The 77 Editor

The 77-Editor

Bob Boyer

J Moore

Julian Davies

Department of Computational Logic Memo 62

School of Artificial Intelligence

Edinburgh

February 1973

We suggest you insert this document in your USER'S MANUAL.

This document is published also as a School of Artificial Intelligence, Category 2 publication. This is a provisional description for software material made available for general information but is not an official component of the School Console Manual and is not maintained by the School Systems Programmers.

INTRODUCTION

This editor is actually a totally integrated part of the multi-access system. The primary features of the editor are:

(1) Text is held in an expandable buffer which may be scanned and modified repeatedly in either direction.

(2) When "in" the editor, one is actually in a very special POPVAL which simulates the top-level of the POP-2 operating system. The only differences between these two states is that in the editor errors do not cause SETPOP to occur, and certain identifiers are interpreted as edit commands. Arbitrary pieces of POP-2 text may be typed and executed.

(3) The edit commands are POP-2 operations which take string, word, numeric, or functional arguments. Thus, it is possible to freely mix POP-2 and edit commands (there is no distinction) to define new commands or perform repetitive sequences of commands.

(4) Text may be compiled directly from the buffer. This permits testing and debugging of functions without recompiling the whole file from disc. Syntax mistakes are easier to find because the editor "points" to the last character read by the compiler if it chokes.

(5) It is possible to "undo" the effects of the last n edit commands which inserted or deleted text in the buffer. Thus it is easy to recover from mistyped or

misguided modifications. The integer n is set by the user.

(6) Facilities are provided for easily finding function definitions and for manipulating them as units of text. This holds for other pieces of POP-2 syntax that represent balanced structures: LAMBDA, IF-THEN-ELSE, open brackets, etc.

EDIT ENVIRONMENT

In order to keep edit commands as short as possible and not conflict with operators and functions defined by the user, all edit identifiers are prefixed with the letters "ED". To save the user the trouble of typing ED in front of each command however, a special mode is provided which appears to be the top-level of the operating system, but which automatically prefixes any identifier which corresponds to an edit command by ED. To enter this mode, type ED.

Once "in" the edit environment, life is just as it was outside except that:

(1) Edit commands are available in their non-prefixed form.

(2) SETPOP is avoided to prevent unwanted exiting from the edit interpreter. ERRORs cause both the user's and the auxiliary stack to be cleared, and all executing functions except the editor are aborted.

To exit the edit environment, type CTRL G, or SETPOP();, or CLEARPOP(); (not recommended operating procedure) or type GOON at the top-level.

EDIT ENVIRONMENT-2

The editor may be entered and exited freely without changing the buffer which contains the text. So that if in the middle of an edit you wish to return to the real top-level (this is sometimes desired, even though you can execute arbitrary POP-2 text in edit mode, e.g., you might wish to leave the edit mode to call a function which you have defined which has the same name as an unprefixed edit command) you may type CTRL G to do so, and resume your editing where you left off by typing ED later. Also, accidental CTRL G's don't hurt anything, just type ED again.

Unlike POPEDIT, when in the editor, one is not restricted to typing edit commands. If you wish to output a file, change tracks, compile a function (either as a command you are going to be using repeatedly or to test a new component of your own program), experiment with acceptable POP-2 syntax before typing it into your file, inspect the contents of other files (or edit them) or run your program, you are free to type the appropriate text and have it executed. However, if you type V; (which outside the editor might just be an identifier with a numeric value, and would mean,"put the value of "V" on the stack") the editor types out the current line; if you type I'FOO'; the text 'FOO' is inserted into the buffer; and if you type SEB FOO; the editor searches backwards for the end of the function definition of FOO (if FOO is a function).

The prefixed versions of edit commands are available outside edit mode. That is, once the editor is compiled, you can use the operations EDV, EDI, and EDSEB just as you would use V, I, and SEB in edit mode.

THE BUFFER AND POSITION POINTER

When you wish to edit a file you type IT filename;, where filename is the name of the file to be edited. This inserts the characters into "the buffer", which is conceptually an elastic character strip (but actually a structure composed of POP-2 records which refer to disc sectors or user-typed character strips to be inserted). You may then freely modify the contents of the buffer, and when you are satisfied with it, write it back to your disc track.

