Using a Command Language as a High-Level Programming Language

J. R. Mashey

Bell Laboratories
Piscataway, New Jersey 08854

Keywords: Command languages, command interpreters, UNIX.

Abstract: The command language for the Programmer's Workbench (PWB) utilizes an extended version of the standard UNIX shell program, plus commands designed mainly for use within *shell procedures* (command files). Modifications have been aimed at improving the use of the shell by large programming groups, and making it even more convenient to use as a high-level programming language. In line with the philosophy of much existing UNIX software, an attempt has been made to add new features only when they are shown necessary by actual user experience in order to avoid contaminating a compact, elegant system through "creeping featurism." By utilizing the shell as a programming language, PWB users have been able to eliminate a great deal of the programming drudgery that often accompanies a large project. Many manual procedures have been quickly, cheaply, and conveniently automated. Because it is so easy to create and use shell procedures, each separate project has tended to customize the general PWB environment into one tailored to its own requirements, organizational structure, and terminology. A summary is given of the usage patterns revealed by a survey of 1,725 existing shell procedures.

1. INTRODUCTION

A good operating system command language (CL) is an invaluable tool for large programming projects. A common occurrence in such projects is the diversion of significant resources from building the end product to creating internal support programs and procedures. If a CL 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 do the job *now*.

The Programmer's Workbench (PWB) environment [IVI75A, DOL76A] supports projects ranging in size from several people to those involving hundreds. The PWB CL is a modified version of the UNIX *shell* language [RIT74A, THO75A]. Experience with it has shown it to be an effective tool for automating procedures and eliminating programming drudgery. Nearly two thousand CL procedures have been inspected to observe real-life usage. Many of the assertions made in this paper are based on the results of this survey.

Existing CLs include a wide range of structures and abilities, ranging from those quite similar to assembly language to a few having the appearance and abilities of high-level programming languages [BRU76A, COW75A]. A diversity of viewpoints exists regarding the relative importance of various CL characteristics [DOL69A, SIM74A, UNG75A]. Several years of PWB experience suggests that the following qualities are valuable in a CL:

- *Ease of on-line use*—Many advantages can be gained by replacing keypunches with terminals, even when jobs are prepared for batch execution. To support this approach, CL syntax should emphasize simplicity and avoid the need for redundant typing.
- *Convenient use of the same language as an on-line CL and as a programming language*—It must be possible to build and store sequences of commands (*CL procedures*) that can be invoked at a later time.
- *Interchangeability of programs and CL procedures*—No user should need to know whether a given command is implemented as a compiled, executable program or as a CL procedure. Syntactic differences should be avoided.
- *Ease of creation and maintenance of CL procedures*—It should require very little effort to write a CL procedure, save it for later use, and maintain it. The overhead of writing a small procedure should be especially small; as will be seen later, many procedures are only a few lines long.
- *Organization of CL procedures*—It must be possible to share easily the use of procedures among people in a way consistent with organizational structure.
- *Programming language features*—It must be possible to write procedures with convenient conditional branching, looping, argument handling, variables, string manipulation, and occasional arithmetic. Thus, many of the capabilities of a typical procedural language are *necessary* for a good CL. However, they may not be *sufficient:* a CL may need additional capabilities, a somewhat different syntax, and a radically different set of priorities concerning the importance of various constructs. For example, a crucial CL capability is that of connecting executing programs in a variety of ways. Pattern-matching and other string manipulation operations are quite useful. Arithmetic operations are also helpful, but seem to be much less important than the others.
- *Separation from operating system*—A CL interpreter should normally be an ordinary command, treated like other commands, and definitely *not* embedded in the heart of the operating system. The operating system should *not* be viewed as an implementation of the CL, but as an environment that can support a variety of good CLs. Different users should be able to have different CLs if they feel like it. This approach permits experimentation and evolution without bothering other users. It often leads to a "survival-of-the-fittest" behavior. Mutations occur, live, and die on their merits, and eventually breed hybrids containing features found useful from actual practice.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of some later discussions depends on familiarity with UNIX. [RIT74A] is a definite prerequisite, and it would be helpful to read at least one of [KER75A, KER76A, THO75A]. 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 *file name* is a sequence of characters. A *path name* is a sequence of directory names followed by a file name, each separated from the previous one by a slash ("/"). 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.* (The first type of name is often called an *absolute path name* because it is invariant with regard to the user's current directory.) The user may change current directories at any time by using the *chdir* command. In general, file names and path names can be used in the same ways. Some sample names are:

