ON THE STRUCTURE AND CONTROL OF COMMANDS

(One man's data are another man's command)

C.J. Stephenson,*
Stanford University

ABSTRACT

An interactive command language, with its underlying data, defines a command environment. In general a command environment supports a number of commands which once issued perform non-interactively, and which when finished leave the old command environment in control. It also supports some special commands which move to other command environments, after which commands are interpreted according to a different set of rules.

The usefulness of a command environment can be extended by programming it, i.e. by dynamically constructing and conditionally executing sequences of its commands; but, unlike a programming language, a command language does not usually contain any general-purpose variables or means for conditional execution. These facilities can however be provided by a command control language, which makes it possible to construct sequences or commands to be issued to the currently active command environment from a program.

A command control language is described, and the usefulness, limitations and repercussions of command language programming are discussed.

1. INTRODUCTION

Advances in computer languages during the past 15 years have not generally been reflected in command and control languages. Those in common use are for the most part primitive and awkward. Perhaps this situation exists because the command and control language interpreters are usually an integral part of shared computer systems; so that stability has been required, development has been difficult, and people who have had an interest in changing or extending them have seldom been able to do so. The advent of single-user systems running in virtual machines has, however, presented the opportunity freely to modify all parts of the system, and thereby to experiment in this area.

This paper presents a modular command language structure, and a control language for commands. The control language resembles the EXEC language of the IBM Conversational Monitor System (CMS), but is not equivalent to it. Finally we consider the repercussions of command language programming on the design of the system.

* Present address: IBM, Thomas J. Watson Research Center, Yorktown Heights, N.Y.

2. COMMAND LANGUAGE STRUCTURE

In this section we describe a modular command language structure, together with a hypothetical operating system environment in so far as it is visible at the command language level. The description that follows sets the stage for the rest of the paper, and also serves to draw attention to certain aspects of interactive systems which seem often to be left not clearly explained.

In what follows, we assume the existence of a file system which is external and static. By this we mean that it is accessible from all processes, and the data in it have a life which is terminated only by explicit erasure.

2.1 Interaction

We consider an operating system which is designed to be interactive. Input from the user is obtained from a typewriter-like keyboard; output may be printed on a typewriter-like device, or presented on a video display. The user's primary input-output device (whatever its exact form) is called the console.

2.2 Commands, Command Languages, and Command Environments

At any stage during the course of a user's transactions, his input is examined and acted upon by the program receiving it. This program is said to support the active command environment. The user may move around between command environments by giving special commands which set up a new command environment, or revert to a previous one.

For example, the initial exchanges with a system are usually concerned with 'logging in'. To start with the user talks to the LOGIN routine, which may support only a single command (say LOGIN). When that command has been correctly given, the user enters a new command environment (say SYSTEM) which supports a series of commands (but not LOGIN).

A command environment is defined by a command language, together with underlying data. The command language is executed and defined by a command language interpreter, of which there is at least one associated with every interactive program. The nature of the underlying data depends upon the semantics of the command language. For example, the LOGIN command language may define the command 'LOGIN user-name', and the underlying data (the list of users) define the possible values of 'user-name'. These two together define the command environment. For another example, consider an interactive editor, in which the command language defines the operations that can be performed, and the underlying data comprise the text of the file being edited.

Note that, according to these definitions, it is only interactive programs which support command environments. This will be a convenient restriction for our purposes (but see Section 5.4). An interactive program is defined to be one which obtains its own input from the console. Thus a program which is invoked as a result of issuing a command from the console is not necessarily itself an interactive program: it is so only if it obtains additional console input, beyond that given to it in the command.

Commands are given in the form of a logical line, which will often correspond to a physical line (i.e. an entry terminated by carriage return). Depending on the command environment, a line may go under the name command, request, statement, etc. Here we shall for the most part use the terms command and command line, which have the desired connotation of immediacy. As a general rule, the first word of a command is the name of the program or routine to be executed (or an abbreviation or synonym for it); and the remainder of the command line comprises its parameter list. (According to these conventions, a more appropriate name for the LOGIN command mentioned above would be SYSTEM, since this is the command environment which is entered as a result of issuing the command.) Here, and in what follows, a word is any string of contiguous non-blank characters. Words are separated from each other by at least one blank.

A command language interpreter, in its simplest form, merely reads command lines, delimits the first word, identifies the corresponding program or routine, and passes control to it, along with access to the remainder of the command line. A command, once its name has been identified, is executed unconditionally. By this is meant that control is unconditionally passed to the program or routine that implements the command. It is of course permissible for the program or routine to decide, on the basis of its parameter list, or from other data, that it should in fact do nothing, i.e. that it should simply return control to the command environment from which it was invoked.

A parameter list may contain literal data, may refer to objects or collections of data by name, or may elaborate on the function to be performed. An example of literal data are the words 'HELLO THERE' in the command:

```
MESSAGE OPERATOR HELLO THERE .
```

An example of an 'object' referred to by name is the word 'OPERATOR'; and an example of a collection of data referred to by name is the word 'PROG' in the compile command:

```
ALGOL PROG .
```

Finally, the word 'ALGOL' is an example of function elaboration in the following command for background compilation:

```
BATCH ALGOL PROG .
```

Except in the case of literal data, a parameter list is treated strictly as a sequence of words, i.e. multiple blanks are equivalent to a single blank.

Suppose (following the example above) that the primary command environment is called SYSTEM. Its command language is the SYSTEM command language, and its underlying data include the user's file directories. From this command environment, programs which reside on file can be invoked by typing their name as the first word of a command. Most of these programs probably do not involve any further interaction, i.e. they perform some action on the data (such as erasing a file from the directory) and return control to the SYSTEM command environment. In this case SYSTEM remains the active command environment. Some however may themselves be interactive, and support their own command environments: examples are editors, debugging programs and query systems. The invocation of one of these will cause the SYSTEM command environment temporarily to become 'dormant', and activate a new command environment defined by the program now in control. A 'dormant' command environment has the potential for being 'reawakened' later, when one of the subsequent environments is terminated.