There is a "pointer", printed as "↑", which marks the current position in the buffer. It is at this point that insertions or deletions occur. However, you arefree to move the pointer at will throughout the buffer. Commands that move the pointer include the search commands (which position the pointer in front or in back of specified units of text in the buffer) and explicit move commands (like, F 3; which means, go forward 3 characters, or L(-4); which means go 4 lines back). You are also able to discover how many characters from the top of the buffer you are, so as to "remember" a position and return to it later. (Just call H.) (more than above (this page) ...

You can print text from anywhere in the buffer (or even go to other places in the buffer (or even other files) and pick up text and move it). Since you are free to move backwards as well as forwards you can inspect previously edited portions of the buffer and re-edit them. Nothing is written out until you give an output command.

Because the buffer actually refers to the sectors on the disc from which the file came, it is important to avoid DTIDY while editing. To facilitate tidying a track (for instance, to make enough room to output the buffer to it) the command EDTIDY (just TIDY in edit mode) is provided. It is described in detail later.

SYNTAX OF EDIT COMMANDS

Since edit commands are POP-2 operators, their syntax is incredibly free-form. We wish to encourage you to adopt whatever style is most convenient for you.

Since they are operators it is not necessary to type parentheses or dots to cause them to be evaluated. However, you can always treat them as functions if you wish. (That is, typing V(); or .V; is exactly the same as typing V;.)

The following are all equivalent ways to exchange the next occurence of the word "FOO" for the string 'BAR':

```
"FOO" X 'BAR';
"FOO"X'BAR';
"FOO",'BAR'X;
X("FOO",'BAR');
"FOO",'BAR',X;
"FOO",'BAR',X();
"FOO",'BAR'.X;
APPLY("FOO",'BAR',NONOP X);
etc.
```

Just be sure the arguments get on the stack before the command is executed.

We have discovered that the simplest sane syntax is to type the commands in the order they are to be evaluated, with no dots or parens, with all arguments and commands separated by commas. Terminate the sequence of commands with ";" or "=>" as usual in POP-2.

To move to the next line, you can thus type:

```
1L;
L 1;
1,L;
L(1);
1.L;
```

(to exhibit a few of the combinations)

If you realize you are typing the same sequence of commands over and over again, define a function which executes them and use it instead.

For example, assume that you wish to exchange some of the SUBSCR's in a file to SUBSCRC's, but you don't want to write the fully automatic edit function for deciding which occurences. So you want a command that when called will exchange the next 'SUBSCR' for 'SUBSCRC' and then print the corrected line:

```
FUNCTION FOO;
X('SUBSCR','SUBSCRC');
V();
END;
```

If you type the above definition in edit mode, you will thereafter be able to type .FOO; to cause the exchange and verification.

Experiment freely with the syntax until you get comfortable. Use V to verify the changes until you trust the editor and your model of it. Remember, all you have to do to undo the last (possibly disastrous) modification (insert or delete) is type UNDO;.

ARGUMENT TYPES FOR EDIT COMMANDS

Edit commands take a variety of arguments. The type of the argument sometimes affects the precise effect of the command. These relationships are described in detail for each command. Our conventions in this document for specifying the type of an argument are:

(1) character strings -- item typed in using string quotes or constructed with INITC.

(2) text items -- items which are POP-2 words, integers, or reals.

(3) integers -- POP-2 integers.

(4) file names -- POP-2 list structures.

(5) character repeaters or consumers -- any POP-2 function which produces or gobbles one character per call.

(6) function objects -- any POP-2 function or a pair the front of which contains a word and the back the word UNDEF, e.g., [FOO . UNDEF] .

The following general guidelines are offered to clarify the distinctions between certain types.

When used as objects to be searched for or inserted, text items are treated more sensitively than character strings. For example, searching for the string 'X1' would succeed in finding the substring 'X1' in the string 'COORX1Y1'. Searching for the word "X1" would succeed only if the characters found were delimited in such a way to distinguish the occurence as a separate POP-2 word, e.g., as in VARS X1 Y1; or SQRT(X1+3).

Similarly, searching for 123 would find only occurences of that integer (even if you said search for 2:1111011), while searching for '123' could find X123 or 5123.46.