/	root of the entire file structure.
/bin	directory of commonly used public commands.
/u0/tnds/tf/jtb/bin	a path name typical of multi-person programming projects. This one is a private directory of commands belonging to person "jtb" of group "tf" in project "tnds".
bin/umail	a name depending on the current directory: it refers to file "umail" found in subdirectory "bin" of the current directory. If the current directory is "/", it names "/bin/umail". If the current directory is "/u0/tnds/tf/jtb", it names "/u0/tnds/tf/jtb/bin/umail".

Large projects require the ability to quickly and easily modify directory structures to fit changing needs. In particular, the "current directory" feature makes it possible for each person to move around in the file system and work where most convenient. This allows simple names to be used, even when the current directory is many levels deep in the structure. It also permits individual directories to remain fairly small, lessening the load on both human and computer; i.e., "locality of reference" is good for the performance of *both.*

2.2 Processes and Their Interactions

An *image* is a computer execution environment, including core 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 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 may continue execution of the image, or may abandon it by issuing an *exec* system call, which initiates execution of another program.

Processes use independent address spaces and data segments,[1] and communicate in a limited number of wavs:

- *Open files*—a child inherits the parent's open files, and can manipulate the associated read/write pointers thus shared with the parent. This ability permits processes to share the use of a common input stream in various useful ways. In particular, an open file possesses a *pointer* that indicates a location in the file, and is modified by various operations. *Read* and *write* copy a requested number of bytes from (to) a file, beginning at the location 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 location within the file, or to a location offset from the end of the file or the current pointer location.
- *Arguments*—a sequence of arguments (character strings) can be passed from one program to another via *exec.*
- *Return code*—when a process terminates, it can set a numeric return code that is available to the process's parent.
- *Files*—some programs arrange conventions to share files in various ways, or to use files of specified names.
- *Pipes*—pipes are interprocess channels that are similar to files in ways of access, but allow very convenient handling of the "producer-consumer" relationship between programs executing in parallel. The "producer" writes into one end of a pipe, while the "consumer" empties it by reading from the other end. Because UNIX handles details of buffering and synchronization, neither program needs explicit information about the other's activities.

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 functions into small programs that are easily connected. The pipe mechanism is especially desirable, both for human comprehension and for computer performance [THO75A, KER75A, KER76A].

3. SHELL BASICS

Most UNIX users utilize the CL provided by a program called the *shell.* It reads input from a terminal or file and arranges for the execution of the requested commands. The shell is a small program (about 20 pages of C code); many CL functions are actually supported by independent programs that work with the shell, but are not built into it. The discussion is adapted from [THO75A, THO75B].

3.1 Commands

A *command* is a sequence of non-blank arguments separated by blanks. The first argument specifies the name of the command to be executed; the remaining items are passed as arguments to the command executed. 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 marked as *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 *command file* (*shell procedure*). A command file is a file of ordinary text—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.

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. Most operating systems can execute existing load modules without requiring the user to state the source language used to produce the load module. CL procedures are treated in the same way, for several reasons. First, the existence of two distinct types of commands is confusing to the novice user. Second, *any* user becomes irritated when forced to type repetitive information, especially when the system already has it. Finally, the implementation of a given command may well change with time, typically from shell procedure to compiled program. This change could cause great pain to the users if it required the invocation method to change also.

3.2 Finding Commands

The shell normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. It first attempts to use the command name without modification, then prepends the string "/bin/" to the name, and then "/usr/bin/". If the original command name is a simple one, the effect is to search in order the current directory, "/bin", and "/usr/bin". A more complex path name may be given, either to locate a file relative to the user's current directory, or to access one via an absolute path name.

This mechanism gives the user convenient execution of public commands and of commands in or "near" the current directory, as well as the ability to execute any accessible command, regardless of location in the file structure. The search order permits a standard command to be replaced by a user's command without affecting anyone else.