At any time, the user has the ability to communicate directly only with the active command environment; however, this command environment will usually support at least one special command which terminates it; and will often support some other special commands which explicitly activate

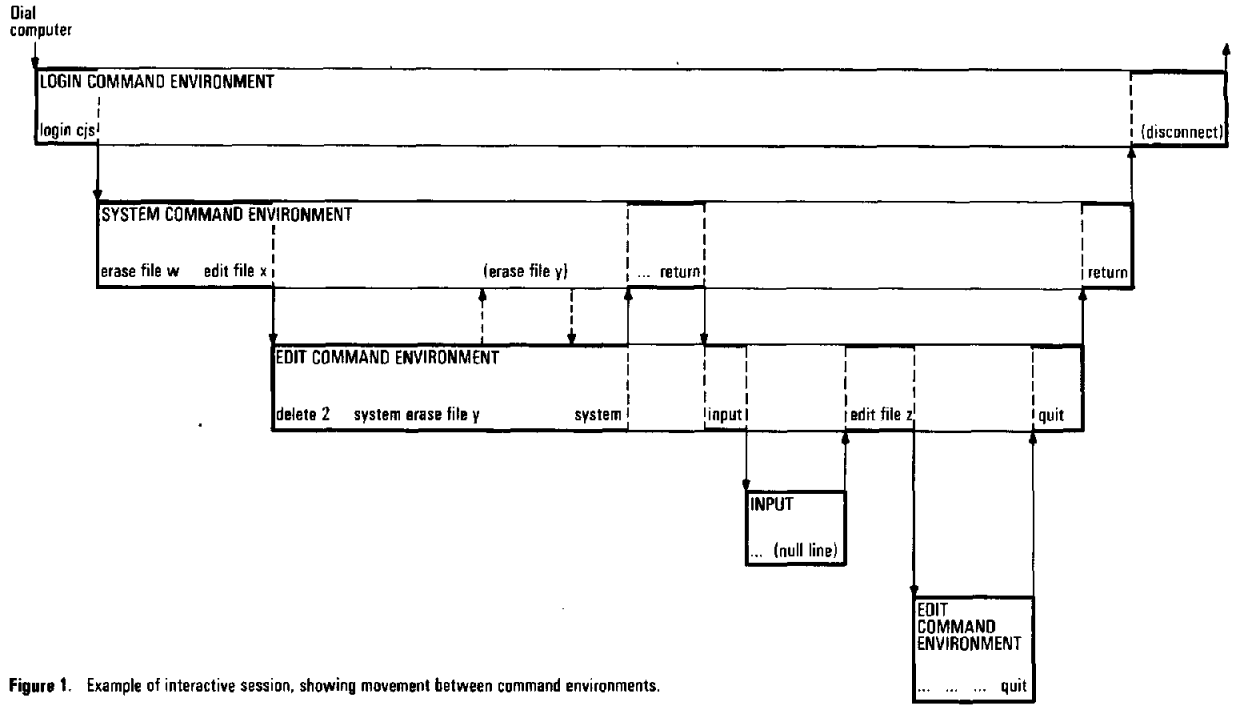

Figure 1. Example of interactive session, showing movement between command environments.

other command environments, or which pass a single command to another command environment. The rules for moving between command environments define the possible hierarchies of command environments.

2.3 Examples

Consider the following scenario, illustrated in Figure 1.

(a) The user is talking to SYSTEM, and types 'ERASE FILE W'. This command invokes a non-interactive program called 'ERASE' which erases FILE W and returns control to the SYSTEM command environment.

(b) The user is talking to SYSTEM, and types 'EDIT FILE X'. This command invokes an interactive program which sets up its own command environment, called 'EDIT'. This becomes the active command environment until the user gives one of the special edit commands which is concerned with a further change of command environment. SYSTEM, meanwhile, becomes dormant.

(c) The user is talking to EDIT, and types 'DELETE 2'. This is in the same class as (a) above. The command invokes a non-interactive edit routine called 'DELETE' which deletes two lines from the file and returns control to the EDIT command environment.

(d) The user is talking to EDIT, and types 'SYSTEM ERASE FILE Y'. There is (let us suppose) a special edit command 'SYSTEM' which, when issued with a parameter list, passes its parameter list as a command to the dormant SYSTEM command environment. Thus the command 'ERASE FILE Y' is issued via SYSTEM, which then returns control to EDIT. EDIT remains the active command environment, since there is no interaction with the SYSTEM command environment.

(e) The user is talking to EDIT, and types 'SYSTEM' alone. This (we shall suppose) reactivates the dormant SYSTEM command environment, making EDIT dormant. Further commands are received and acted upon by SYSTEM (or its descendants). The command 'ERASE FILE Y' (for example) could now be issued without the prefix 'SYSTEM'. More usefully, system commands could be given to modify the file directory access, so that the editor could then get at additional files. Finally, the user types 'RETURN': this is a special system command which makes SYSTEM dormant again and reactivates the command environment from which it was invoked (in this case EDIT).

(f) The user is talking to EDIT, and types 'INPUT'. This causes the editor to enter a new mode in which it receives an indefinite number of lines for insertion into the file, until a null line is entered which causes return to EDIT. INPUT mode can be thought of as a degenerate command environment in which all non-null lines are deemed to be preceded by the word 'INSERT', and in which a null line means 'return'. The INPUT command is (then) in the same class as (b) above, i.e. it activates a new command environment and causes the previous one to become dormant.

(g) The user is talking to EDIT, and types 'EDIT FILE Z'. This is a special edit command which invokes the editor recursively, activating a new

instance of the EDIT command environment, with (in this example) a different file to be edited. Subsequent commands will be received and acted upon by the new instance of the EDIT command environment (or its descendants) until the previous instance is reactivated.

(h) The user is talking to EDIT, and types 'QUIT'. This is a special edit command which terminates the active instance of the EDIT command environment and reactivates the command environment from which it was set up (e.g. EDIT or SYSTEM).

Note that the non-interactive programs ERASE and DELETE could have been written to be interactive: typing 'ERASE' alone could for example enter a command environment which reads a list of file names to be erased. Similarly, the interactive programs EDIT and INPUT could have been written to be non-interactive: typing 'EDIT FILE X DELETE 2 QUIT' could for example accomplish a small editing operation without involving any further interaction. Whether a particular program should set up a command environment is ultimately a matter of convenience and taste.

2.4 Additional Notes on Command Environments

1. By convention, the identity of the active command environment can in most cases be ascertained by entering a null line, to which the command environment responds by displaying its name.