When searching for function objects one finds units of text which define functions, operations, or macros (depending on the IDENTPROPS of the function's print name in FNPROPS). Searching for the end of a function object finds the matching END of the appropriate definition. Thus, if you have a function FOO and want to find its definition, search for FOO (not 'FOO' or "FOO") and you will find the text 'FUNCTION FOO . . .'.

LOCATION SPECIFIERS

Several edit commands take a pair of arguments, LOC1 and LOC2, called location specifiers. These arguments may be of various types and delimit a window in the buffer.

The location of the start of the window is:

(1) If LOC1 is an integer, then position LOC1 in the buffer.

(2) If LOC1 is a string or POP-2 word, then the beginning of the next occurence of the string or word in the buffer (unless the string or word cannot be found in the forward direction, in which case, the beginning of the last occurence of LOC1).

(3) If LOC1 is a function object, then the beginning of the definition of LOC1.

The location of the end of the window is:

(1) If LOC2 is an integer, then position LOC2 in the buffer.

(2) If LOC2 is a string or word, then the end of the first occurrence of the string or word after the location determined by LOC1.

(3) If LOC2 is a function object, then the end of the definition.

(4) If LOC2 is ME or EDME, then the location of the matching end corresponding to LOC1.

The edit commands which take location specifiers are: DC, C, GRAB, MKS, O, and VC.

For example, to compile the function FOO in the buffer, type C(FOO,ME);(or C(FOO,FOO);). To type out the definition of FOO, type VC(FOO,ME);. To GRAB the definition (delete it from the buffer in preparation for moving it somewhere), type GRAB(FOO,ME); (As explained in SYNTAX OF EDIT COMMANDS, the examples above exhibit an arbitrary choice of POP-2 syntax to call the functions involved on the two arguments.)

DESCRIPTIONS OF EDIT COMMANDS

The edit commands are described in detail below. Commands are grouped into classes, and classes are ordered alphabetically. The section SUMMARY OF EDIT COMMANDS, at the end of this document, lists all commands and their classes.

Each command is described in three parts. The first line gives the (unprefixed) identifier for the command followed by the essence of the command. The next line exhibits the number and type of the arguments permitted. When there are options, all are listed. Following this is a detailed description of the command.

To give the arguments we exhibit a call of the command (using parentheses and commas for clarity only). We use the following words to denote arguments of fixed types:

N, M -- integers

FILENAME -- an EASYFILE file name (POP-2 list structure)

CHARREP -- a character repeater

LOC1, LOC2 -- location specifiers

SEARCH-ARG -- a character strip, word, integer, real number, or function object

INSERT-ARG -- a character strip, word, integer, real number, character repeater, or GRABbed object

Thus, we exhibit:

X(SEARCH-ARG, INSERT-ARG)

to mean that the command X takes two arguments, the first of which is of one of the types listed under SEARCH-ARG, and the second of the types under INSERT-ARG. You may of course call X with any syntax you prefer, e.g., 'ABC' X 12.3;.

N.B. We print zero as ∅ in this document.

C Compile

C(Ø) or C(FILENAME) or C(LOC1, LOC2)

C compiles POP-2 text from the buffer. C(Ø) compiles the function, operation, or macro definition the pointer is in. If the pointer is not in a definition, then the largest balanced piece of text containing the pointer is compiled. C(FILENAME) saves the current buffer, inserts file FILENAME, and compiles it. C(LOC1, LOC2) compiles the characters in the window.

If compilation is successful, the pointer is not moved. If a file was compiled successfully, the old buffer is restored along with the old position in it.

If the compilation was not successful, the pointer is immediately behind the last character read by the compiler. If a file was being compiled the old buffer (and NAME) is lost; the current buffer contains the file being compiled; NAME contains FILENAME. The position of the pointer is extremely helpful. In particular, when you are debugging a function definition for syntax errors, you will generally find the pointer only a few characters past the location of the bug that the compiler choked on.

C compiles outside the editor (although you do not leave edit mode). That is, non-prefixed edit commands in the text are not interpreted as edit commands during the compilation.