1. The text segment of a reentrant program is shared by all processes executing that program. Almost all programs are reentrant.

2. As shown by a set of flag bits associated with the file.

3.3 Command Lines

A series of commands separated by "|" 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 −mm text | col | reform
```

Nroff is a text formatter whose output may contain reverse line motions; *col* converts them to a form that can be printed on a terminal lacking reverse motion, and *reform* is used here to speed printing by converting the (tab-less) output of *col* to one containing horizontal tab characters. The flag "−mm" indicates one of many possible formatting options, and "text" is the name of the file to be formatted.

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 command line 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 the previous example.

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

If the last command in a pipeline is terminated by a semicolon (";") or new-line, the shell waits for the command to finish before continuing execution. It does not wait if the command is terminated by an ampersand ("&"); both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily, or until it is *killed* (by one of various means). A *command line* consists of zero or more pipelines separated by semicolons or ampersands.

For example, the following command line is used to run timing tests on an empty system. *Makeload* is a cyclic shell procedure used to generate a heavy, repeatable load of disk accesses, and *test1* performs timing tests on various programs. The shell runs *test1* with no load on the system, then starts one *makeload* to create a single unit of disk load for the second *test1.* Another *makeload* is invoked to yield two units of load for the last *test1.*

```
test1; makeload &  test1; makeload &  test1
```

Each *makeload* runs until explicitly killed by the user. A minimum of three processes are active by the time the final *test1* is run (two *makeloads* and one *test1*). In this particular case, all commands are implemented as shell procedures, so there is a separate invocation of the shell for each of the five commands on the above line, and each shell may well spawn hundreds of additional processes. Thus, a single user may consume all system resources by creating large numbers of long-lived asynchronous processes.[3] More typical uses of "&" include off-line printing, background compilation, and generation of large jobs to be sent to remote computers.

3. Lockout of other users in this way occurs several times per year on PWB systems; it is usually caused by overly enthusiastic beginning users.

3.4 Redirection of Input and Output

When a command begins execution, it usually expects three files to be already open, a "standard input," a "standard output," and a "diagnostic output." When the user's original shell is started, all three are 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 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.

These forms of I/O redirection complement that of piping: files and programs can both be used as data "sources" and "sinks."

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal, file, or pipe. Commands are thus easily used 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 or pipe).

3.5 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 directories. Most characters match themselves. The "?" matches any single 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 single one of the characters in that set. Inside 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 names in the current directory, "*temp*" matches any names containing "temp", "[a−f]*" matches all names beginning with "a" through "f", and "/u0/tnds/tf/bin/?" matches all single-character names found in "/u0/tnds/tf/bin".

This capability saves much typing, and more importantly, provides convenience in organizing files for various purposes. It allows convenient use of large numbers of small files.

3.6 Quoting Mechanisms

If a character has a special meaning to the shell, that meaning may be removed by preceding the character with a backslash ("\"); the "\" acts as an escape and disappears. Sequences of characters enclosed in double (") or single (') quotes are in general taken literally, except that substitution of shell arguments and variables is normally performed.

A "\" followed by a new-line is treated as a blank, permitting convenient continuation of multi-line commands.

4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES

4.1 Invoking the Shell

The shell may be invoked explicitly in various ways:

sh −	The new shell reads the standard input, but does not prompt. This is often used to let the shell act as filter, i.e., it can be used in a pipeline to read and execute a dynamically-generated stream of commands.

sh file [args]	A new instance of the shell is created to begin reading the file. Arguments can be manipulated as described in the next section.
file [args]	As noted in Section 3.1, if the file is marked executable, and is neither a directory nor a load module, the effect is the same as "sh file [args]".

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. A procedure may possess several different names and can check $0 to determine the specific name being used, then vary its actions accordingly.

This notation permits up to 10 arguments to be referenced. 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. For example, consider the file "loopdump" below. *Echo* writes its arguments to the standard output; *if, exit,* and *goto* are discussed later, but perform fairly obvious functions.

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

If the file is invoked by "loopdump a b c" it would print:

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

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

4.3 Shell Variables

Adding *if* and *goto* commands (described later) to the existing facilities permits convenient expression of some kinds of procedures: repetitive ones that perform a given set of actions for each argument and those that use simple conditional logic. Clear expression of many procedures requires at least a few shell variables.

The PWB shell provides 26 string variables, $a through $z. Those in the first half of the alphabet are guaranteed to be initialized to null strings at the beginning of execution and are never modified except by explicit user request. On the other hand, some variables in the range $m through $z have specific initial values, and may possibly be changed implicitly by the shell during execution. As will be seen later, few shell procedures ever use more than a few variables. A variable is given a value as follows:

```
= letter [argument]
```

If *argument* is given, its value is assigned to the variable given by *letter.* As an example of common usage, the procedure below expects to be called with a list of file names, optionally preceded by a flag "−w". If the first argument is "−w", the fact is recorded by setting $a to "w", and the argument is shifted off the argument list, leaving only file names. If the first argument is not "−w", $a is left unchanged, i.e., it is a null string.

```
if "$1" = −w then
        = a w
        shift
endif
... code to process file names, using $a as needed
```

If no argument follows the letter in the "=" command, a single line is read from the standard input, and the resulting string (with the trailing new-line, if any, removed) becomes the value of the variable. A common use is 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
```

Thus, $d would be set to a value such as:

```
Tue Jul 13 19:06:02 EDT 1976
```

A second use is in writing of *interactive* shell procedures, which are heavily used in PWB work. The following example is part of a procedure to ask the user what kind of terminal is being used, so that tabs can be set, delays changed, and other useful actions taken. The "</dev/tty" indicates a redirection of the standard input to the user's 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.

```
: loop
echo "terminal?"
= a </dev/tty
if "$a" = ti goto ti
if "$a" = hp goto hp
        echo "$a no good; try ti or hp"
        goto loop
: hp
... processing for terminal type "hp"
exit
: ti
... processing for terminal type "ti"
```

Currently, five variables are 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. *Shift* never changes the value of $n.
$p	permits alteration of the names and order of directory path names used when searching for commands. It contains a sequence of directory names (separated by colons) that are to be used as search prefixes, in order from left to right. The current directory is indicated by a null string. The string is normally initialized to a value producing the effect described in Section 3.2: ":/bin:/usr/bin". A user could possess a personal directory of commands (e.g., "/u2/pw/bin") and cause it to be searched before the other three directories by using: `= p /u2/pw/bin::/bin:/usr/bin`
$r	gives the value of the return code of the most recent command executed by the shell. 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. Users can avoid embedding unnecessary absolute path names in their procedures by using this variable. This is of great benefit when path names are changed either to balance disk loads or to reflect project organizational changes.
$t	is initialized to the user's terminal identification, a single letter or digit.

In addition to these variables, the following is provided:

$$	contains a 5-digit number that is the unique process number of the current shell. In some circumstances, it is necessary to know the number of a process, in order to *kill* it, for example. However, its most common use to date has been that of generating unique names for temporary files.

4.4 Extended Order of Search for Commands

The user may request automatic initialization of each shell's $p by creating a file named ".path" in the login directory. This file should contain a single line of the form shown for $p. Every instance of the shell looks for this file and initializes its own $p from it, if it exists; otherwise ":/bin:/usr/bin" is used. Thus, the ".path" value propagates through all of the user's shells, but changing $p in one shell has no effect on the $p of any other.

This facility is heavily used in large projects. It greatly simplifies the sharing of procedures, and can be quickly changed to adapt to changing organizational requirements.

4.5 Control Structures

The more complex shell control structures are actually implemented as independent commands that cooperate with the shell, but are not actually part of it. They are designed specifically for use in shell procedures, but are treated as ordinary commands. This separation of function allows the shell to remain a small program, efficient for on-line use, but still able to support powerful control structures in procedures.

4.5.1 Labels and Goto. The command ":" is recognized by the shell and treated as a comment. The most common use of ":" is to define a label to act as a target for *goto*. *Goto* is a separate command. Using "goto label" causes the following actions:

1. A *seek* is performed to move the read/write pointer to the beginning of the command file.
2. The file is scanned from the beginning, searching for ": name" on a line, either alone or followed by a blank or a tab.
3. The read/write pointer is adjusted (via the *seek)* to point at the line following the labeled line.

