UNPL 1452

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **Using a Command Language as the Primary Programming Tool**

Date: **August 15, 1979**

TM: **79-3624-2**
79-9322-1

Other Keywords: **UNIX**
Shell

Author(s)	Location	Extension
T. A. Dolotta	**MH 2F-207**	**7797**
J. R. Mashey	**WH 1D-301B**	**6844**

Charging Case: **39380-980**
49110-954
Filing Case: **40125-001**

ABSTRACT

The attached paper has been accepted for presentation at the IFIP Working Conference on Command Languages, September 10-14, 1979, and will be published in the proceedings of that conference by the North-Holland Publishing Company.

Pages Text: 1	Other: 15	Total: 16
No. Figures: 0	No. Tables: 2	No. Refs.: 20

Bell Laboratories

subject: **Using a Command Language as the Primary Programming Tool**
Case: **39380-980,49110-954**
File: **40125-001**

date: **August 15, 1979**

from: **T. A. Dolotta**
MH 3624
2F-207 x7797

J. R. Mashey
WH 9322
1D-301B x6844

TM: **79-3624-2**
79-9322-1

MEMORANDUM FOR FILE

The attached paper has been accepted for presentation at the IFIP Working Conference on Command Languages, Berchtesgaden, West Germany, September 10-14, 1979, and will be published in the proceedings of that conference by the North-Holland Publishing Company.

T. A. Dolotta

J. R. Mashey

MH/WH-3624/9322-TAD/JRM-ae

Att.
As above

Using a Command Language as the Primary Programming Tool

T. A. Dolotta and J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

The command (or *shell*) language of the UNIX† system has evolved, over a period of several years, into a full programming language. It includes commands designed primarily for use within command files (*shell procedures*). Changes to the language have been aimed at making it more convenient to use as a high-level programming language and better suited for use by large programming groups. As a result, UNIX users have been able to eliminate a great deal of the programming drudgery that often accompanies software-development projects. Experience over the last five years has shown that a good command language can serve as the primary programming tool for thousands of users.

A brief discussion of the capabilities and evolution of the shell is followed by an analysis of patterns of shell usage by people who use the "Programmer's Workbench" version of the UNIX system in a variety of work environments. By analyzing what users really do when they have access to a command language that is both simple and powerful, we hope to provide insight into some open questions of command language design and implementation.

1. INTRODUCTION

A common occurrence in large programming projects is the diversion of significant resources from the building of the end product to the creation of support programs and procedures. A good operating-system command language (CL) is an invaluable tool for such projects. If a CL is a flexible programming language, it can be used to solve many such support problems *without* requiring "compilable" programs to be written, debugged, and maintained.

Existing CLs encompass a wide range of structures and abilities, ranging from those that are quite similar to assembly language to a few that have the appearance and abilities of high-level programming languages [2,4,11]. A diversity of viewpoints exists regarding the relative importance of various CL characteristics [5,16,19]. Several years of Programmer's Workbench UNIX (PWB/UNIX) experience [6] suggest that the following qualities are valuable in an on-line CL:

- *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.
- *Ease of on-line use*—CL syntax should emphasize simplicity and avoid the need for redundant typing. For example, blanks are often better delimiters than commas or parentheses.

† UNIX is a Trademark of Bell Laboratories.

- *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 consist of one-line invocations of single commands.
- *Organization of CL procedures*—It must be possible to share easily the use of procedures among users in a way that is both consistent with organizational structure and capable of adapting to changes in that structure; thus, as an example, it is usually *not* practical to require that each CL procedure have its command-search path embedded within itself.
- *Programming language features*—It must be convenient to write CL procedures with conditional branching, looping, handling of arguments and 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 will usually need additional capabilities, a somewhat different syntax, and a radically different set of priorities concerning the importance of various constructs. For example, the ability of a CL to connect executing programs in various ways appears to be crucially important, although this ability is quite limited in most CLs. Pattern-matching and other string manipulation operations are quite useful. Arithmetic operations are also helpful, but seem to be much less important than other capabilities.
- *Interchangeability of programs and CL procedures*—No user should need to know whether a given command is implemented as a compiled program or as a CL procedure. Syntactic differences of this sort should be avoided.
- *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. 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 in actual practice.