2. In command environments generally, command lines are treated literally, in the sense that there is no replacement of any part of the command line by substitution or expansion.

3. In the cases illustrated above, the command environments behave analogously to subroutines, i.e. they can be activated ('be called'), in some cases they can activate others ('call'), and they can be terminated ('return'). They do not, however, necessarily behave in this way, as can be seen from the following example:

```
P0: BEGIN;
        CALL P1;     /* ACTIVATE E1 */
        CALL P2;     /* ACTIVATE E2 */
    END.
```

The program P0 consists simply of an invocation of an interactive program P1 which supports a command environment E1, followed by an invocation of a second interactive program P2 which supports a command environment E2. If P0 is invoked, from (say) command environment E, then E1 is activated, E becoming dormant. When E1 is terminated, however, activation is not returned to E, but is passed to E2.

4. In the above discussion, command environments have been classified as active, dormant or (by implication) not activated. It turns out that the difference between dormant and not activated is difficult to define with precision except in terms of the implementation: in some situations it may be impossible to distinguish functionally between reactivation of a dormant command environment and activation of a new one. This distinction is not however crucial: the essential difference is between active and not active.

5. Syntactically, an interactive 'session' can be expressed as follows.

```
<session>       ::=  <conversation>
                   | <session>  <conversation>

<conversation>  ::=  <talk>  <special command>

<talk>          ::=  <null>
                   | <talk>  <command>
```

A 'conversation' is a sequence of commands issued to a single command environment. A 'special command' causes a transfer to a different command environment, and thereby suspends or terminates one conversation and starts another.

6. The syntax rules for commands, which have been described above in terms of words and lines, together with the absence of substitution or expansion in the command line, have been given in order to establish a definite picture of what is going on. The main ideas of this paper could be adapted to situations in which these rules did not apply.

7. We may note, in passing, that the modular command structure described here, giving clear separation of command environments, has several advantages over alternatives in which the separation is less rigorous.

(a) An indefinite number of interactive programs can be added to the system without restricting the commands they may use. There is no danger of their command names clashing and interfering with others in the system, since the commands of a program are recognized only when that program is the one which receives them.

(b) Good diagnostic messages can be given in the event of an invalid command being issued, since there is no ambiguity over the command environment to which the command was issued.

(c) A new user, armed with the description of an interactive program (such as an editor) can be confident that he has a complete description of his environment, provided only that he does not explicitly move outside it.

2.5 Input Buffer

There is in the system an input buffer in which an arbitrary number of logical lines, or command lines, can be deposited. It is under program control. All console input is obtained by way of a system routine which reads an actual line from the console only if the buffer is empty; otherwise it returns (and removes) a line from the buffer. We shall consider the case in which the buffer acts as a push-down stack, i.e. the last line entered is the first to be removed.

The input buffer can play a useful part in communications between command environments, for

it can be used for the deposit of lines or messages from one command environment, to be read at a later time when another command environment is in control.

By these means it is possible to initialize one command environment by stacking lines from another, or to modify the rules for transferring between command environments. Consider, as a trivial example, the program P0 given above. Suppose that the program P2 stacked a line reading 'P0' during its termination, just prior to its returning control to P0. This line would read by E, in lieu of the next command from the console, and (we may assume) would cause the reinvocation of P0, and hence the reactivation of E1, and then E2, and so on.

2.6 Return Codes

Every command, in every command environment, finishes with a return code. This is for simplicity an integer, is passed back to the command environment from which the command was issued, and conveys success (if appropriate) or the type of failure. The command environment may choose whether or not to display the return code. The following are hypothetical return codes for two system commands which have already been used as illustrations.

ERASE	0 - file successfully erased
	1 - syntax error in parameter list
	2 - file does not exist
	3 - file cannot be erased
	4 - I/O error when erasing file
EDIT	0 - end of normal edit conversation
	1 - syntax error in parameter list
	21-40 - input error while reading file
	41-60 - output error while writing file

3. EXECUTION CONTROL

The commands of a command language, being immediate and unconditional, are analogous to the instruction set of a computer, less those instructions which make use of the condition code (or which affect the flow of control in any way). Typing the commands one by one is analogous to single-cycle operation of the machine, each instruction being stored and then executed. Commands which move between command environments are analogous to loading new microcode.

Note that a command language is not a programming language; for it lacks the notion of evaluation (except for the return code); in general it lacks variables; and it lacks the ability to execute sequences of actions conditionally on the result of some particular evaluation.

A command environment can be extended by introducing execution control at the level of the command language, i.e. by introducing the ability to issue commands from a program which can receive and build parameter lists, issue commands to the active command environment, examine return codes, and control its own execution. This enables several or many commands to be combined into a single 'macro' command which can be executed by typing the name of the program as the first word of a command. This corresponds to full programming of a machine, with address modification, and setting and testing of a condition code; but instead of machine programming we have command programming.

By these means, it is possible to add programming to any command environment, and thereby to write 'macro' system commands, 'macro' edit commands, 'macro' queries, and so on. This is the kind of programming which is likely to be useful to people using editors, query and reservation systems, and other interactive programs, who are not primarily programmers. Having learned the syntax and function of the command languages in which they converse, they could in many cases benefit from being able to combine commands, either in trivial sequences, to save repeated typing, or in more complicated programs, with parameters, to provide essentially new functions.

The following terminology will be used. A command which is implemented without recourse to a program of commands will be called a primitive or primitive command of the command environment. All the examples in Section 2.3 above are assumed to be primitive commands. A program of commands will be called a command language program.

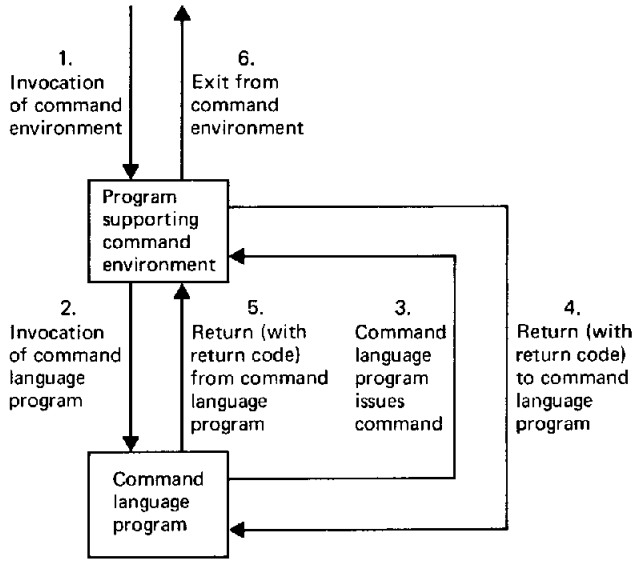

Figure 2. Relationship between program supporting command environment and a command language program for the same environment.