Thus, the only effect of *goto* is the adjustment of the shell's file read/write pointer to cause the shell to resume interpreting commands starting at the line following the labeled line.

4.5.2 If.

```
if expr command [args]
```

If is also a separate command. If the conditional expression *expr* is found to be true, *if* executes the command (via *exec* system call), passing the arguments to it. If it is false, *if* merely exits.

The following primaries are used to construct the expression:

−r file	true if the file exists and is readable by the user.
−w file	true if the file exists and is writable by the user.
s1 = s2	true if strings *s1* and *s2* are equal.
s1 != s2	true if the strings are not equal.
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 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 (as \(or ")", for example) to remove their significance to the shell.

All of the operators, flags, and values are *separate* arguments to *if,* and must be separated by blanks. The following are typical argument-testing operations:

```
:       check a file argument to make sure it exists
if ! −r "$1" echo "can't read $1"

:       assure either that:    $1 is a and $3 is either b or c
:                 or that:     $1 is d and $2 is e
if "$1" = a −a "(" "$3" = b −o "$3" = c ")" \
        −o "$1" = d −a "$2" = e goto legal
```

Recall that the effect of the "\" at the end of the line is that of a blank. It is generally desirable to quote arguments when they might possibly contain blanks or other characters having special meaning to the shell.

4.5.3 If-then-else-endif. To improve the readability and speed of shell procedures, *if* was extended to provide the "if-then-else-endif" form. The general form is:

```
if expr then
        ... commands
else
        ... commands
endif
```

The "else" and the commands following it may be omitted, and it is legal to nest if's within the commands.

When *if* is called with a command, using the form of Section 4.5.2, it acts as described there, directly executing (or not) the supplied command. When called with *then* instead of a command, *if* simply exits on a true condition, allowing the shell to read (and interpret) the immediately following lines. On a false condition, *if* reads the file until it finds the next unmatched *else* or *endif,* thus skipping it and the lines in between. *Else* reads to the next unmatched *endif,* and *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 read/write pointers.

4.5.4 Switch-breaksw-endsw. The *switch* command manipulates the input file in a way quite similar to *if.* It is modeled on the corresponding "switch" statement of the C language [RIT75A], 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:

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

Again, from the shell's viewpoint, the only effect of *switch* is to adjust the read/write pointer so that the shell effectively skips over part of the procedure, then continues executing commands following the chosen label or *endsw.*

Value is obtained from an argument or variable; if the label "default" is present, it must be the last label in the list; i.e., it indicates a default action to be taken if *value* matches none of the preceding labels. This construct may be nested; labels enclosed by interior "switch-endsw" pairs are ignored.

The command *breaksw* reads the input until the next unmatched *endsw,* and commonly ends 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". The fact that *exit* is *not* part of the shell permits straightforward use of it as an argument for *if.*

4.5.6 The Missing Loop. Conspicuous by its absence is some form of *while* or *do.* All of the control structures described so far are implemented outside the shell; it appears that any useful looping construct requires significant changes to the shell itself. In any case, the most frequently observed kind of loop is that used to process arguments one at a time. For example, the following applies the first argument as a command to every remaining argument:

```
: loop
if 0$2 = 0 exit
$1 $2
shift 2
goto loop
```

The "2" causes *shift* to leave the first argument in place.

4.5.7 Transfer to Another Command File—Next. The command "next name" causes the shell to abandon the current procedure and begin reading from file *name. Next* with no arguments causes the shell to read from the user's terminal. The idea of *next* is to permit the use of a file to initialize variables for use at the terminal:

```
= a /u2/pw/mash/articles
= b "nroff -rT2 -mm"
...
next
```

If this text were stored in file "init", it could be invoked by using "next init", causing the current shell to process it and return to the terminal. The user can then reference $a and $b appropriately. The user could of course use "=" to accomplish this directly, but at the cost of more typing.

Next is an attempt to obtain an effect like that of a subroutine call with a shared environment. It handles some problems well, but will probably be changed somewhat to make it more useful. Its most common application has actually been in very complex procedures that analyze their arguments, set up variables, then pass control to one of several successor procedures.