The UNIX command language, called the *shell*, possesses all of these qualities.

The reader is urged to remember, first, that the UNIX shell has existed, in one version or another, for about ten years and that there are nearly 2,000 installed UNIX systems in the world today (of which almost 200 are PWB/UNIX systems), collectively serving tens of thousands of users daily; second, that we have purposely omitted in this paper any discussion about how the UNIX shell may evolve in the future (e.g., planned extensions for networking, inter-process communication); all the features that we describe exist today in a production environment.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of this paper depends on familiarity with UNIX; [13] (or, even better, [14]) is a definite prerequisite, and it would be helpful to read one or both of [10,17]. For completeness, a short overview of the most relevant concepts is given below.

2.1 The UNIX File System

The UNIX file system's overall structure is that of a rooted tree composed of *directories* and other files. A (simple) *file name* is a sequence of characters other than a slash (/). A *path name* is a sequence of directory names followed by a file name, with each name separated from the previous one by a /. If a path name begins with a /, the search for the file begins at the root of the entire tree (this type of path name is often called an *absolute path name* because it is not relative to the user's *current directory*); otherwise, the search begins at the user's current directory. (The user may change current directories at any time by

using the `cd` command.) In general, a file name can be used wherever a path name can be used, and vice versa. Some sample names are:

`/`	the directory that is the root of the entire file structure;
`/bin`	directory of commonly used public commands;
`/u3/tf/jb/bin`	an absolute path name typical of programming projects involving several users. This one is a private directory containing commands and belonging to person `jb` in project `tf`;
`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 `/u3/tf/jb`, it names `/u3/tf/jb/bin/umail`.

The generic name `.` is the name of the current directory, while `..` is the name of the current directory's *parent* directory.

With each file, there is associated a set of read/write/execute permission flags (independent of one another, and different ones for the file's owner, for a separately-defined group of users, and for everyone else, respectively). As far as the file system is concerned, every file is simply a sequence of bytes.

Large projects require the ability to modify quickly and easily 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 file names to be used, even when the current directory is many levels deep in the directory structure. It also permits individual directories to remain fairly small, lessening the user's effort and the load on the computer.

2.2 Processes and Their Interactions

An *image* is a computer execution environment, including "core" image, register values, name of the current directory, status of open files, information recorded at login time, etc. 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 replace itself by issuing an `exec` system call, which initiates execution of another program.

Processes use independent address spaces and communicate in a limited number of ways:

- *Open files*—a child inherits the parent's open files. This permits processes to share the use of a common input or output stream in various useful ways. In particular, an open file possesses a *pointer* that indicates a location in the file and that may be modified by various operations: `read` (`write`) copies the 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 value of the pointer to an absolute location within the file, or to a location offset from the end of the file or from the location given by the current value of the pointer. The read/write pointers are shared between parent and child processes.
- *Arguments*—a sequence of arguments (character strings) can be passed from one program to another via `exec`.
- *Return codes*—when a process terminates, it can set a numeric return code that is available to the process's parent.

- *Files*—some programs have built-in conventions to share files in various ways, or to use files of specified names.
- *Pipes*—pipes are interprocess channels that are accessed like files, but that also allow very convenient (and efficient) handling of the "producer-consumer" relationship between concurrent programs. 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, nor does it need to be aware, most of the time, of the differences between pipes and files (of course, trying to `seek` on a pipe results in an error).

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 helpful in this regard.

We must emphasize here our belief that, even though a CL can, in certain cases, compensate for some deficiencies of the underlying operating system, it is very difficult to implement a *good* CL without such operating-system facilities as pipes, I/O redirection, convenient process creation, and a simple, homogeneous file system.

3. SHELL BASICS

Most UNIX users utilize the CL provided by the shell. It reads input from a terminal or a file and arranges for the execution of the commands named in that input. The shell can be a small program—some CL functions are actually implemented as independent programs that work with the shell, but are not necessarily built into it. The discussion below is adapted from [17,20].