Figure 2 shows the relationship between the program supporting a command environment (including the primitives of the environment) and a command language program for the same command environment. The numbers show the order in which the paths are taken for a single invocation of the command language program, which is assumed to issue only a single command to the environment.

CONTINUED TO PAGE 127

CONTINUED FROM PAGE 26

These questions now arise.

(A) What language or languages should be used for writing command language programs?

(B) What should be the rules and style for invoking a command language program?

(C) To what extent is it feasible or desirable to provide common facilities for writing and executing command langauge programs for different command environments?

(D) What are the repercussions of command language programming on the design of command environments?

3.1 Language Considerations

Following Wilkes (1968), let us regard a programming language as composed of an 'inner' language, which defines actions, and an 'outer' language, which provides a control structure. An example par excellence of such a language is APLGOL, by Kelley (1973), in which for instance the IF statement can be written:

```
IF  <APL-expression>  THEN
   BEGIN
      <APL-statement>
      ...
   END
ELSE
   ...
```

In this case the inner language is APL, and the outer language is a subset of ALGOL.

Now we can regard a command language, in the sense developed in Section 2 of this paper, as an inner language, defining actions, and embed its commands within an outer control language. This will give us one of the essentials for writing programs of commands. It is not however sufficient, since our inner language (the command language) does not contain any variables, expressions or values (except for the return codes). These are needed for two purposes: (a) to yield truth values to the outer language, such as in the IF clause; and (b) to construct or modify command lines. This leads us to propose an additional inner language, to manipulate variables and evaluate expressions, which 'coresides' with the command language and is controlled by the same outer language. Henceforth we shall refer to these three language components as control language, manipulation language and command language. The first two, taken together, comprise a command programming language or command control language. Note that the control and manipulation languages often need not be cognizant of the detailed syntax or function of a command line: to the control language a command is an unspecified action; to the manipulation language it is data.

3.2 Design Choices

Within this framework, there are still several overall design choices to be made.

(1) We could choose to use an existing general-purpose programming language for the functions of control and manipulation, and provide an 'escape' from the language for the issue of

```
LISTSORT: PROC (PARM,CODE);
DECLARE PARM CHAR (*) VARYING, CODE BINARY FIXED;
DECLARE ESCAPE ENTRY EXTERNAL (CHAR(*)VARYING,BINARY FIXED);
DECLARE RETCODE BINARY FIXED;
    CALL ESCAPE ('LIST ' || PARM || ' (FILE)',RETCODE);
    IF RETCODE ¬= 0 THEN  /* IF THESE FILES DO NOT EXIST, THEN... */
        DO;               /* ...EXIT PRONTO WITH RETURN CODE OF 1. */
            CODE = 1;
            RETURN;
        END;
    CALL ESCAPE ('SORT FILE LIST',RETCODE);
    CALL ESCAPE ('PRINT FILE LIST',RETCODE);
    CODE = 0;
END;
```

(a) PL/I version

```
LIST &1 &2 (FILE)
* IF THESE FILES DO NOT EXIST, EXIT PRONTO WITH RETURN CODE OF 1...
&IF &RETCODE ¬= 0 &EXIT 1
SORT FILE LIST
PRINT FILE LIST
```

(b) CMS EXEC version.

Figure 3. Simple command language program, written in two languages.

commands. (A simple escape mechanism is a reserved procedure name which accepts a command line as a character-string parameter.) Alternatively we could design a special-purpose language (or pair of languages), in which the data-types and functions which are heavily used for command language programming are made more accessible, at the expense of those which are rarely used. Figure 3 shows a small command language program written in two languages, (a) in PL/I, representing the general-purpose languages, and (b) in CMS EXEC, a special-purpose command programming language of the IBM Conversation Monitor System. The program issues three hypothetical primitive system commands, to list a subset of the user's file names with the 'file' option, to sort the list, and finally to print it. The subset of the file names to be listed is specified in the parameter list of the command language program. The difference in lengths between the two versions of the program seems to be sufficient to warrant giving serious consideration to the use of a special-purpose language. The difference in length would have been even greater if the program included proper checking for a valid parameter list.

(2) We could choose a language which is suitable for interpretive execution, is suitable for compilation, or is suitable for either. There are clear advantages to the last possibility. We may note, however, that efficiency in control and manipulation may not be important, since often the greater part of the total time will be spent in executing the commands.

(3) We must decide how to distinguish syntactically between the different parts of the language. One possibility (which is inevitable if we use an existing general-purpose language) is to embed commands within a statement of the control or manipulation language. The disadvantage of this is that it is then impossible to write a small command language program simply as a list of commands. An alternative is to allow the commands to appear 'unclad', in exactly the same form as if they had been typed in interactively, and to distinguish the other parts of the language by some syntactic features which do not occur (or are assumed not to occur) in the commands.

4. A COMMAND CONTROL LANGUAGE

In this section we describe the essentials of a command control language, derived from CMS EXEC. It is not necessarily an 'ideal' language in any sense; it does however have a number of unusual and interesting features, and is easy to use.

The original version of CMS EXEC was developed at the IBM Cambridge Scientific Center around 1967. (See VM/370: Command Language User's Guide for a description of the present version.) For a review of some other command control languages in common use, see Barron (1972). See also Grant (1970), who describes a system, based on SNOBOL, which can issue commands, and then inspect their results by redirecting their console output to a pseudo-console from which they can be read into variables of the command language program.

4.1 Design Decisions

The three 'design choices' of Section 3.2 above are made as follows.

(1) The language is special-purpose, for controlling commands. It has a convenient notation for handling parameters (which are passed by value), manipulating restricted classes of character strings (based on the 'word'), and issuing commands. It will handle only limited types of data and expressions, and is powerless in non-integer arithmetic.