C makes debugging very fast. Typically one compiles a file the first time with C. When the compilation chokes, use V, or -2 VL 2; to see where the syntax error is. You are still in the editor, so fixing the error is easy. Then type C(FOO,ZZ) where FOO is the function containing the bug. Compilation will continue. When the whole file compiles, you might SAVE it (to keep a reasonably good copy on disc), but keep it in the buffer for further editing. Then run the functions to find bugs. When you find a bug in function FOO, enter the editor if you left it, type S FOO; to get to the definition, and maybe FOO VC ME, to type it out. After fixing it, type C Ø; to recompile the edited function. When you are satisfied with the file, type FILE; to write it to your disc track. With the function definitions at your fingertips and the ability to find and manipulate POP-2 text and words (like all occurrences of a misspelled word, or function calls of a certain function) you will find it easier to both find and correct bugs. In addition, you will not dread modifying a function definition simply to help debug it because you need never write the redefinition back to your disc to compile it.

EDITFROM compile EDIT commands FROM charrep

EDITFROM(CHARREP)

EDITFROM is a function, not an operator. It compiles the characters delivered by the repeater in edit mode. Thus, non-prefixed edit commands are prefixed by ED. To compile a file of your own standard edit commands, execute EDITFROM(DIN(FILENAME)); When you type ED to enter the editor, you really just execute EDITFROM(CHARIN);. If you wish to include edit commands in a file to be compiled with the standard DCOMP, or COMPILE, then you should prefix the commands with ED yourself.

EDSETPOP EDit SETPOP

EDSETPOP()

EDSETPOP is a function, not an operator. It behaves just like SETPOP does, except that the editor is not exited. It aborts all executing functions, clears both stacks, and returns control to the top-level (of the editor). You can use it to get into the editor from outside if you want to. More commonly, it is used within user defined edit functions to abort if the function failed and subsequent commands are to be ignored.

D Delete characters

D(N)

If Nis negative, the -N characters to the left of the pointer are deleted. If N is positive, the N characters to the right of the pointer are deleted. N = Ø is a nonop. The pointer is left unchanged. All delete commands reset NAME to UNDEF if the buffer is cleared by the command.

DC Delete Characters in window

DC(LOC1, LOC2)

The characters in the window are deleted. The pointer is left immediately after the deleted text.

DL Delete Lines

DL(N)

If N is positive all the text from the current position up to and including the Nth newline character in the forward direction is deleted. If N is not positive all the text strictly between the current position and the (N-1)th newline character in the backward direction is deleted.

DS Delete through Search-arg

DS(SEARCH-ARG) or DS(SEARCH-ARG,N)

If not present, N is assumed to be 1. All of the text between the current pointer and the end of the Nth occurence of SEARCH-ARG in the forward direction is deleted. If successful, the pointer is left at the point where the deletion stopped. If SEARCH-ARG is not found N times, "SFL" is printed and no deletion occurs; the pointer is left unchanged and EDSETPOP is called.

DSB Delete through Search-arg Backwards

same as for DS

Same as DS except that the text between the current pointer and the beginning of the Nth occurence of SEARCH-ARG in the backward direction is deleted.

DAZ Delete from A to Z

DAZ()

The entire buffer is cleared. NAME is reset to UNDEF (see IT and FILE commands).

X eXchange

X(SEARCH-ARG, INSERT-ARG) or X(SEARCH-ARG, N, INSERT-ARG)

If N is not present it is assumed to be 1. The Nth occurence of SEARCH-ARG in the forward direction is replaced by INSERT-ARG. If the search is successful, the pointer is left immediately behind the inserted text. If SEARCH-ARG is not found N times, "SFL" is printed, EDSETPOP is called, and the pointer is not moved.

XT eXchange and Test

same as for X

Same as X except that TRUE is returned if the exchange is made and FALSE if it is not (due to failure to find SEARCH-ARG).

XB eXchange Backwards

same as for X

Same as X except that the Nth occurence of SEARCH-ARG in the backward direction is replaced by INSERT-ARG.

XBT eXchange Backwards and Test

same as for X

Same as XB except that a truth value is returned.

GRAB GRAB and delete

GRAB(LOC1, LOC2)

The idea of GRAB is to delete a chunk of the buffer with the intention of reinserting it elsewhere. The chunk deleted is that portion contained in the window specified. An object representing that portion is then left on the stack. This object can be inserted (with I) once (and only once). The pointer is left immediately behind the deleted text.

