmp2; <tmp2 cmd3; rm tmp[1-3]
```

Starting with a file of test data and working from left to right, each command is run taking its input from the previous file and putting its output in the next file. The final output file 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 "distinct1" 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.

distinct2:

```
:       'distinct2'
:       'reads standard input, reports sorted list of identifiers that differ'
:       'in case only, listing all such distinct identifiers'
onintr cleanup
tr -cs '[A-Z][a-z][0-9]' '[\012*]' <-- | sort -u | tee t1$$ | tr '[A-Z]' '[a-z]' >t2$$
pr -s -t -l1 -m t1$$ t2$$ | sort +1 >t3$$
:       'third argument to pr in above line is "minus ell one"'
sort t3$$ >t4$$
uniq -u -1 t3$$ | sort | comm -23 t4$$ - | sort +1
: cleanup
rm t?$$
```

Note: This procedure is similar to the previous one, but provides more explicit information. As an exercise, work through this procedure in the way described above. The commands used here (plus *grep* and *sed*) form the basis for many "data stream" operations.

draft:

```
:       'draft file ...'
:       'prints the draft (-rC3) of a document on a DASI450 terminal in 12-pitch'
:       'using PWB/MM'
nroff -rC3 -T450-12 -mm $*
```

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:

```
:       'edfind file arg'
:       'find the last occurrence in file of a line that matches arg,'
:       'then print 3 lines (the one before, the line itself, and the one after)'
pump | ed - $1
?$2?;-,+p
!
```

Note: This illustrates the typical practice of using *pump* to substitute Shell variables into *ed* scripts.

edlast:

```
:       'edlast file'
:       'prints the last line of file, then deletes that line'
ed - $1
$p
$d
w
q
echo done
```

Note: This procedure illustrates the effects of a command that reads input from a file shared with the Shell.

fsplit:

```
:       'fsplit file1 file2'
:       'read standard input and split it into three parts:'
:       'append any line containing at least one letter to file1, any line'
:       'containing digits but no letters to file2, and throw the rest away'
= i 0; = j 0
while 1
        = a <-- || break
        expr $i + 1 | = i
        switch "$a"
        : *[A-Za-z]*
                echo "$a" >>$1; breaksw
        : *[0-9]*
                echo "$a" >>$2; breaksw
        : default
                expr $j + 1 | = j
        endsw
end
echo "$i lines read, $j thrown away"
```

Note: Each iteration of the loop reads a line from the input and analyzes it. The *break* terminates the loop only when "=" 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}.

loop:

```
:       'loop arg ...'
:       'one or more command lines'
:       'endloop'
:       'execute the group of command lines once for each argument,'
:       'substituting each argument as $1 in the command lines'
onintr cleanup
echo 'while "$1"' >tmp$$
pump - + endloop <-- >>tmp$$
echo 'shift \n end' >>tmp$$
next tmp$$; rm tmp$$
: cleanup
rm tmp$$
```

Note: Such a procedure is typically used from a terminal to repeat some commands for a list of arguments. It creates a temporary file that sandwiches user input between a *while* and *shift-end.* It then transfers to that file. For example, all files in the current directory could be copied to "place" by:

```
loop *
cp $1 place
echo $1 copied
endloop
```

lower:

```
:       'lower'
:       'reads standard input, converts it to lower case, writes to standard output'
:       'can thus be used in a pipeline if desired'
tr '[A-Z]' '[a-z]' <--
```

Note: This is the most common type of use for "<--".

mkfiles:

```
:       'mkfiles prefix [number]'
:       'makes number (default = 5) files, named prefix1, prefix2, ...'
=. a "$2" 5
= i 1
while $i -le $a
        cp /dev/null $1$i
        expr $i + 1 | = i
end
```

null:

```
:       'null file ...'
:       'create each of the named files as an empty file'
while "$1"
        cp /dev/null $1
        shift
end
```

nx:

```
:       'next nx'
:       'asks for module name, initializes variables to useful values,'
:       'prints variables.  Note that variables are set within the invoking Shell,'
:       'so nx can be invoked only from terminal or from .profile'
= a /sys/source/s1
= b /usr/man/man1
echo "m: \c"
= m </dev/tty
= g "get -e s.$m; ed $m"
= d "delta s.$m"
pump
a: $a    b: $b
d: $d    g: $g    m: $m
!
next
```

phone:

```
:       '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
```

writemail:

```
:       'writemail message user'
:       'if that user is logged in, write message on terminal;'
:       'otherwise, mail it to that user'
echo "$1" | ( write "$2" || mail "$2" )
```

Note: Replacing "echo" above by "pump . <--" writes or mails the standard input, in the same way as 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 author's 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 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 spawning processes. The CPU overhead per process lies in the range of 0.07 to 0.1 seconds, depending on the specific hardware configuration. 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:

:	chdir	endsw	newgrp	shift
=	continue	exit	next	switch
break	else	goto	onintr	test
breaksw	end	if	opt	wait
cd	endif	login	pump	while

Pump actually executes as a child process, i.e., the Shell does a *fork*, but no *exec;* "()" executes in the same way. 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. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by:

$$\text{processes} = k*n + c$$

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 procedures whose complexity measures include n^2 terms or higher powers of n are 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:

fsplit2:

```
onintr cleanup
= b '[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]'
cat <-- >tmp$$
grep "$b" tmp$$ >tmp$$1
grep -v "$b" tmp$$ | grep "[0123456789]" >tmp$$2
cat tmp$$1 >>$1 ; cat tmp$$2 >>$2
wc -l <tmp$$ | = i
wc -l <tmp$$1 | = j
wc -l <tmp$$2 | = k
expr $i - $j - $k | = a
echo "$i read, $a thrown away"
:       cleanup
rm tmp$$*
```

This version is often ten times faster than "fsplit", and it is even better 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 Bytes of Data 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 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 pathnames. Judicious use of *cd* can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system).[9]

```
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 File. The ".path" file is a popular and convenient mechanism for organizing and sharing 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 pathname that precedes the needed pathname in the current $p variable. As an example, consider the effect of invoking *nroff* (/usr/bin/nroff) when $p 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" can increase this number significantly.

The vast majority of command executions are of commands found in "/bin" and, to a 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 (at least with regard to efficiency):

9. You may have to do some reading in the *PWB/UNIX User's Manual* [3] to understand exactly what is going on in these examples.

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

The first one above should be avoided. The others are acceptable—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 changing Sp to resemble the last of the above four examples.

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 254 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.

ACKNOWLEDGMENTS

The Shell was originally written by K. Thompson; its basic structure has remained unchanged since then, although many features (and some warts!) have been added. The PWB/UNIX extensions were added by R. C. Haight, A. L. Glasser, and the author. Some constructs have been derived from similar ones in the recent Shell written by S. R. Bourne. A number of colleagues provided helpful comments during the writing of this tutorial; T. A. Dolotta, in addition, provided a great deal of editorial assistance. Finally, many thanks must go to the PWB/UNIX user community, and especially M. H. Bianchi and J. T. Burgess, who provided many suggestions and 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] 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.

[3] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[4] Kernighan, B. W., and Plauger, P. J. Software Tools. *Proc. First National Conference on Software Engineering,* pp. 8-13, Sept. 11-12, 1975.

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

[6] Mashey, J. R. Using a Command Language as a High-Level Programming Language. *Proc. Second Int. Conf. on Software Engineering,* pp. 169-76, Oct. 13-15, 1976.

[7] Mashey, J. R. PWB/UNIX Documentation Roadmap. Bell Laboratories, 1977.

[8] Ritchie, D. M. C Reference Manual. Bell Laboratories, 1977.

[9] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* 17(7):365-75, July 1974.

[10] 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.

[11] Thompson, K., and Ritchie, D. M. *UNIX Programmer's Manual—Sixth Edition.* Bell Laboratories, May 1975.