(2) The language contains many inherently interpretive features. A statement is analyzed anew each time it is executed, and it is possible for the same statement to be of a different type on different occasions. This has the disadvantages (a) that compilation would be difficult and at best partial, and (b) that automatic program analysis would be difficult and strongly data-dependent.

(3) Commands can be written in a command language program exactly as if they had been typed in. They are terminated by the end of the line. There is no prologue or epilogue: the name of a command language program is determined by the name of the file containing it (the 'EXEC' file). Therefore a trivial command language program need consist only of a list of commands. This has obvious advantages for small programs and for new users; however, it requires that statements which are not commands be distinguishable syntactically from those that are.

4.2 Main Features of the Language

In this section an attempt is made to convey the flavour of the language. This is followed by the BNF description in Section 4.3, and tying up of the loose ends in Section 4.4.

1. The smallest syntactic unit is a word, where, as before, a word is defined to be a string of contiguous non-blank characters. Thus a word plays the part of what is usually called a 'token' in a lexical analyzer. This is true for both the control and the manipulation components of the language.

2. Literal strings are written without quotation marks. If quotation marks are given, they become part of the string. This is to some extent a concomitant of the decision to allow commands to appear 'unclad', and in this respect the language follows more nearly the conventions of English than of most programming languages.

3. It now becomes necessary to use unconventional notation to distinguish words which are not literals. This is done by using an arbitrary special character, here chosen to be '&', as the first character of all variable names. Wherever this character appears, in any word of any statement, it, with the following string of arbitrary characters to the end of the word, is taken as the name of a variable, and is replaced

by its value. (An exception is made for a variable which is the target of an assignment, in which case its name is retained.) This is done before the statement is executed, and irrespective of whether the statement is part of the control, manipulation or command language. There is here a resemblance to some assembly-language preprocessors; however, we shall see later that the notation is extended to allow indirection, and hence the construction of arrays.

4. Variables are not declared: they become known simply by their use. All variables have a value which is a word, in general of arbitrary length and contents. In some contexts a word is required to represent an integral numeric quantity. Most variables have an initial value of the null string (or null word). The following are however automatically set or updated. They are not <u>reserved</u>, since any automatic updating ceases if they are explicitly set in the program.

```
&1, &2, &3, ...   (arguments received by program)
&NUMARGS          (number of arguments received)
&LINENUM          (current line number of program)
&RETCODE          (return code from last command)
```

5. The statements of the outer and manipulation languages are (except for the assignment statement) distinguished by keywords, which (like variables) start with a special character. It turns out that the <u>same</u> special character can be used for the keywords as is used for variables, without introducing any reserved variables or reserved keywords (see Section 4.4).

6. The control language contains the following statement types:

```
&IF ... [&ELSE  ...]
&DO ... [&WHILE ...]      (iterative or non-iterative)
```

The '&DO' statement has the form:

```
     (number-of-lines)  [(        number-of-times              )]
&DO  {                } [{                                     }]  [&WHILE condition]
     (      label     )  [(var = expr [BY expr] [TO expr])]
```

If 'label' is given, it must be attached to the last statement of the group or loop. A label may be any word which starts with a hyphen and appears as the first word of a line. '&DO' groups (or loops) may be nested.

7. The manipulation language contains the following statement types:

```
variable = ...          (assignment)
&READ ...               (input from console: see Section 4.4, Note 5)
&PRINT ...              (output to console)
&STACK ...              (enter into console input buffer)
&ARGS ...               (reset arguments &1, &2, ..., and reset &NUMARGS
                             to number of arguments thus set)
```

In assignments only, it is possible to use the arithmetic operators

```
+  -  *  / .
```

Evaluation is from left to right. Nested expressions are not supported, and parentheses are not treated as special characters. Also in assignments, it is possible to use the following built-in functions, which must appear <u>after</u> any arithmetic operators:

```
&SUBSTR OF ...          (2 or 3 arguments, as in PL/I)
&INDEX OF ...           (2 arguments, as in PL/I)
&LENGTH OF ...          (1 argument, as in PL/I)
&DATATYPE OF ...        (1 argument; returns NUM or CHAR)
&CONCAT OF ...          (builds one word from several or many)
&LITERAL OF ...         (1 argument; inhibits scan for variables)
&EQUIVALENT OF ...      (1 argument; establishes equivalence)
```

(For a description of the PL/I built-in functions, see the <u>Pl/I Language Specifications</u> given in the References. For '&LITERAL' and '&EQUIVALENT', see Section 4.4, Note 6.)

8. In comparisons, following &IF or &WHILE, the following may be used:

```
=  ¬=  >  ¬>  <  ¬<  >=  <=  &AND  &OR .
```

9. Any line which starts with an asterisk is treated as a comment.

10. Any executable statement which, after substitution for any variables, does not start with an ampersand is taken as a command, and is issued to the active command environment.

11. This section concludes with three examples of command language programs.

(a) The first program, called REPEAT, issues a command which is given in its parameter list a given number of times, or until a non-zero return code is obtained from it. It is not dependent upon the primitives or data of any particular command environment.

```
&IF &NUMARGS = 0
    &GOTO -TELL

* IF INVOKED WITHOUT ANY ARGUMENTS, TELL HIM HOW TO USE IT.
* OTHERWISE, CHECK THAT FIRST ARGUMENT IS NUMERIC, AND >= 0...

&X = &DATATYPE &1
&IF &X ¬= NUM &OR &X < 0
    &GOTO -BADPARM

* MAKE SURE HE HASN'T GIVEN TOO MANY ARGUMENTS...

&IF &NUMARGS > 16
    &GOTO -PARMBUST

* ALL SET TO EXECUTE THE FOLLOWING LOOP '&1' TIMES...

&DO -END &1
    &2 &3 &4 &5 &6 &7 &8 &9 &10 &11 &12 &13 &14 &15 &16
    &IF &RETCODE ¬= 0
        &EXIT &RETCODE
-END

&EXIT 0

-PARMBUST &PRINT 'REPEAT' PARAMETER LIST TOO LONG
&EXIT 102

-BADPARM &PRINT INVALID 'REPEAT' PARAMETER LIST
&EXIT 101

-TELL &PRINT CORRECT FORM IS: REPEAT N COMMAND PARM PARM ...
&PRINT STOPS IMMEDIATELY IF RETURN CODE ¬= 0
&EXIT 100
```