4.6 Interrupt Handling in Shell Procedures

Many PWB users have taken advantage of the ease and speed of writing shell procedures to automate various operations. In many cases, such procedures need to be used by clerical personnel who have no knowledge of these procedures' inner workings. A terminal interrupt (depression of "rubout" or "del" key) can be ignored or intercepted by a compiled program, or can cause termination of that program. The lack of interrupt-handling facilities in the shell quickly led to the usual problems:

- No procedure could use a terminal interrupt as a control mechanism.
- Any procedure that created files for temporary use left them in existence if interrupted before it could remove them. In practice, any procedure that prints very much information is likely to be interrupted sooner or later.
- Some procedures need to temporarily ignore interrupts so they can guarantee consistency among files making up data bases. The PWB supports a profusion of packages that consist of file groupings accessed only through shell procedures.

The *onintr* command was added to solve these problems. It takes three forms: "onintr label" causes the effect of "goto label" to occur upon receipt of an interrupt; "onintr -" causes interrupts to be ignored completely; "onintr" alone causes normal interrupt action to be restored. A typical use of *onintr* is:

```
onintr cleanup
:       create temporary file
ls -l | tee temp$$a | grep -c "^d" | = d
grep -c "^-" temp$$a | = f
echo "directories: $d, files: $f"
: cleanup
rm temp$$a
```

This procedure displays the numbers of subdirectories and ordinary files in the current directory. The output of the *ls* command is a listing of the current directory; it is passed to *tee,* which makes an extra copy of it in "temp$$a", but also passes it to *grep,* which, in this instance, counts the number of lines whose first character is "d". This is the number of subdirectories, and is saved in variable $d. The ordinary files (whose listing entries begin with "-") are counted in a similar way, and both counts are displayed. If the process is interrupted by the user, control transfers to "cleanup", where the temporary file is removed.

4.7 Additional Supporting Commands

4.7.1 Evaluation of Expressions. Expr supports arithmetic and logical operators 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

:       strip off first 2 chars. of $1 and put result in $b
:       expr substr abcde 3 999 returns cde
expr substr "$1" 3 999 | = b

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

The most common uses of *expr* are counting (for loops) and using "substr" to pick apart strings (such as the output from *date,* as in Section 4.3).

4.7.2 Echo. The *echo* command, invoked as "echo [args]", copies its arguments to the standard output, each separated from the preceding one by a blank, with a new-line appended to the last argument. It is often used to perform prompting or issue diagnostics in shell procedures, to add a few lines to an output stream in the middle of a pipeline, and to create editing scripts; "\n" yields a new-line and "\0n" yields the ASCII character given by the octal number *n;* "\c" is used to get rid of unwanted new-lines. For example, the following code prompts the user for some input and allows the user to type on the same line as the prompt:

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

4.8 Creation of Shell Procedures

A shell procedure can be created in two simple steps. The first step is that of building an ordinary text file, normally using the UNIX text editor *ed.* The second step 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", as discussed in Section 3.1. The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for longer-lived procedures.

The following shows the *entire* user input needed to set up a simple procedure to format text files according to a standard format and print the output on a particular type of terminal:

```
ed
a
nroff -rT1 -rC3 -mm $1 $2 $3 $4 $5 $6 $7 $8 $9 | gsi +12
.
w draft
q
chmod 755 draft
```

In the sequence above, the user called the text editor, entered a single line of text, wrote that line (creating the new file "draft"), and finally changed its *mode* to make it executable. The user may then invoke this command as "draft file1 file2", for example. The procedure calls the formatter with certain fixed arguments and any others supplied by the user; the formatter output is passed to *gsi* to convert it to a form that is appropriate for the user's terminal (in this case, a GSI-300).

If the sequence above were performed in a directory included in the user's ".path" file, the user could change directories and still use the "draft" command. Other people might make use of it also, especially if it were placed in a shared directory of commands.

