VW: A Small But Potent
Machine-Independent Text Editor

Nicholas F. Vitulli

Technical Report No. 49

VW: A Small But Potent
Machine-Independent Text Editor

Nicholas F. Vitulli

April, 1976

Submitted to the faculty of the Graduate School in partial fulfillment of the requirements for the degree Master of Science in the Department of Computer Science, Indiana University.

VW: A Small But Potent Machine-Independent Text Editor

Nicholas F. Vitulli

May 1976

Submitted to the faculty of the Graduate School in partial fulfillment of the requirements for the degree Master of Science in the Department of Computer Science, Indiana University.

VW: A Small But Potent Machine-Independent Text Editor

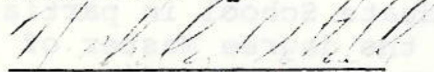
Nicholas Vitulli

Abstract:

An on-line interactive text-editor has been designed and implemented, in FORTRAN, on the DEC PDP-10 and CDC 6600. Its design advantages are ease of use, editing versatility, and modest core requirements. It is also potentially suitable for use on minicomputers such as the DEC PDP-8 and the TI 980.

Ease of use is established by confining the command set to a few basic commands but permitting the range of operations to be anywhere from a single character to the entire file contents. A versatile macro control structure is incorporated by considering each command a predicate, associated with a truth value indicating the success or failure of the predicated operation. The entire PDP-10 version of the editor is contained in only 7K 36-bit words of memory, which includes nearly 3K of core for the unshareable data structure.

Submitted by:



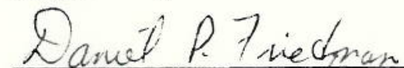
Nicholas F. Vitulli

Date

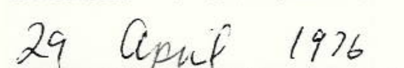
Accepted by:

David S. Wise

Date



Daniel P. Friedman



Date

I. Introduction

In interactive processing the creation and modification of on-line files from a console replaces hand punching and editing of card decks. The text editor is a utility program which has two primary functions: to translate and store keyed-in characters as machine readable codes, and to permit the alteration of stored text--insertion, deletion and movement of characters.

Although the term "editor" has been applied to programs such as linking-loaders and formatted manuscript generators, in this paper it refers only to programs creating and modifying files through user interaction. Existing editors can be categorized either as line-oriented (e.g., LINED (1)) or character (context)-oriented (e.g., TECO (2)). In line-oriented editors, the text of interest is located by line number, and only complete lines may be added, deleted or changed. The advantage of such an editor is simplicity of use; the disadvantage is its relative inflexibility. In character-oriented editors, single characters (including the end-of-line character) may be added, deleted or changed; areas of interest may be located by counting the number of characters or lines, or, alternatively, by searching for a sequence of characters. The obvious advantage is the flexibility afforded by allowing modifications to portions of a line; the attendant disadvantage is complexity of operations.

One editor, SOS, does attempt to combine the features of both line- and character-oriented editors (3). While the importance of interactive editors has long been recognized (4,5) no standard editor has been adopted. Few have been designed to be machine-independent (6), and the majority rely on specific hardware configurations.

An effective editor should maximize programmer productivity by providing simple commands for trivial editing, as well as supporting sophisticated commands for more complex jobs, without taxing the interactive system environment. It should also provide ease of operation, power and security, where:

<u>Ease of Operation</u> requires a concise, consistent, convenient command set which also supports easy recovery from typographic errors.

<u>Power</u> pertains to the ability of single commands to perform simple editing, or for combined commands to effect more complex modifications.

<u>Security</u> guarantees that changes to a file will be made permanent only when requested by the user, and that the file will be protected from system malfunction.

<u>Compactness</u> refers to minimization of demands for critical resources made on the host system.

Design of VW

VW was designed to be a character-oriented, machine-independent interactive text editor. Its design was strongly influenced by two other character-oriented editors, TECO and COMPATIBLE (7). The design goals described above were achieved as follows:

Machine Independence: source language to be FORTRAN.

Ease of Operation: simple, small command set with clearly defined and logically consistent syntactical structure.

Power: simple editing tasks accomplished by single commands operating on characters or lines. Complex tasks accomplished by combination of simple commands into larger, macro-commands, as well as access to an auxiliary memory buffer for use as a scratch pad.

Security: disjoint input and output files are used during the editing process.

Compactness: a small linked, unshared structure driven by a structured re-entrant program. Time is not considered a critical resource because the editing process is normally I/O bound.