(b) The following is a command language program for a hypothetical editor which (we suppose) does not support a primitive command for moving lines around in the file, but which can 'stack' them in the console input buffer. The command format for the program is:

MOVE m UP/DOWN n

where m and n are integers.

```
* CHECKING FOR VALIDITY OF PARAMETER LIST...
* ...SHOULD GO IN HERE.
DOWN &1
UP 1
&STACK
* CURRENT LINE IS NOW SET TO THE LAST LINE TO BE MOVED,
* AND WE HAVE STACKED A NULL LINE.  THE FOLLOWING LOOP
* RIPPLES UP THE LINES TO BE MOVED, STACKING AND DELETING
* EACH ONE IN TURN.
&DO 3 &1
    STACK 1
    DELETE 1
    UP 1
* NEXT ISSUE 'UP N' or 'DOWN N', AS GIVEN IN PARAMETER LIST...
&2 &3
* FINALLY ENTER INPUT MODE AND READ IN STACKED LINES.
* THE NULL LINE WILL EVENTUALLY THROW THE EDITOR BACK INTO EDIT MODE...
INPUT
* EXIT (WITH RETURN CODE = 0) BY DROPPING OFF END OF FILE...
```

(c) The last example is not strictly a command language program, since it does not have any net effect on any command environment. It is simply an example of string manipulation. Its effect, when executed, is to type a line of the form:

11:30 AM - 15 OCT 1973.

The program is as follows.

```
* STACK THE DATE AND TIME, BY ISSUING 'STACKDT', WHICH
* IS A PRIMITIVE COMMAND...

STACKDT

* GIVE UP IF SOMETHING WENT WRONG...

&IF &RETCODE ¬= 0
    &EXIT &RETCODE

* READ MM/DD/YY HH:MM:SS, EXTRACT DAY, AND STRIP POSSIBLE
* LEADING ZERO...

&READ VARS &DATE &TIME
&DAY = &SUBSTR OF &DATE 4 2
&DAY = &DAY + 0

* EXTRACT NUMBER OF MONTH, AND USE IT AS AN INDEX TO OBTAIN
* NAME OF MONTH...

&J = &SUBSTR OF &DATE 1 2
&ARGS JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
&MONTH = &&J

* THE REST OF THE PROGRAM IS SELF-EXPLANATORY...

&YEAR = &SUBSTR OF &DATE 7
&HOUR = &SUBSTR OF &TIME 1 2
&HOUR = &HOUR + 0
&MINUTE = &SUBSTR OF &TIME 4 2

&IF &HOUR < 12
    &AMPM = AM
&ELSE
    &AMPM = PM

&IF &HOUR = 0 &AND &MINUTE = 0
    &AMPM = MIDNIGHT

&IF &HOUR = 12 &AND &MINUTE = 0
    &AMPM = NOON

&IF &HOUR = 0
    &HOUR = 12

&IF &HOUR > 12
    &HOUR = &HOUR - 12

&TIME = &CONCAT OF &HOUR : &MINUTE
&PRINT &TIME &AMPM - &DAY &MONTH 19&YEAR
```

4.3 BNF Description

The following is a BNF description of the combined control and manipulation language.

```
<program>      ::=   <line>
                   | <program>  <line>

<line>         ::=   <statement>  <end of line>

<statement>    ::=   <comment>
                   | <label>  <executable stmt>
                   | <executable stmt>
```

```
<comment>                 ::=   *  <anything>

<executable stmt>         ::=   <null>
                              | <if stmt>
                              | <unconditional stmt>

<if stmt>                 ::=   &IF  <condition>  <unconditional stmt>
                              | &IF  <condition>  <unconditional stmt>
                                    &ELSE  <unconditional stmt>

<unconditional stmt>      ::=   <assignment>
                              | <keyword stmt>
                              | <command>

<assignment>              ::=   <variable>  =  <right-hand side>

<right-hand side>         ::=   <null>
                              | <word>
                              | <expression>

<expression>              ::=   <number>
                              | <built-in fn ref>
                              | <number>  <operator>  <expression>

<built-in fn ref>         ::=   <built-in function>  OF  <argument list>

<operator>                ::=   + | - | * | /

<condition>               ::=   <word>  <comparator>  <word>
                              | <condition>  &OR   <condition>
                              | <condition>  &AND  <condition>

<comparator>              ::=   = | ¬= | > | ¬> | < | ¬< | >= | <=
```

A 'keyword statement' is any statement which starts with a keyboard, other than the '&IF' statement. (The statements comprising an '&DO-group' are of course treated in the syntax as a single statement.) A 'number' is a positive integer, with or without sign, or a negative integer. The meanings of 'command', 'variable', 'word', 'built-in function' and 'keyword' are described in Section 4.2. '&AND' has higher precedence than '&OR'.

4.4 Language Description, continued

In this section we conclude the language description.

1. A statement is terminated by the end of line, except that the '&IF' clause and the '&ELSE' keyword may be followed by a statement on the next line.

2. In Section 4.2 it was stated that variable names which appear in a statement are replaced by their values before the statement is executed. The algorithm can be described as follows.

(1) Each word is scanned for ampersands, starting with the rightmost character of the word and proceeding to the left.

(2) If an ampersand is found, then it, with the rest of the word to the right, is taken as a variable name, and replaced by its value.

(3) Scanning then resumes at the next character to the left, and the procedure is repeated from (2) above, until the word is exhausted. An exception is made if the word is the target of an assignment or an input statement: in this case scanning for ampersands effectively stops on the second character of the word. (This is the same rule that is applied in conventional high-level languages; e.g. the old value of X is disregarded in the ALGOL assignment X:=Y .)

(4) If in the process of substitution for variables a word is reduced to the null string (or null word), then it is deleted from the statement, and the next word is deemed immediately to follow the previous one.

Note that any characters which are substituted for variables are not scanned for ampersands. (They will however be included in the next name if another ampersand is found to the left.) This prevents indefinite looping during substitution.

This algorithm makes it possible to get the effects of subscripted variables. A simple illustration appears in example (c) above, where the name of the month is extracted from an array containing JAN, FEB, ..., DEC.