The command sequence above could itself be stored as a shell procedure, although this particular sequence is an unlikely candidate for such an action. Note that the five lines following the *ed* call are processed by *ed* rather than the shell. It is quite reasonable to include data for other programs inside shell procedures, as long as those programs are careful in their method of reading, i.e., do not read beyond their own data. This method has beneficial results for performance, because I/O buffers can be shared without need for separate temporary files.

Shell procedures may be dynamically created by other shell procedures. 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 command file, allowing use of existing values of shell variables and avoiding the spawning of an additional process for another shell. In some cases, the use of a temporary file may be eliminated by using the shell in a pipeline. For example:

```
ls a*  |  sed "s/.*/cp & x&/"  |  sh -
```

The output of *ls* is a list of all file names in the current directory whose names begin with "a", one per line. *Sed* (a "stream editor") converts each line of the form "name" into the form "cp name xname",[4] and passes it to a shell to be interpreted. A copy of each named file is generated under the name prefixed by "x".

Implied in the above discussion are several reasons why users like shell procedures better than compiled code. First, it is trivially easy to create and maintain a shell procedure, since 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 procedures "on the fly," use them, and remove them, without having to worry about managing libraries or about allocating disk storage. Finally, because such procedures are short in length, written at a high programming level, and kept in their source-language form, they are generally easy to find, understand, and modify.

4. *Cp* copies the file named by its first argument onto that named by the second, creating the latter, if necessary.

5. PATTERNS OF USAGE

5.1 Survey Methodology

A survey of PWB shell procedures was conducted, with the following goals:

- Discovery of procedures that could easily be redone in better ways, mainly to help users learn better ways of using existing tools, but also to improve system performance.
- Analysis of usage patterns, in order to improve existing PWB tools or build new ones. Rearrangements of command functions often occur when people recognize that the functions should be combined in different patterns. Examining user code is a good way to discover real needs.
- Determination of overall properties of procedures to help in redesign or enhancement to the shell and its supporting commands. This is part of an educational process aimed at putting existing tools to better use.

A program was written to find accessible shell procedures and print them for analysis by the author. Two facts made this a simple program. First, shell procedures are easy to manipulate because they are stored as simple text files. Second, the UNIX directory structure supports simple methods of traversal that permit easy investigation of *all* accessible files.

The files summarized in the next section represent a sample taken over a user population of more than 500 people. The data was collected in March, 1976.

5.2 Survey Results

A total of 1,725 procedures was analyzed. Table 1 summarizes the different forms of overall structure.

TABLE 1. Control Flow Summary

Category	Number	Percentage
1. single line	369	21
2. straight-line	935	54
3. argument loop only	83	5
4. branching, no loops	201	12
5. more complex	137	8

Procedures in category 1 consist of a single command line. Procedures in category 2 possess no control logic, but contain more than one line of text. The distribution of lines per procedure clearly favored small files, with a long "tail" of larger ones. This indicates that users find it helpful to "can" even one-line sequences. The large number of small procedures found on the PWB is at least partly due to the ease of creating them.

Thus, most procedures (75%) contain no control logic, but consist instead of straight-line scripts. The remaining 25% contain significant use of control logic. This percentage will probably increase with time, as more people become familiar with the shell's programming ability. In any case, it does indicate a significant need for control structures not provided conveniently in many CLS.

Procedures in category 3 are those whose control flow consists only of a single loop for processing the argument list, one at a time, such as:

```
: loop
if "$1" = "" exit
        ... commands
        shift
goto loop
```

Procedures in category 4 are those possessing conditional branching, but no loops. Those in category 5 thus have at least one loop and at least one conditional branch in addition to the one implementing the loop.

Each entry in Table 2 gives the number of shell procedures and the percentage of the total in which the specified construct was found. These figures were obtained by visual inspection of the files, and should probably be taken as lower bounds on the number of occurrences.