Many editors, KRONOS EDITOR (8), for example, will not allow the combining of commands. Because of this, an editing task which requires more than one primitive function must be

expressly repeated. This repetition prevents the user from describing a complex editing operation which is to be performed in many places. The two character editors, TECO and COMPATIBLE do not share in this restriction since they both allow combinations and repetitions of commands but in neither case is their control structure strong enough to search for flexible patterns in the file.

The use of sequential files is a distinct advantage over in place or random editors in which changes are made permanent immediately (9). In the two-file-structure changes are made by the user in the output file without damaging the integrity of the input file. Only on request is the output renamed as the edit file, leaving the original file available to the user as a backup.

Implementation of VW

VW is an interactive character-oriented editor, implemented in FORTRAN, and presently available on the CDC 6600, DEC PDP-10, DEC PDP-11 and HARRIS 6024/4. It is supported by any type of machine-readable sequential storage (even paper tape) and an interactive console. The full ANSI character set is allowed, restricted only by the particular host system. Its minimal core requirements permit implementation on a mini-computer and place no burden on an interactive timesharing system.

The remainder of this paper is in five parts. Two sections discuss data structure and the types of text manipulation available. The next two sections present the formal definition of the operating language and examples of _VW_'s power in solving problems. In the concluding section the potential of the universal application and acceptance of _VW_ is discussed.

II. Data Structure

The single most important factor influencing the success of a text editing system is the method used to allocate and access work space. The input file, a sequential list of characters from external storage, is read into this working area where it is modified by the user. To facilitate the editing process at the character level a linked-list (10) data structure containing one character per node is used by VW. The linking is bi-directional to allow traversal through the structure in both forward and reverse directions. The actual data structure employed is a "two-way" linked list and not a doubly linked list (11), since two link fields are not used. This technique allows space reduction without functional loss by linking a linear list with only one link field per node as described by D. E. Knuth (12) and expanded in detail by D. S. Wise (13).

The top of the data structure is linked to external sequential output (e.g., disk, magnetic tape, papertape, etc.) and the bottom linked to sequential input. This input, the edit file, is read into the linked memory buffer where it is modified. The output file, the result of the editing process, reflects the changes made.

The linked memory buffer is the window into the edit file. Movement of the file through the window is called

"windowing". Input text is read from the edit file and linked into the window for editing. As additional text is required the window expands until available space is exhausted. At this point the top portion of the window is written to the output file, thereby freeing space for additional input. This windowing process offers advantages over "paging" (14), a method which replaces the entire window with the new text, by retaining backwards context for the added text.

The <u>cursor</u> is a pointer within the window, positioned between (13) two characters. All processing performed in the window affects the characters adjacent to the cursor. Windowing occurs when editing is performed in the right (or forward) direction and the cursor reaches the bottom of the window. When reverse editing is requested and the cursor is at the top of the window, additional processing is impossible since the required text has already been written to the output file.

An alternate two-way linked structure called the auxiliary window is also available to the user for text storage and manipulation. The auxiliary window has its own cursor and is connected to an optional read-only secondary file. It functions like the main window but has no output capability. Characters may be transferred or copied between the auxiliary and main windows.

A third two-way linked structure, called the command buffer, is used by the VW processor to hold the current command entered by the user.

All three structures (main window, auxiliary window, command buffer) and their cursors share the same available free space and because they are linked in the same manner, they are processed internally by the same routines, thereby reducing implementation core size. Figure 1 summarizes these structures.