3. The keywords of the language are represented by variables of the same name which are initialized to their own names. They can in fact be reset, and the function which they provide then

becomes 'lost' (unless another variable is set to the value of the keyword). This has the advantages that users need not be aware of all the keywords, and new keywords can be added to the language without invalidating old programs.

4. The assignment statement (as seems often to be the case) presents a problem in the language. Does the statement

```
&PRINT = X
```

mean 'print "= X"', or 'assign "X" to "&PRINT"'? The problem could be resolved by introducing a keyword for assignments, e.g.

```
&SET &PRINT = X .
```

There is, however, a reluctance to do this for reasons of human engineering. An alternative is to reserve the '=' sign in the sense required by the following rule.

> If the first word of a statement starts with an ampersand and the second word is a literal equal sign, then (and only then) the statement is taken to be an assignment.

Then to print '= X', one could write:

```
&EQ = =
&PRINT &EQ X
```

5. The '&READ' statement can take any of the following forms.

&READ n

> Read n lines from the console and execute them individually as if they had been in the program. Reading ceases if an '&DO' statement or a statement which transfers control is encountered. The number of reads outstanding can be incremented or decremented by entering another '&READ n' statement.

&READ ARGS

> Read a line and reset the arguments &1, &2, ..., to the words in it, without scanning it for variables; and reset &NUMARGS to the number of arguments thus set.

&READ VARS [var1 [var2 ...]]

> Read a line and set the variables 'var1', 'var2', ... , to the words in it, without scanning it for variables.

6. '&LITERAL' and '&EQUIVALENT' are a pair of built-in functions which inhibit scanning for variables, or delay the scan. '&LITERAL' enables ampersands to be included in the value of a variable; for example

```
&NAMEX = &LITERAL OF &X
```

gives the variable &NAMEX the value '&X'. For '&EQUIVALENT', consider the statement

```
&RC = &EQUIVALENT OF &RETCODE .
```

Henceforth, whenever &RC appears other than as the target of an assignment (or &READ VARS) it will be replaced by the current value of &RETCODE.

7. A leading plus sign, and leading zeros, can be removed from an integral numeric quantity by performing any arithmetic operation on it. This is illustrated in example (c) above.

8. Comparisons are numeric if both comparands are integral; otherwise they are treated as character strings, and the comparison is done according to their internal representation. To force a character-string comparison between two words which may also be valid numbers, an arbitrary non-numeric character can be put at the front of each word, e.g.

```
&IF  /&1  =  /02  ...
```

9. When an error is found, a message is printed describing it and its location, and an implicit '&EXIT error-code' statement is executed, where 'error-code' has a defined value for each of the possible error conditions.

4.5 Limitations and Possible Extensions

1. All variables are local to the command language program: there are no global variables. This forces data to be transferred between programs in parameter lists, via the input buffer, or through the file system. This can be inconvenient, but is almost essential in a situation where a command language program is (almost in principle) unaware of what other programs may be 'dormant' in the execution stack.

2. There are no internal procedures, and there are no user-defined functions. To add them would unfortunately involve significant extra complexities in the language. On the other hand, it would be easy to define an internal subroutine call without parameters: it can in fact be done already, thus:

```
&RETURN = &LINENUM + 2
&GOTO -SUB
...
-SUB
...
&GOTO &RETURN
```

3. The manipulation language, as it stands, is convenient only for manipulating lines which are comprised of words. There is, for example, no ability to embed multiple blanks in a line, or to print or stack a complete line exactly as it appears in the program. This is an area for possible extensions. (A partial solution is given by the '&BEGPRINT' and '&BEGSTACK' statements of CMS EXEC.)

4. The language as it stands is incapable of issuing a command which starts with an '&'. This could be made possible by introducing an '&COMMAND' statement of the form:

```
&COMMAND  statement
```

which issues 'statement' as a command, irrespective of its syntax.

5. There are two kinds of 'ON' statement which would be useful, one to deal with errors, overriding the default action of Section 4.4, Note 9, and the other to specify special action after return from a specific command. Possible keyword statements are:

```
&ON     ERROR          statement
&AFTER  command-name   statement
```

where 'statement' may not be an '&DO' statement, but may transfer control. (If it does not transfer control, return will be made, after its execution, to the line following the point of interruption.) In particular, 'statement' may contain an '&IF' clause:

```
&AFTER LIST &IF &RETCODE ¬= 0 &EXIT &RETCODE .
```

To go along with these statements, there could be the special variables:

```
&ERRORCODE      (contains error code)
&ERRORLINE      (contains line number of
                     statement in error)
&COMLINE        (contains line number of
                     command)
```

and possibly the additional keyword statements:

```
&STACKERROR     (stack the statement in error)
&STACKCOM       (stack the command).
```

5. DISCUSSION

In Section 4 we have given a possible answer to question (A) of Section 3, viz. 'What language or languages should be used for writing command language programs?' We now address the remaining questions in the context of that answer.

5.1 Invoking a Command Language Program

One of our premises concerning the system was that programs could reside on file and be invoked by typing their name as the first word of a command issued to the SYSTEM command environment. Our examples of Section 2 were assumed to be of primitive commands. However, none of our premises or proposals concerning the command language depend upon the language of implementation of a program, and it is therefore possible to bring over all of our statements on primitives and apply them equally to command language programs. In particular, a command language program can be invoked in exactly the same way as a primitive, and can invoke another command language program (or itself recursively) in exactly the same way as it can invoke a primitive.

It now becomes necessary to define what will happen if a command name is matched by both a primitive and a command language program. One possibility is to exclude this possibility, i.e. to prevent the creation of two programs with the same name. Perhaps a more useful solution is to define a search rule, such as to look for a command language program first, and look for a primitive only if the first search fails. To override this, when required, there could be a primitive called (say) PRIMITIVE which executes its parameter list as a primitive command.

5.2 Generality of Command Language Programming Facilities

It would seem that, provided that the necessary links and mechanisms exist, the command language programming facilities described here could be used with all the command environments in a system. The facilities do not depend upon the syntax or the function of the commands being controlled, provided only that a command is a line consisting of a character string which is 'issued' to an identifiable command language interpreter. The facilities which have been described here will, however, probably be convenient only for command languages which are more or less word-oriented.