TABLE 2. Occurrence of Shell Constructs

Shell Construct	Number	Percentage
switch (or obvious use for it)	45	3
non-shell commands in file	514	30
pipe into shell (command | sh –)	32	2
parenthesized commands (i.e., implicit sh call)	39	2
explicit sh calls or obvious shell procedure calls	170	14
= a </dev/tty (read line from terminal)	81	5
command | = a (assign output of command to variable)	124	7
= a string (assign string to variable)	146	8
onintr label (intercept interrupt)	35	2
onintr – (ignore interrupt)	14	1
expr substr	33	2
expr + or –	24	1
next	13	1

The figures indicate the frequent intermingling (30%) of shell and non-shell commands in the same file, utilizing the fact that the shell and other programs share the same input. This intermingling occurs most often in the straight-line scripts. At least 14% of the procedures invoke another level of shell, in one or more of the three ways listed. When CL subroutines are available, people do make use of them, and tend to become extremely irritated with any CL that does not provide them. At least 5% of the procedures expect to write prompts to the user and read from the terminal; the figure would be much higher if it included prompts issued by programs other than the shell.

Despite the fact that *expr* supports multiplication and division, *no* serious uses of these operations were found.[5] From observation of the procedures, it is clear that users are doing much more string manipulation than arithmetic. Although *if* can perform arithmetic comparisons, the bulk of the operations performed by *if* were string comparisons. Likewise, the string operations of *expr* occurred more frequently than the arithmetic ones. It appears that CL arithmetic is a helpful facility, but string manipulation is a necessity, an ordering contrary to that of some existing CLS [COL76A].

5.3 Informal Observations

In many cases, a shell procedure is kept in a directory with the files that it manipulates, and is never used except in that directory. This is convenient in practice, and allows the user to forget about possible naming conflicts with procedures in other directories. It also supports the common practice of packaging related files in a separate directory.

5. Multiplication was used in two procedures, but they were ignored in the survey because they were obviously "play" procedures, i.e., two versions of a procedure to compute factorials, one iterative and the other recursive.

Multi-person projects often possess several sets of directories used to store shared procedures, and individuals may well have their own directories in addition. For example, project "tnds" may use "/u0/tnds/bin" as a repository for procedures needed by the entire project. Group "tf" within "tnds" may use "/u0/tnds/tf/bin" for procedures needed only by the group, and individual "jtb" might use another directory for personal commands. The ".path" file used in this case might contain:

```
:/u0/tnds/tf/jtb/bin:/u0/tnds/tf/bin:/u0/tnds/bin:/bin:/usr/bin
```

In general, people use this type of sharing mechanism to adapt the system to their own organizational needs.

5.4 Command Usage

Although the survey concentrated on static analysis, a few dynamic usage statistics may be of interest. Each of the larger PWB computers execute 25,000-40,000 commands per day. Approximately 80% of these commands are executed within shell procedures. These figures provide clear evidence that people utilize the shell programming capabilities to a very large degree.

6. AN ASSESSMENT

In the environment described in this paper, advances occur in three distinct ways: improving old tools, building new ones, and improving the framework used to put them together. Work is under way in all three areas, including most of the following:

- Addition of *while* to the shell.
- Creation of a "high-speed" shell that includes the bulk of the control structure code.
- Addition of different methods of manipulating pipes. For example, one would like to use programs that can have several input or output pipes, rather than only one of each.
- Cleanup of the overall syntax, which has its ugly aspects.
- Better ways of handling variables, including more powerful default and substitution mechanisms.
- Commands to perform argument processing without loops.

It is clearly recognized that the PWB CL does not do everything that its users would like it to do, so it is likely to continue evolving at a rapid rate. This paper has attempted to present a consistent view of the PWB CL as it has existed for a relatively long time (about nine months). It has emphasized the description of existing facilities whose usage patterns are well known, rather than the forecasting of things to come.

7. CONCLUSIONS

This paper has described a minicomputer-based CL that is more powerful and convenient to use than many of those available on much larger systems. It is a real CL used daily by people doing production work under tight schedules. Although much work remains to be done on it, it has definitely shown its worth as a programming language for many applications, and has made major contributions to the ability of users to get their work done with a minimum of effort and irritation.

ACKNOWLEDGEMENTS

The shell was originally written by K. Thompson; that it has been so adaptable to a wide variety of uses and machine configurations is a great tribute to its creator. Many of the additions were implemented by R. C. Haight and A. L. Glasser.

REFERENCES

All references cited in this paper appear at the end of *"An Introduction to the Programmer's Workbench,"* by Dolotta, T. A., and Mashey, J. R., in these Proceedings.