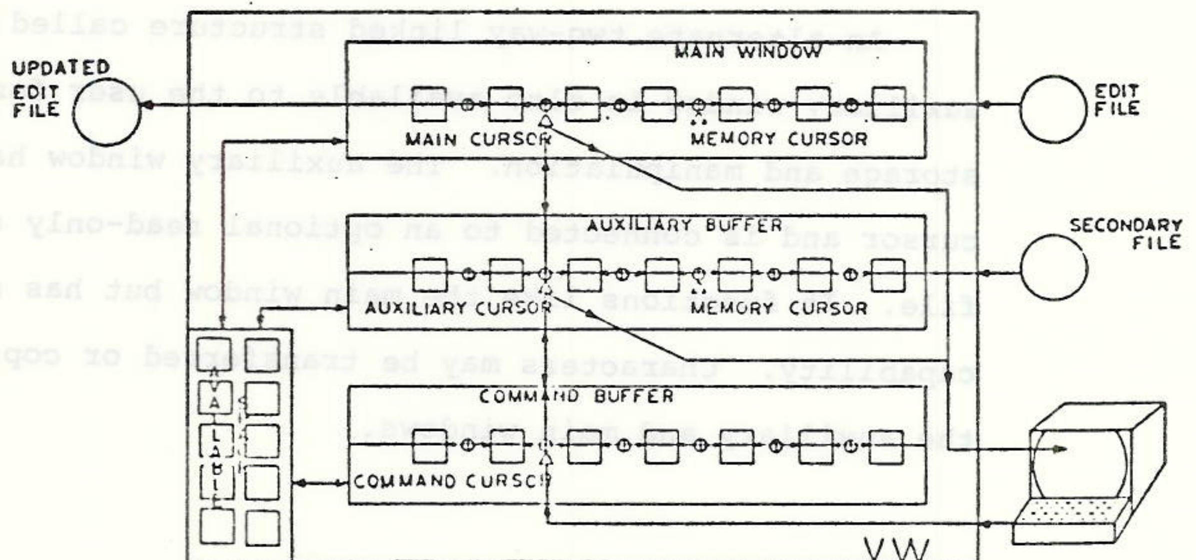


Figure 1. VW Memory Allocation and File Organization

III. Text Manipulation

VW supports text insertion, deletion, searching (15) and copying, all within a simple, concise command structure. Commands and their associated arguments are entered in a syntax that has been designed for user convenience. Modifiers may be used in conjunction with commands to limit range of operations or processing sequence. Finally, conditional execution is available to support complex editing tasks provided through a complete macro structure.

Command Structure and Function

The VW command language consist of concise one-letter commands which may take arguments. Multiple commands, optionally separated by blanks, may be entered on a single line. The new line character terminates command input, signalling the VW processor to execute the completed command string.

Command action begins at the cursor location and proceeds forward (i.e., left to right) through the file unless otherwise specified. A repetition factor associated with each command indicates the number of times that command is to be re-executed.

The first class of commands is character-oriented, consisting of those operations which act on single

characters. The editing functions provided by character commands include cursor movement, text insertion, deletion, display and copying.

The second class of commands is line oriented. The editing functions provided include cursor movement, and display, deletion, and copying of entire lines of text. These functions are analogous to the character-oriented commands, but the scope is lines rather than characters.

Additional commands provide cursor movement by means of character string searching and character string verification. All these commands are executable in both the main and auxiliary windows. (Either the main or auxiliary window may be active at any time according to user directive. The copy commands copy characters or lines from the active window to the inactive.)

Processing Scope

A processing limit is associated with all commands. This limitation, called command scope, restricts the amount of text that serves as the command's operand. The scope may encompass either the entire file, a fixed number of lines, or all characters up to a pre-determined stopping point. The stopping point may be an explicit marker, placed when cursor was at that point earlier, or it may be determined by the closing bracket which balances the open bracket found at

the current cursor position. The bracket pair (parentheses, angle brackets, etc.) is set by the user.

Conditional Constructions and Macro Structure

Truth values are associated with all commands and are returned by VW following command execution, indicating whether or not the command has been successfully executed. For example, if the repetition factor asks the cursor to be moved beyond the scope of the command, processing halts and a FALSE value is returned. However, if the repetition count can be satisfied within the scope, a TRUE value is returned and the next command in the input string is executed. (Arguments are available to reverse or reset the truth value to be returned.)

Commands and their associated arguments may be grouped within parentheses to form additional editing functions. The grouping is called a macro command. The macro may have its own repetition factor, direction and truth operators. Execution of a macro halts, returning FALSE as a value as soon as one of its constituents fails. That is, when a command within a macro fails, execution is interupted. Arguments of the macro may reverse this value so that the next command in input string following the macro is executed.

IV. VW Syntax and Semantics

We assume that the reader is familiar with BNF notation (16). The following symbols are defined to mean:

::= is defined as

| or

( ) optional

{ } grouping

' ' terminal symbol

Δ space

↵ new line (the carriage return, line feed sequence is considered to be one character)

INPUTLINES ::= INPUT '↵'

All commands to VW are input at the user's console. VW prompts this action by displaying the character ':'. The user terminates his input instruction with the '↵' character.

INPUT ::= ERRORINPUT

If an error is made while entering a command, VW is signaled to ignore the input when the character '+' is included before the '↵'.

INPUT ::= SPECIALCMDS
SPECIALCMDS ::= REPEAT SAVE INDIRECT

There are three special commands provided for by VW which operate on commands. Each must follow the prompt immediately and be the only command given. (Any other input following will be ignored.)

REPEAT ::= '='

Re-execute the previous command.

SAVE ::= '/'

Store the previous command for future processing.

INDIRECT ::= '↑'

Process a stored command.

INPUT ::= CMDSTR
CMDSTR ::= CMD
CMDSTR ::= CMD CMDSTR

Normal input to VW is a command string which is comprised of several commands.