3.1 Commands

A *command* is a sequence of non-blank arguments separated by white space (i.e., blanks and/or tabs). The first argument specifies the name of the command to be executed; the remaining ones are passed to it as arguments. 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* (as shown by its *mode*, i.e., a set of flag bits associated with the file) and is, in fact, a compiled program, the shell (as parent) spawns a new (child) process that immediately executes that program. If the file is marked as executable, but is not a compiled program, it is assumed to be a *command file* (or *shell procedure*), in which case the shell spawns a new instance of itself to read that file and execute the commands included in it; a command file is a file of ordinary text consisting of shell command lines and, possibly, lines meant to be read by other programs.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. Many operating systems can execute compiled programs without requiring the user to identify the source language used to produce such programs. In UNIX, CL procedures are treated in the same way, for several reasons. First, the existence of two distinct types of commands is confusing to users. Second, users become irritated when forced to repeatedly type in the same information, especially when the system can be legitimately expected to have that information already. Finally, the implementation of a given command may well change with time, typically from shell procedure to compiled program. Such a change would cause great discomfort to users if it were to force them, in turn, to change the way in which they must invoke that command.

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 to the name the string `/bin/`, and finally the string `/usr/bin/`. If the original command name is a simple file name, the effect is to search, in order, the current directory, the `/bin` directory, and the `/usr/bin` directory. A file may also be accessed by an absolute path name, or by a path name relative to the user's current directory

This mechanism gives the user the ability to conveniently 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. The search order permits a standard command to be preempted by a user's command without affecting anyone else.

3.3 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 that file. In either output case, the shell creates the file if it did not already exist.

3.4 Command Lines

A series of commands separated by the symbol `|` make up a *pipeline*. Each command is run as a separate process connected to its neighbor(s) by pipe(s), i.e., the output of each command (except the last) becomes the input for the next command in line. A *filter* is a command that reads its standard input, transforms it in some way, and then writes it on its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute concurrently, 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 the extreme example that requires *all* input to be read before *any* output is produced.

The following is an example of a typical pipeline:

```
nroff -cm text | col | lpr
```

The command `nroff` invokes a text formatter whose output may contain reverse line motions; `col` converts the output of `nroff` to a form that can be printed on a terminal lacking reverse motion, and `lpr` is a line-printer "spooling" daemon. The flag `-cm` indicates one of many possible formatting options, and `text` is the name of the file to be formatted.

An unadorned 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 -cm text1; nroff -cm text2) | col | lpr
```

If the last command in a pipeline is terminated by a semicolon (;) or a new-line, the shell waits for the command to finish before continuing its own 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.

Typical uses of & include off-line printing, background compilations, long-running administrative jobs (e.g., monthly accounting), and the staging of large jobs that are to be sent to other computers; a more realistic version of the first example in this section would therefore be:

```
nroff -cm text | col | lpr&
```

which would put the formatting and off-line printing into the background and free the terminal for other tasks. As a side comment, note that the above pipeline creates a minimum of three background processes; it is, of course, possible to run commands that, during their lifetime, spawn literally hundreds of processes.

Thus, I/O redirection and pipes complement one another: files and programs can *both* be used as data "sources" and "sinks." As a result, most commands neither know nor care whether their input (output) is coming from (going to) a terminal, a file, or a pipe. Commands are thus easily used in many different contexts. A few commands do 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 a 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 and the * matches any string of any characters other than / and new-line, including the null string.[1] Enclosing a set of characters within square brackets (`[...]`) causes the construct to match any one character in that set. Inside brackets, a pair of characters separated by a hyphen (-) includes in the set all characters lexically within the inclusive range of that pair.

For example, * matches all the names in the current directory that do not begin with a period, `*temp*` matches all names containing `temp`, and `[a-f]*` matches all names beginning with one of the letters a through f, while `/u3/tf/bin/?` matches all single-character names found in directory `/u3/tf/bin`.

This capability saves much typing and, more importantly, 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 (" or ') are, in general, taken literally, except that substitution of shell arguments and variables is normally performed inside double quotes, but not inside single quotes.

1. To be precise, the ? and * do *not* match a *leading* period (.).

3.7 Invoking the Shell

The shell may be invoked explicitly in various ways:

`sh file [ args ]` A new instance of the shell is created to begin reading `file`, where `file` can be a file name or a path name. Arguments can be manipulated as described in §4.2 below.

`file [ args ]` As noted in §3.1 above, if `file` is marked as executable and is not a compiled program, the effect is the same as with `sh file [ args ]`, except that `file` is searched for as described in §3.2 above.

The shell reads its input until end-of-file, so that a sequence of commands that can be executed interactively can also be placed in a file for subsequent execution as a shell procedure, as explained in §4.4 below. Because the shell is an ordinary program, separate invocations permit the use of shell "subroutines" without requiring that an explicit subroutine mechanism be built into the shell.

4. SHELL PROGRAMMING

4.1 Evolution of the Shell

Our discussion until now has treated the shell as a single program, whereas it actually represents an entire family of programs, most of which are still being used. The original UNIX shell (hereafter referred to as *osh*) offered, as of 1973, essentially all the features described above [17]. An elegant program of less than twenty pages of C code [9,15], it was easy enough to modify so that few could resist the temptation to do so. However, almost all resulting variants have retained its basic syntax and features for interactive use. The major area of difference, at least at Bell Laboratories, has been in the implementation of various programming features.

Osh provided ways to access up to nine positional arguments at any one time and to iterate over the entire argument list. Flow of control was provided by the *independent* commands `if`, `goto`, and `exit`. *Osh* remained in use at most installations over the next few years, and is still used on many machines, especially the smaller ones.

In mid-1975, PWB/UNIX developers added to *osh* a small number of string variables and a command for trapping terminal interrupts. A new "search path" mechanism allowed each user to specify a list of directories to be searched for commands. The capabilities of the `if` command were expanded and some additional commands were written specifically to support shell programming, whose level of usage, as a result, began to increase dramatically. By mid-1976, further changes had been made to provide more flow-of-control constructs and to improve performance by moving the (by then heavily-used) flow-of-control commands into the shell itself. The resulting forty-page C program (called the PWB/UNIX shell or *psh*) has been used with only minor modifications since then [12].

Finally, a complete redesign and reimplementation of the shell was carried out during 1976-78 [1]. Although somewhat longer (almost 100 pages of C code), the "New Shell" (or *nsh*) offers a number of new features (e.g., better looping constructs). Most UNIX users within the Bell System have converted to it, or are in the process of doing so.