If the same command language programming facilities are to be used for the control of more than one command environment, then there is a naming problem, which has two aspects: (a) a command language program should in general be associated only with the command environment for which it is intended, e.g. it should not be possible inadvertently to execute an EDIT command language program (which will attempt to issue edit commands) from the SYSTEM command environment; (b) a command language program, once invoked, should be able to issue commands only to the command environment from which it has been invoked: so that anything which can be done by typing in commands to a command environment can also be done by issuing commands from a command language program; and, conversely, anything which cannot be done by typing in commands cannot be done from a command language program.

A possible resolution to the naming problem makes use of 'primary' and 'secondary' file names, as in CTSS (continued in CMS as 'filename' and 'filetype'). Each command language program could have a secondary name (or 'filetype') which associates it with one command environment, both for its invocation and for the invocations from within it.

5.3 Repercussions on the Design of Command Environments

1. In the foregoing, we have assumed the feasibility (a) of going outside the program supporting a command environment to find a command language program on file, and (b) of issuing commands from a command language program back to the command environment. In fact, the ability to do these things requires cognizance by the program supporting the command environment that command language programs may exist, and mechanisms for passing control, receiving control for the execution of a single command, and giving return codes. These unfortunately require some extra complexity in every program which sets up a command environment and is to permit the facilities to be used.

2. In the scheme which has been proposed here, there are facilities for passing data, via parameter lists, from a command language program to the associated command environment. The facilities for extracting data from the command environment to a command language program are however much weaker: the only datum which comes automatically is the return code. One way of transferring data in this direction, which has been used extensively in some experimental command language programming, and which has been illustrated in the examples of Section 4.2, Note 11, is to stack them in the console input buffer, so that they can then be read into variables of the command language program. It is therefore useful to include in every command language primitives which will stack any data from the command environment. In fact, if a command environment provides basic functions for adding, modifying, deleting and stacking arbitrary data at arbitrary locations in its data areas, then it is possible in principle to define all other functions in terms of these, and therefore to implement all but the most basic commands in the form of command language programs.

3. The existence of command language programs lends still more weight to the advantages of modular command environments (as described in Section 2.2), compared with the other extreme in which all commands are directly accessible at the same time. For, given a modular structure, existing command language programs are not affected by the addition of new interactive programs to the system; and the removal of an old program will have repercussions which are fairly easily delimited.

5.4 Final Remarks

1. Since the command programming language is itself an interactive programming language, it can be used to write programs which support their own command environments. In fact, the word-handling features of the language proposed in Section 4 make it in some ways particularly convenient for this. We may then go on to ask whether we could write command language programs for such a command environment; and here we find that there is a needed function missing from the language. This is the ability to receive control (from a descendent command language program) for the execution of a single primitive command of the environment, and to give back a return code. This could be dealt with by adding the keyword statement:

```
&ENTRY statement     (set re-entry action)
```

and by using '&EXIT' (in addition to its existing function) to return control, with a code, following re-entry.

2. In this paper we have made use of a console input buffer which acts as a stack. For some purposes it would be more convenient if it acted instead as a queue, so that a sequence of lines could be deposited in the same order as they were later to be read. Unfortunately this can produce incorrect results if the buffer is not empty before the lines are deposited. A possible strategem to deal with this would be to allow the creation of multiple buffers, each permitting lines to be deposited on either end (for collection FIFO or LIFO). Each new buffer would be created by command, and 'stacked' on the existing ones; and a buffer would be automatically 'unstacked' when it was exhausted.

3. Given command language programming, as outlined here, any task which can be done interactively can also be programmed to run non-interactively. Each conversation can be replaced by a programmed sequence of commands; and the first command to a new command environment (which invokes the program of commands) can be stacked in the input buffer before the command environment is activated. This method has been used with success to run CMS in 'batch' mode. The system was modified simply (a) to read the initial commands from the card reader (instead of from the console) and deposit them in the console input buffer, and (b) to terminate a 'job' when there is an attempt to read a line from the console and the input buffer is empty.

4. It can often happen that a command language program wishes to suppress printing generated by commands issued from it. In the case of a question-answer conversation, for example, there is no sense in the questions being printed if the answers are supplied from a program. This can be accomplished by means of a command which sets or resets a flag in the console output routine, to suppress or resume subsequent printing. A variant of this could divert console output to the input buffer, so that the command language program could read it and examine it. (This is effectively the technique used by Grant (1970).)

5. We have not here addressed the handling of interrupts from the user. None of the ideas treated in this paper seems to have any particular relevance to them. For completeness, however, we may note that there is no special difficulty in directing named interrupts to any program which is resident in the system, in order (say) to modify its behaviour. (See Dolotta and Irvine, 1968.) Obtaining the desired results from the interrupt is more difficult. In general, either the system (hardware or software) must support multiple levels of interrupt control, so that the program which is to be affected does not receive the interrupt until it has indicated that it is ready to do so; or the immediate effect of the interrupt must be confined to the setting or resetting of a flag which is then polled by the program. Either way, consistent results will in general be obtained only if the program chooses an appropriate point in its execution to effect the modification, and (in some cases) does the correct cleaning up.

6. In this paper we have restricted ourselves to interactive command environments. It would, however, be possible to declare other well-defined input streams, such as the input to a compiler, as command environments, and then use the command language programming facilities for writing source language macros. (See Leavenworth, 1966.) This would appear to be viable for the class of macros which require only a single pass, e.g. do not involve the building of a common symbol table.

REFERENCES

Leavenworth, B. M., Syntax Macros and Extended Translation, CACM, 9, 790 (1966).

Wilkes, M. V., Computers Then and Now, JACM, 15, 1 (1968).

Dolotta, T. A. and Irvine, C. A., Proposal for a Time Sharing Command Structure, IFIP Congress, 1968, C14.

Grant, C. A., An Interactive Command Generating Facility, CACM, 13, 403 (1970).

IBM, PL/I Language Specifications, GY33-6003-2 (1970).

Barron, D. W. and Jackson, I. R., The Evolution of Job Control Languages, Software, 2, 143 (1972).

IBM, VM/370: Command Language User's Guide, GC20-1804-0 (1972).

Kelley, R. A., APLGOL, An Experimental Structured Programming Language, IBM J. of Research and Development, 17, 69 (1973).