CMD ::= CMDARGS CMDBODY

An individual command is generally specified in two parts. The command arguments are used to qualify the editing action while the command body describes the action to be taken.

CMDARGS ::= (BLANKS) (TRUTHREVERSE) (TRUTHFORCE) (DIRECTION) (REPSCOPE)

Each of the command arguments is optional however if used, the relative positional ordering must be observed.

BLANKS ::= 'Δ'
BLANKS ::= 'Δ' BLANKS

Spaces may be added for command readability. They are ignored by the VW processor.

TRUTHREVERSE ::= '$'

Truth reversal is indicated by the addition of the character '$'.

Semantics

All commands to VW return a truth value (TRUE or FALSE) after processing. This value is used as a control check during the execution of a command string. If a command returns FALSE processing is interrupted. When the '$' is included, a command which normally would return FALSE returns TRUE, and a command which would be TRUE becomes FALSE.

TRUTHFORCE ::= 'A'

A command will always return TRUE when preceeded by the character 'A', read "always".

Semantics

The argument 'A' performs the command but forces the value to be TRUE. A command whose arguments contain both characters '$A' will always return FALSE.

DIRECTION ::= '-'

Direction reversal is indicated by the addition of the character '-'.

Semantics

The editing process will normally proceed from left to right and from top to bottom of the edit file. The direction of process is altered to go bottom to top, right to left with the addition of the character '-'.

REPSCOPE ::= LONEREP|LONESCOPE|REPANDSCOPE

The last two command arguments specify the number of times the editing action is to be repeated and the amount of text which may be processed. They are referred to as the repetition factor and scope of the command.

Semantics

The amount of text or scope is specified with each command. If the editing action can be repeated the requested number of times within the scope the command returns the value TRUE. If the repetition factor cannot be satisfied within the scope the command fails returning FALSE. In this case however processing will be completed within scope specified.

```
LONEREP ::=  Quantity(',')
QUANTITY ::=  NUM|'*'
NUM ::=  DIG
NUM ::= DIG NUM
DIG ::=   'Ø'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

A repetition factor is either a number or the character '*'. It is separated from the scope by the character ','. The comma may be used at all times, however, it is only necessary if ambiguities arise without it.

Semantics

If the '*' repetition factor is used the command will repeat as necessary until the entire scope has been processed. When this is completed the value TRUE is returned. This action is obvious when the '*' is interpreted as Kleene star which means any number of times, even zero.

If the repetition factor is non-specified or 'Ø' a repetition of one is assumed.

```
LONESCOPE ::= ',' SCOPEQUANTITY|(',')(UNTIL|BALANCE|'.')
SCOPEQUANTITY ::= NUM('.')|'*'
UNTIL ::= 'U'
BALANCE ::= 'B'
```

A scope is specified by a number, a '.', a number followed by a '.', an '*', a 'U' (read "until") or a 'B' (read "balanced"). Again the comma preceeding the scope is necessary only to avoid ambiguity.

Semantics

When scope is a number it refers to the number of ' ' characters which may be processed during command execution. A scope of 'Ø' therefore would allow processing of the current line only since no line terminators could be processed. A scope of '1' allows the current line and next line to be processed. When the '.' is appended to the number, the scope is extended to include the last line terminator, but not beyond. A scope of 'Ø.' is the present line plus its terminator. An '*' scope allows processing of the entire file. A 'U' scope allows processing until a previously saved character position. A 'B' scope allows processing to continue over a balanced expression. (The bracketing characters such as parentheses or angle brackets are previously defined.) The balance count is set to Ø with each new command string. When a command using the 'B' scope is being executed and an open bracket is processed the count is incremented, if a closing bracket is processed the count is decremented. When the count becomes Ø or negative processing is terminated.

A non-specified scope is set to 'Ø'.

```
REPANDSCOPE ::= NUMNUM|OBVIOUSREP|OBVIOUSSCOPE
NUMNUM ::= NUM ',' NUM ('.')|','
OBVIOUSREP ::= '*' (',') {SCOPEQUANTITY |'.' |UNTIL|BALANCE }
OBVIOUSSCOPE ::= NUM (',') {'*' |'.' |UNTIL |BALANCE }
```

When a repetition factor and scope are both specified the comma is necessary only to avoid ambiguity.