Note again that a shell is an ordinary program, and that some machines offer several distinct shells (and/or old and new versions of the same shell). During periods of rapid change, a new shell is often made available to users for testing before it is installed officially, and an old shell is often kept for a while to permit users to test suspicious cases with both shells. Such a *modus operandi* would clearly be difficult, if not downright impossible, if the shell were implemented as part of the operating system itself.

During the past year, the rate of change of the current version of the UNIX shell (*nsh*) has decreased very markedly. It is our perception that, aside from perennial questions of efficiency, there remains one major area of user dissatisfaction with both *psh* and *nsh:* their overall syntax is not altogether as natural, clean, and simple as one might wish; on occasion, it can be unpleasantly awkward.

4.2 Programming Features of the Shell

A brief description of selected *psh* and *nsh* features is given below; details may be found in the references cited in §4.1 immediately above. Square brackets ([]) indicate that the enclosed items are optional.

4.2.1 Arguments. Like *osh*, *psh* recognizes the positional arguments `$1` through `$9` and uses the built-in `shift` command to iterate over the argument list. The notation `$*` is used to obtain the entire list of arguments at once. *Nsh* adds keyword arguments of the form *name=value.*

4.2.2 Variables. Psh provides 26 variables, `$a` through `$z`, a few of which are reserved for special uses. Such variables are local to each shell invocation. The following assigns *value* to the variable *$letter* (if *value* is omitted, one line is read from the standard input and assigned to the variable):

= *letter* [*value*]

Nsh allows variables that have arbitrary names and whose values are set by:

name=value

Nsh variable reference forms include *$name*, `${`*name*`}`,[2] and several forms that test whether or not the variable has a value. For instance, the form `${A=XYZ}` is replaced by the value of `A` if `A` is set; otherwise, it first causes `A` to be set to the value `XYZ` and is then replaced by that value. A user may create *environment* variables accessible to all programs; the values of these variables are inherited by all of that user's processes.

4.2.3 Flow of Control. Psh provides the following control constructs:

- `:` *label*
- `goto` *label*
- `onintr` [*label*]
- `if` *expression command*
- `if` *expression* `then`
  command-list
  `else`
  command-list
  `endif`
- `while` *expression*
  command-list
  `end`

2. The braces ({}) delimit *name*; they may be omitted if there is no ambiguity.

- `switch` *word*
  `:` *pattern1*
  command-list
  `:` *pattern2*
  command-list
  ⋮
  `: default`
  command-list
  `endsw`
- `next` *file-name*

The `:` is the null command; it is used for comments and to position a label. The `onintr` command traps a terminal interrupt and transfers control to the given *label*. The *expression* used by `if` and `while` may include string comparisons, arithmetic comparisons, and tests of file attributes. Inside a `while` loop, `break` terminates the loop and `continue` requests the next iteration. The `switch` command transfers control to the commands after the first *pattern* that matches the argument *word*, using the pattern-matching described in §3.5 above; `breaksw` is used to jump out of a `switch`-`endsw` group. The `next` command causes the shell to abandon the current input file and begin reading the one designated by *file-name*.

Nsh provides the following control constructs:

- `:`
- `#` *comment*
- `trap` *arguments*
- `if` *command-list* `then` *command-list* [`else` *command-list*] `fi`
- `for` *name* [`in` *word* ...] `do` *command-list* `done`
- `while` *command-list* `do` *command-list* `done`
- `until` *command-list* `do` *command-list* `done`
- `case` *word* `in`
  pattern [| *pattern* ...]) *command-list*`;;`
  ⋮
  `esac`
- `.` *file-name*
- `exec` *file-name* [*argument* ...]

The `:` is again the null command; the `#` signals the beginning of a comment. The `trap` command provides general control over the handling of interrupts. The `if` construct may also include `elif` parts. The `goto` command is absent, although `break` and `continue` are provided, as in *psh*. The *expression* evaluation of *nsh* is performed by a built-in command whose result can be tested. The `for` loop iterates over a list, which defaults to the entire argument list (`$*`). The `.` command makes the shell suspend execution of the current file in order to execute the new file, then return and continue the execution of the old file. The `exec` command causes the shell to abandon its own execution in favor of *file-name*. The *nsh* flow-of-control constructs work interactively at the terminal; `for`, in particular, is often used in this manner.

4.2.4 Capturing Command Output. *Psh* allows one line of output from any UNIX command to be assigned to a variable:

command ¦ = *letter*

With *nsh*, the *entire* output of a UNIX command may be substituted by enclosing its invocation in grave accents:

`` `command argument ...` ``

so that:

```
D=`date`
```

assigns to the shell variable $D the output of the UNIX `date` command, e.g.:

```
Fri Jun 29 19:34:45 EDT 1979
```

4.3 Examples

The following is a simple *psh* procedure (called `makefiles`) that creates a list of files with numbered names, but avoids overwriting already existing files (the `if` command succeeds if the file named by `$1$i` exists and is readable). The *expr* command performs arithmetic and various string operations; in this example, it is used to run down a counter:

```
:       Usage: makefiles name n
:       Creates n files called name1, name2, ...
= i $2
while $i -gt 0
        if -r $1$i then
                echo "$1$i already exists"
        else
                echo "This is file $1$i." >$1$i
        endif
        expr $i - 1 ¦ = i
end
```

The following is a *nsh* version of the "Towers of Hanoi" program found in [7]:

```
if test $1 -eq 0; then exit; fi
hanoi `expr $1 - 1` $2 $4 $3
echo "move disc $1 from $2 to $3"
hanoi `expr $1 - 1` $4 $3 $2
```

It is invoked via:

```
hanoi N S D I
```

where `N` is the number of discs one wishes to use and `S`, `D`, and `I` are the names of the starting, destination, and intermediate needles, respectively. Note the use of grave accents to capture the output of the `expr` command and pass it as a (single) argument to `hanoi`.

4.4 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. The second step is that of changing the mode of that file to make it executable, thus permitting it to be invoked by `name` *args*, rather than by `sh name` *args*, as discussed in §3.1 above. 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 create a procedure that formats text files into a standard format and prints them on a particular type of terminal:

```
ed
a
nroff -rT1 -T300-12 -cm $*
.
w draft
q
chmod +x draft
```

The user called the text editor ed, entered a single line of text (`nroff ...`), wrote out that line (creating the new file `draft`), and finally changed the mode of `draft` to make it executable. The user may then invoke this command as `draft file1 file2`, for example. The procedure calls the formatter, passing to it all user-supplied arguments (`$*`) in addition to the ones built into the procedure (`-rT1`, etc.). An even shorter way to create such a one-line shell procedure is:

```
echo 'nroff -rT1 -T300-12 -cm $*' >draft; chmod +x draft
```

If either of the above sequences is executed in a directory whose name is included in the user's command-search sequence, then the user can invoke the `draft` command from any directory. Other users can also invoke it, especially if it is placed in a shared directory.

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, because it is just a file of ordinary text; creating shell procedures is not very different from executing shell commands, one at a time, from a terminal. Second, there is no corresponding object program that must be generated and maintained. 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 understand, to modify, and to debug (the shell has good facilities for tracing the execution of command procedures). The next section provides pragmatic evidence of the success of this approach.

5. PATTERNS OF USAGE

5.1 Survey Methodology

During March 1979, a survey of PWB/UNIX shell procedures was conducted. The results are compared with three-year old figures from an earlier paper [12]. A program was written to find accessible shell procedures and print them for analysis. In addition, usage figures were obtained from various system files and accounting procedures.

5.2 System Characteristics

The system surveyed is a large PWB/UNIX machine at Bell Laboratories in Piscataway, NJ. Many of its users develop programs (written in PL/I and other languages) to be run on IBM System/370 computers; some users develop C programs to be run on UNIX systems. Almost all users make significant use of the excellent UNIX text-processing facilities and many use UNIX for project-administration work. The system recognizes 240 distinct login names. A few users have several login names and some login names are used for system administration; thus, the number of real users served by that system is just over 200. A sample of a day's activity on that machine is shown in Table I; "prime time" is the eight-hour interval from 9 A.M. to 5 P.M. Note that the numbers of processes and shell procedures executed per connect hour are averages; one must remember that many connect hours are spent in editing files, during which times there are only two processes per user, namely the shell and the editor.

TABLE I. A Day's Activity (3/79)

Number of distinct users	103
Maximum observed number of simultaneous users	40
Mean number of simultaneous users (prime time)	28
Number of user sessions	346
Mean connect hours per user session	0.8
Total CPU hours used (prime time)	4.7
Processes run (total)	29,398
Processes run (prime time)	22,775
Processes run per connect hour (prime time)	102
Shell procedures run (prime time)	4,933
Shell procedures run per connect hour (prime time)	22

5.3 Survey Results

Our survey program found a total of 1,637 shell procedures written by users. Table II summarizes the different forms of their structure and compares the results of the 1979 survey with those of the 1976 survey. The 1976 survey covered a number of machines that, together, served more than twice the number of users surveyed in 1979. Thus, although the *total* numbers of shell procedures are similar for the two surveys, the *average number of shell procedures per user* has more than doubled between the surveys (from about 3.5 to about 8).

TABLE II. Summary of Shell Procedure Structure

Shell Procedure Category	1976 Survey (≈500 users)		1979 Survey (≈200 users)	
	Number	*Percentage*	*Number*	*Percentage*
Single-line	369	21	269	16
Straight-line	935	54	653	40
More complex	421	25	715	44
Totals	1,725	100	1,637	100

Procedures in the first category of Table II consist of a single command line. Procedures in the second category possess no control logic, but contain more than one line of text. The last category includes all procedures that contain conditional logic or loops. The distribution of lines per procedure clearly favors small procedures, with a long "tail" of larger ones. The mean number of lines per file is sixteen; the mode is one. This indicates that users find it helpful to "can" even one-line sequences; the large number of tiny procedures is at least partly due to the ease of creating them.

The figures of Table II indicate a significant use of control structures that are not provided conveniently in many other CLs. These figures also indicate that the use of shell programming has increased with time.

Our survey program also recorded the number of compiled C programs that it found; in 1979, there were 313 compiled programs, yielding a ratio of five shell procedures for every compiled program.[3] These figures actually understate the use of shell procedures, because

3. These numbers include 206 compiled programs and 266 shell procedures belonging to one particular user group; subtracting these numbers from the above totals yields a shell-procedure-to-compiled-program ratio of 12:1, more typical of most large PWB/UNIX systems.

our survey program counts every compiled program it can find, but must use heuristic methods to identify shell procedures; thus, our shell procedure counts should be taken as lower bounds.

5.4 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 makes viable the common practice of "packaging" related files in separate directories, thus using the UNIX directory structure to express the relationships among the various collections of files.

Projects that involve several users usually possess a number of directories for storing shared shell procedures; in addition, individual users often have private procedure directories. For example, project `tf` might use `/u3/tf/bin` as a repository for procedures needed by the entire project; group `dv` within `tf` might use `/u3/tf/dv/bin` for procedures needed only by the `dv` group, and individual `jb` might use `/u3/tf/dv/jb/bin` for personal shell procedures.

More generally, we have observed that PWB/UNIX user groups almost invariably create for themselves a set of nested computing environments that are specialized to their individual nomenclature, organization, etc., and which they can change dynamically:

- Current directory.
- Login environment.
- Group environment.
- Project environment.
- System environment.

We believe that this phenomenon is so widespread because such environments are very easy and inexpensive to build on UNIX, and because they enable users to stand on other users' shoulders, rather than toes. It has been also suggested that programming groups have functional (and perhaps social) needs to customize their respective work environments.

We observe, in passing, that such encapsulated environments are strongly reminiscent of the concept of "Abstract Machines" [18].

The PWB/UNIX shell has been able to do arithmetic for the last four years. However, study of thousands of PWB/UNIX shell procedures has shown that these procedures are used much more for string manipulation, pattern matching, and for testing file attributes than they are for arithmetic; the emphasis that arithmetic facilities have often received in other CLs [3] is at odds with this observation.

The complex argument specifications and default facilities found in some CLs are not provided by the shell. In practice, the lack of such facilities is overcome by the ease of creation of shell procedures. In particular, many one-line procedures are used to customize a general-purpose command for a specific use, often supplying parameters for which no reasonable a priori defaults are possible.

As for efficiency, some shell procedures are almost as efficient as equivalent compiled programs. Many, however, are as much as three orders of magnitude *less* efficient. The usage figures we have observed clearly indicate that, to many users, this is an acceptable trade-off, because the gain in "people efficiency" that results from the ease of creation, modification, and debugging of shell procedures much more than recoups the loss of "machine efficiency." Given a shell procedure and a compiled program that perform the same

function, the ratio of their respective sizes (measured in, say, lines of code) often exceeds two orders of magnitude.

Each year, more people become convinced that software should be built using tool-kits of established, reliable modules. The worth of this idea is clearly proven by years of UNIX experience, which has shown that an unusual, labor-saving methodology [8,10] spreads rapidly when a good set of tools can be easily interconnected by a CL that is both powerful and easy to use.

6. CONCLUSIONS

This paper has described a minicomputer-based CL that is more powerful and convenient to use than many CLs available on much larger systems. It is a *real* CL used daily by thousands of people whose computer expertise ranges from essentially none to very extensive and many of whom are doing production work under tight schedules. It has proven its worth as a programming language in a broad variety of situations and has increased significantly its users' ability to get their work done with much less effort and irritation than was previously possible.

ACKNOWLEDGEMENTS

The original shell (*osh*) was written by K. Thompson; that it has been so adaptable to so wide a variety of uses is a tribute to its creator. Most of the PWB/UNIX additions (*psh*) were implemented by A. L. Glasser, R. C. Haight, and the second author. The latest version of the UNIX shell (*nsh*) is due to S. R. Bourne.

REFERENCES

[1] Bourne, S. R., UNIX Time-Sharing System: The UNIX Shell, *Bell System Technical Journal 57*, 6 (July-Aug. 1978, Part 2) 1971-90.

[2] Brunt, R. F., and Tuffs, D. E., A User-Oriented Approach to Control Languages, *Software—Practice and Experience 6*, 1 (Jan. 1976) 93-108.

[3] Colijn, A. W., Experiments with the KRONOS Control Language, *Software—Practice and Experience 6*, 1 (Jan. 1976) 133-36.

[4] Cowan, R. M., Burroughs B6700/B7700 Work Flow Language, in: Unger, C. (ed.), *Command Languages—Proc. IFIP Working Conference on Command Languages* (North Holland, Amsterdam, 1975) 153-66.

[5] Dolotta, T. A., and Irvine, C. A., Proposal for a Time Sharing Command Structure, in: *Information Processing 68—Proc. IFIP Congress 1968*, Vol. 1 (North Holland, Amsterdam, 1969) 493-98.

[6] ———, Haight, R. C., and Mashey, J. R., UNIX Time-Sharing System: The Programmer's Workbench, *Bell System Technical Journal 57*, 6 (July-Aug. 1978, Part 2) 2177-200.

[7] Griswold, R. E., Poage, J. F., and Polonsky, I. P., *The SNOBOL4 Programming Language*, Second Edition (Prentice-Hall, Englewood Cliffs, NJ, 1971).

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

[9] ———, and Ritchie, D. M., *The C Programming Language* (Prentice Hall, Englewood Cliffs, NJ, 1978).

[10] ———, and Mashey, J. R., The UNIX Programming Environment, *Software—Practice and Experience 9*, 1 (Jan. 1979) 1-15.

[11] Madsen, J., CCL—A High-Level Command Language, *Software—Practice and Experience 9*, 1 (Jan. 1979) 25-30.

[12] Mashey, J. R., Using a Command Language as a High-Level Programming Language, *Proc. Second International Conference on Software Engineering* (Oct. 13-15, 1976) 169-76.

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

[14] ———, The UNIX Time-Sharing System, *Bell System Technical Journal 57*, 6 (July-Aug. 1978, Part 2) 1905-29.

[15] Ritchie, D. M., Johnson, S. C., Lesk, M. E., and Kernighan, B. W., UNIX Time-Sharing System: The C Programming Language, *Bell System Technical Journal 57*, 6 (July-Aug. 1978, Part 2) 1991-2019.

[16] Simpson, D. (ed.), *Job Control Languages—Past, Present, and Future* (National Computing Centre Ltd., Quay House, Quay St., Manchester, England, 1974).

[17] Thompson, K., The UNIX Command Language, in: *Structured Programming—International Computer State of the Art Report* (Infotech Information Ltd., Maidenhead, Berkshire, England, 1975) 375-84.

[18] Unger, C., The KIWINET/NICOLA Approach: Concepts of the KIWINET/NICOLA Proposal, in this proceedings.

[19] Unger, C. (ed.), *Command Languages—Proc. IFIP Working Conference on Command Languages* (North-Holland, Amsterdam, 1975).

[20] *UNIX Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Vol. 1 (Bell Laboratories, Murray Hill, NJ, Jan. 1979).