```
CMDBODY ::= SIMPCMD '(' CMDSTR ')'
```

Command bodies describe the editing actions to be taken. They are qualified by command arguments as described above. Command bodies are of two forms: Simple and Macro. Simple commands are single action primitive commands. A macro command is a combination of simple commands with their arguments grouped within the characters '( )'. A macro may have command arguments of its own and contain other macros within it.

Semantics

All command arguments apply to macros with the exception of scope. <u>If a scope is specified it is ignored</u>. If the direction of a macro is reversed by the character '-' all commands within its bounds are reversed. The other command arguments, repetition factor and truth operators, apply to the macro itself. A macro fails (return FALSE) if one of its constituents fails.

SIMPCMD ::= CHARCMD|STRCMD|LINECMD

Simple commands are divided into three classes: character commands, string commands and line oriented commands.

CHARCMD ::= '<'|'>'|'D'|'E'|'W'|'R'|'X'|'P'|':'|'?'

Character commands are replicated for each character.

Semantics

- < move cursor one character to the left.
- > move cursor one character to the right.
- D DELETE character to right of cursor.
- E ENGROSS one character to inactive window.
- W WRITE the character to the right of cursor.
- R RESTART VW, making all changes permanent by closing and reopening files.
- X EXIT VW (normal termination).
- P save cursor POSITION (-P will forget the saved position).
- ? confirm cursor may be moved one character to right (the ? is implied with the closing parenthesis of a macro).
- : open main window (-: will open auxiliary window).

STRCMD ::= STRLET STRING

String commands take an addition argument called a string.

STRING ::=A sequence of characters beginning and ending with the same character (which acts as a delimiter) but with no other occurrence of that character. The '↵' may be part of a string but may not be used as a delimiter.

The characters '↵' and '+' are treated as any other character when they are contained within a string. They do not cause input to be terminated or aborted.

STRLET ::= 'I' |'F' |'L' |'V' |'S' |'Z'

Semantics

I/str/ INSERT "str" into file to the left of cursor.
F/str/ FIND 'str" in rest of line.
L/str/ LOCATE "str"in rest of file.

V/str/ VERIFY "str" is to right of cursor, without moving the cursor.
S/str/ SUBSTITUTE "str" for target of last F or L command. command
Z/ <>/ SET bracket characters for B scope (default Z/()/).

For F V and L COMMANDS an empty target will reuse the last target string.

LINECMD ::= 'M' |'K' |'C' |'T'

Line mode commands process full lines at a time.

Semantics

M MOVE cursor to beginning of next line.
K KILL rest of line.
C COPY rest of line to inactive window.
T TYPE rest of line.

Line mode commands have a default scope of the entire file and therefore do not fail (a TRUE value is always returned). The repetition factor specifies the number of line to be processed. The relative line numbering conventions are as follows: The prefix of the current line from the cursor back to the last new line character is line -1. The suffix, from the cursor to the next new line in line 1. Line -2 is the previous; line 2, the following, etc.

An '*', 'U' or 'B' are used instead of the numeric repetition factor to indicate the following:

'*' process the entire file.
'U' process until a previously saved character position.
'B' process a balanced expression.

Command Summary

	COMMANDS
C	COPY rest of line to inactive window
D	DELETE character to right of cursor
E	ENGROSS one character to inactive window
F/str/	FIND "str" in rest of line
H	type this text
I/str/	INSERT "str" into file to the left of cursor
K	KILL rest of line
L/str/	LOCATE "str" in rest of file
M	MOVE cursor to beginning of next line
P	save cursor POSITION (see U qualifier)
R	RESTART VW, making all changes permanent
S/str/	SUBSTITUTE "str" for target of last F or L command
T	TYPE rest of line
V/str/	VERIFY "str" is to right of cursor
W	WRITE the character to the right of cursor
X	EXIT VW (normal termination)
Z/<>/	SET bracket characters (default Z/()/) (see B qualifier)
>	move cursor one character to the right
<	move cursor one character to the left
+	abort command string without executing any commands
?	confirm cursor may be moved one character to right
:	open main window (-: will open auxiliary window)
↑	execute next command from auxiliary window
/	save last command string in auxiliary window
=	re-execute previous command string

	QUALIFIERS
-	perform command in the opposite direction
*	perform command as many times as possible (*=infinity)
15	perform command 15 times (for any integer)
A	force command to succeed and return true
B	perform command for BALANCED delimiters (see Z command)
U	perform UNTIL memory cursor (see P command)
$	reverse truth value of command
.	allow command to process just beyond the new line '←↓'

For string COMMANDS any character may be used where the "/" was used. For F V and L COMMANDS an empty target will reuse the last target.

Buffer Commands

As illustrated in Figure 1, the three memory structures share common available space and are linked in the same manner, each with its own cursor. Editing instructions, entered at the terminal, are linked in the command buffer where they are interpreted by the VW processor. The ':' command (default) activates the main cursor and window for the editing process. The '-:' command activates the auxiliary cursor and buffer. Communication between the structures is accomplished by the 'E' and 'C' commands which copy characters from the active to the inactive buffer. This method of data transfer allows the auxiliary buffer to be used as temporary storage or for file merging if a secondary input is specified.

Also, as indicated by the diagram, data may be transferred between the auxiliary and command buffers. The '/' command moves the previous command string from the command buffer to the auxiliary buffer, where it may be altered or saved for subsequent processing. Conversely, the '↑' command loads the command buffer from the auxiliary buffer and initiates execution of the string. As a result useful VW command strings may be stored externally and read as the secondary file providing a library of editing macros.

V. VW Examples

Only single action commands are provided as primitives in VW. Therefore, complex commands must be formed from them via a building block process. As a result, certain tasks for which it might be desirable to have a single command have to be programmed. However, since there is an unlimited number of function that one may wish to perform, this building process permits variety and versatility.

The following are a set of problems which can be solved easily with VW. The more complex of which are solvable in existing editors only with great difficulty.

Problem 1. <u>Character Substitution</u>

A most frequent editing task is to replace one character sequence by another. The following <u>VW</u> macro substitutes all occurrences of the word TECO by VW for the remainder of the file.

<u>*(L'TECO'S'VW')</u>

This command string consists of one loop; the instruction <u>*(</u>, starts the loop. The command, <u>L'TECO'</u> finds the next occurrence of the word TECO. The following command, <u>S'VW'</u>, deletes the found word and inserts VW. The final parenthesis, <u>)</u>, completes the loop and returns control to the beginning for repeated processing. The command stops with the cursor at the end-of-file and with the value TRUE.

Problem 2. <u>Character Substitution at a Specific Column</u>

Often the text to be replaced is differentiated from a similar string of character solely by its position on the line. The following macro replaces TECO by VW only when found at column 5 of any line.

<u>*(*(4>V'TECO'4DI'VW'M)M)</u>

This macro consists of two loops. The instruction, <u>*(</u>, starts the first loop. The first instruction of this loop, <u>*(</u>, initiates a second loop. The command, <u>4></u>, moves the cursor 4 characters to the right, pointing to column 5. The next command <u>V'TECO'</u> checks for the word TECO. If found, <u>4DI'VW'</u> deletes it and inserts VW. The command, <u>M</u>, moves the cursor to the next line and <u>)</u> causes the repetition of the inner loop.

If TECO was not found, the inner loop is terminated and control is transfer to the next command of the first loop, M. The cursor is set to the next line and the) repeats the entire command string. As in the first example, the cursor is left at the end-of-file and the value TRUE is returned after execution.

Problem 3. Truncating long lines

Many programming languages, like Fortran, are restricted to a fixed number of characters per instruction line and, as a result, the programmer is sometimes required to insert continuation characters. This would normally require counting characters in advance but VW enables one to type in his program at random and edit it afterwards using the following macro:

((73><I'←↓ΔΔΔΔΔ -'-M)M)

This instruction consists of two loops. The initial instruction, *(, starts a loop. The first command of the loop which is also *(starts the nested loop. The commands of the inside loop, 73><I'←↓ΔΔΔΔΔ-'-M, will attempt to move the cursor 73 characters on the present line, 73>.

If this succeeds there are at least 73 characters on the line and the next command may proceed. The next command moves the cursor back one space, <; inserts a newline character, 5 blanks and a hyphen, I'←↓ΔΔΔΔΔ-'; which has the effect of limiting the previous line to exactly 72 characters and starting the new line with a continuation character (-) in column 6 followed by the remainder of the original line. The last command of the inner loop, -M, brings us to the beginning of the newly formed line and the closed parenthesis

returns control to the beginning of the loop for continued processing. If the command to move the cursor 73 characters fails, then the loop is terminated and control is passed to the next command after the loop, M. Since, at this time, we know that the current line has 72 or less characters, we simply want to go to the next line and the final closed parenthesis returns control to the beginning of the loop for continued processing. It should be noted at this point that both asterisks are repetition factors, the second allowing the inner instruction to continue through an entire line and the first allowing this procedure to continue through the entire file. All processing ends at end-of-file.

Problem 4. Selective text replacement

Often, we want to do a string substitution over the entire file but within restricted areas. For example, amending last year's Fortran report program requires replacing the last two digits of the previous year with the two digits for the current year. However, since there are other places where such a combination of two digits could exist, we want to limit our changes to the format statements. The following is such an instruction:

```
*(L'FORMAT'*(*(F'75'S'76')M5 >$V'Δ'))
```

This instruction consists of three loops. The oustide loop *(L'FORMAT' finds the next occurrence of the keyword "FORMAT". The second loop *(immediately starts a nested loop. The nested loop, *(F'75'S'76', finds all occurrences of 75 on the current line and replaces them with 76. When this is completed, we return to the second loop M5 >$V'Δ') which moves us to the sixth character of the next line ready to check to see whether or not there is a line continuation character. If it is, we repeat the second loop until there is no line continuation character in position six of the following line and when this situation arises, we repeat the first loop.

Problem 5. Inverting the order of subscripts in a two-dimensional array

Sometimes it is necessary after having defined your subscripts as column, row, to change the order to row, column. The following instruction will allow you to do this:

```
*(L'ARRAY('PF','-D-UC-UKF')'<I','-:-C-K:-P)
```

This instruction consists of only one loop and is a straight right to left instruction. Initially, we look for the name of the array followed by an open parenthesis and save the current position of the pointer which is situated following the parenthesis, L'ARRAY('P. Next, find the comma separating the two subscripts and delete it F','-D. We save the text from our present position back to the saved pointer (the first subscript) by copying it into the auxiliary buffer and then delete it from the main buffer -UC-UK. We search for the closed parenthesis and insert a comma preceding it F')'<I','. We have just made the second subscript the first subscript and the text of the original first subscript remains in our auxiliary buffer. We take the text from the auxiliary buffer and place it between the comma and the closed parenthesis making it the second subscript, -:-C. In order to complete the instruction, we kill the text in the auxiliary buffer and release the saved pointer, -K:-P.

In the above example, we imposed the restrictions that the array named must be immediately followed by the open parenthesis and that the subscripts contained no parenthesis or commas. These are necessary restrictions in exisiting editors. However, VW allows both restrictions to be removed with a little additional coding. In fact the subscript may contain the name of the original array and may itself be subscripted.

Complex text can be processed with the following command to invert the subscripts:

(L'ARRAY('P(*($V','$V'('>)V'('BM)-UC-UKD<BM>I','-:-C-K:>-BM).

Substituting L'ARRAY(' by *(L'ARRAY'*(V'Δ'>)$V'(')?> will allow for a floating open parenthesis.

VI. Conclusion

VW has proven itself a powerful editing tool. The use of simple commands, coupled with a macro capability, provide extensive text manipulation capabilities. Features, such as the auxiliary window, the secondary input file and the balance scope operator are the additional devices suitable for complex problem solving. Its internal data structure and macro implementation keep core requirements low, allowing its use on the mini-computers. For larger machines, VW can readily be made re-entrant as the core windows and cursors are already localized in the one data structure.

The ease with which the FORTRAN versions of VW have been adapted to a variety of machines brings us one step closer to a universal editor for the interactive computer user. At Indiana University the parallel implementation of VW on two different systems, shared by a common group of users, has provided preliminary proof that a universal editor does, in fact, encourage programmer awareness of the more fundamental aspects of computer facilities, resulting in more effective allocation of available computer resources.

REFERENCES

1. Digital Equipment Corporation. Decsystem 10 Lined Software Note Books, Maynard, Mass. (1972).

2. Digital Equipment Corporation. Decsystem 10 TECO Programmers Reference Manual, Maynard, Mass. (1972).

3. Digital Equipment Corporation. Decsystem 10 SOS Software Note Books, Maynard, Mass. (1975).

4. Van Dam, Andries, and Rice, David E. On-line text editing: a survey. ACM COMP. SURVEYS 3, 3 (September, 1971), 93-114.

5. Deutsch, P. L., and Lampen, Butler, W. An on-line editor. Comm. ACM 10, 12 (December, 1967), 793-799, 803.

6. Benjamin, Arthur J. An extensible editor for a small machine with disk storage. Comm. ACM 15, 8 (August, 1972), 742-747.

7. Dewar, Hamish. Compatible context editor. Computer Science Department, University of Edinburgh, Scotland (1972).

8. Control Data Corporation. Control Data Kronos 2.1 Test Editor Reference Manual, St. Paul, Minn. (1973).

9. Wilkes, M. A. Scroll editing: an on-line algorithm for manipulating long character strings. IEEE Transactions on Computers C-19, 11 (November, 1970), 1009-1015.

10. Knuth, D. E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd Ed., Addison-Wesley, Reading, Mass. (1973). Chapter 2.2.3.

11. Knuth, D. E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd Ed., Addison-Wesley, Reading, Mass. (1973). Chapter 2.2.5.

12. Knuth, D. E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd Ed., Addison-Wesley, Reading, Mass. (1973). Problem 2.2.4-18.

13. Wise, D. S. Referencing lists by an edge. Comm. ACM, (to appear).

14. Digital Equipment Corporation. Decsystem 10 TECO Programmers Reference Manual, Maynard, Mass. (1972). Chapter 3.9.2.

15. Morris, James H. and Pratt, Vaughn. A Linear Pattern Matching Algorityn. Report No. 40, Computing Center, University of California at Berkeley (1970).

16. Naur, P. Revised report on the algorithmic language ALGOL 60. Comm. ACM 6, 1 (January 1963), 1-17.

APPENDIX A
VW SEMANTIC MODEL

The follow LISP 1.6 program is presented as the Semantic Model for the VW editing system. User input as described by CMDSTR in chapter IV is preprocessed into a list of the form:

(CMD1 CMD2 CMD3 ...)

Each element, (CMDn) is a sub-list of the following form:

(DOL AA DIR REP SCOPE DOT CMDL STR)

WHERE:

DOL	is T for '$'	CMDARG, NIL otherwise.
AA	is T for 'A'	CMDARG, NIL otherwise.
DIR	is T for '-'	CMDARG, NIL otherwise.
REP	is numeric;	if input was '*' REP has been set to the maximum integer and AA has been set to T.
SCOPE	is numeric;	if input was '*','B','U' SCOPE has been set to the maximum integer or left as 'B' or 'U'.
CMDL	is the command	letter (I,D,W,E,etc.) or a MACRO which is a CMDn. In this case, the preprocessor has inserted the '?' prior to the closing parenthsis.
STR	is a list of	characters to be inserted, located, etc. It is NIL if empty or unnecessary.

PROCESSCMDSTR is the calling function which processes the above.

The LISP code reflects the semantics of the control structure. Only two commands, D and I, are actually presented below, but all commands are as easily implemented in terms of these and the commands E, F, P, V, W, X, and >. When information is Written to the console in the negative direction (-W or -T) the order of the text is reversed in the output buffer so that it is displayed left-to-right as plaintext.

```
(SETQ MAINL NIL)
(SETQ MAINR NIL)
(SETQ  AUXL NIL)
(SETQ  AUXR NIL)
(SETQ OPENL @MAINL)
(SETQ OPENR @MAINR)
(SETQ SHUTL @AUXL)
(SETQ SHUTR @AUXR)

(DE PROCESSCMDSTR (CMDSTR) (COND
        ((NULL CMDSTR) T)
        ((APPLY @PROCESS (CAR CMDSTR)) (PROCESSCMDSTR (CDR CMDSTR)))
        (T NIL)
))

(DE PROCESS (DOL AA DIR REP SCOPE DOT CMDL STR) (COND
        ((ZEROP REP) (NOT DOL))
        ((NOT (ATOM CMDL)) (COND
                ((PROCESSCMDSTR (FIXDIR DIR CMDL)) (PROCESS DOL AA DIR
                        (SUB1 REP) SCOPE DOT (FIXDIR DIR CMDL) STR))
                (T (TRUTHVAL DOL AA))
        ))
        (T (COND
                ((EXECUTE DOL AA DIR REP SCOPE DOT CMDL STR)
                        (PROCESS DOL AA DIR (SUB1 REP) SCOPE
                                        DOT CMDL STR))
                (T (TRUTHVAL DOL AA))
        ))
))
(DE TRUTHVAL (DOL AA) (COND
        (AA (NOT DOL))
        (T DOL)
))

(DE FIXDIR (DIR CMDSTR) (COND
        (DIR (MAPLIST @REVDIR CMDSTR))
        (T CMDSTR)
))
(DE REVDIR (CMDN) (APPEND
        (LIST (CAAR CMDN) (CADAR CMDN) (NOT (CADDAR CMDN)))
        (CDDDAR CMDN)
))

(DE EXECUTE (DOL AA DIR REP SCOPE DOT CMDL STR)
        (CMDL DIR REP SCOPE DOT STR) )

(DE D (DIR REP SCOPE DOT STR) (COND
        (DIR (SET OPENL (CDR (EVAL OPENL))))
        (T (SET OPENR (CDR (EVAL OPENR))))
))
```

```
(DE I (DIR REP SCOPE DOT STR) (COND
      (DIR (SET OPENR (APPEND STR (EVAL OPENR))))
      (T (SET OPENL (APPEND (REVERSE STR) (EVAL OPENL))))
))

(DE : (DIR REP SCOPE DOT STR) (COND
      (DIR (PROG2 (SETQ OPENL @AUXL) (SETQ OPENR @AUXR)
                  (SETQ SHUTL @MAINL) (SETQ SHUTR @MAINR) T))
      (T (PROG2 (SETQ OPENL @MAINL) (SETQ OPENR @MAINR)
                  (SETQ SHUTL @AUXL) (SETQ SHUTR @AUXR) T))
))
```