GRAB does not cause a significant amount of consing to be done (it merely returns a pointer to the chain of records representing the deleted text in the buffer). It is thus an efficient way to move large blocks of text. The following commands GRAB the text defining the function FOO in the buffer and move it to the top of the file:

GRAB(FOO,ME) → X1, A, I X1;

GRAB can also be used to clear the buffer but save the structure, thus allowing another file or grabbed object to be inserted and edited. The orginal buffer can later be reinserted. Thus, GRAB can be used to save several buffers at once and edit them in turn, possibly inserting some into others. (Note: I is a nonop if its argument is a grabbed object that has already been reinserted.)

GRABbed objects are cleaned up when TIDY is called, even though they are not in the buffer. The list EDGRABLIST contains all objects grabbed but not yet reinserted. You may set this list to NIL to let the garbage collector reclaim the space of unwanted grabbed objects.

MKS MaKe Strip

MKS(LOC1, LOC2)

Constructs a character strip containing the characters in the window. The constructed strip is left on the stack. The pointer is left unchanged and the text in the window is not deleted. MKS is used to save a piece of text to be searched for or reinserted several times.

CC Current Character

CC()

Returns the integer representation of the character to the right of the current position. The pointer is left unchanged.

NI Next Item

NI()

Returns the item starting with the character to the right of the current position. The pointer is left unchanged.

IT insert file to be edITed and filed

IT(FILENAME)

If the buffer is not empty, the message 'BUFFER NOT EMPTY' is printed and EDSETPOP is called. (You should either write the buffer out or do a DAZ to kill it.) If the buffer is empty the file FILENAME is inserted in the buffer and the pointer is left at the top. The name FILENAME is stored in NAME for future reference. This is the standard way to begin an edit. When you are finished with the edit, you can file it using the command FILE. Unlike POPEDIT, nothing is written to your disc track until you execute the FILE command (or use either 0 or SAVE, which are also output commands).

I Insert

I(INSERT-ARG)

The pointer is left immediately to the right of the insertion. The text inserted depends on the value of INSERT-ARG and its datatype:

(1) If INSERT-ARG is a character string, the characters in the string are inserted.
(2) If INSERT-ARG is a quoted word, the characters representing the word are inserted. Blanks are inserted at either end when required to make the insertion parse as a word.
(3) If INSERT-ARG is a number, the characters in the decimal representation of it are inserted. Blanks are inserted at either end if required to make the insertion parse as a number.
(4) If INSERT-ARG is a list, it is assumed to be a filename. The user's disc tracks are searched and if a file of that name occurs, the entire file is inserted. If such a file is not found, "IFL" is printed and EDSETPOP is called.
(5) If INSERT-ARG is a function, it is assumed to be a character repeater. The character repeater is consumed and the resulting characters are inserted. (R is I(%CHARIN%)).
(6) If INSERT-ARG is a structure of the type returned by GRAB which has not previously been inserted, it is linked into the buffer. This has the effect of inserting the text represented by that structure. INSERT-ARG is removed from EDGRABLIST.

IC Insert Character

IC(N)

The character represented by N is inserted into the text at the current position. The pointer is left immediately behind it. Notice that you can mung yourself with 19IC;. IC is efficient and can be used as a character consumer to print output into the buffer (i.e. NONOP IC -> CUCHAROUT causes output to be inserted).

R Read from the console

R()

The system accepts input from the teletype and inserts it into the buffer at the current position. CONTROL T terminates the reading and normal edit mode is restored. The pointer is left immediately behind the inserted text. R is used for inserting large blocks of text since only about 900 characters can be typed between string quotes. Characters are inserted as they are read, hence CONTROL G terminates but does not abort the R. UNDO will undo not one but all of the characters inserted during the R.

In order to make it easier to repeat a sequence of commands, the following macros are provided in edit mode (only).

test and repeat

Definition:

```
MACRO <<;
MACRESULTS([ ;LAMBDA; LOOPIF ]);
END;
```

count, test and repeat

Definition:

```
MACRO <*;
MACRESULTS([ ;LAMBDA EDN; EDN+1->EDN; LOOPIF (EDN-1->EDN; EDN;) THEN]);
END;
```

close repeat

Definition:

```
MACRO >>;
MACRESULTS([ ;CLOSE END.APPLY; ]);
END;
```

Examples of use:

```
<< "FOO" XT 'BAR' THEN V >>
```

will replace all occurrences of "FOO" by the string 'BAR' from the current position forward. Each modified line is then printed. The pointer is left after the last modification.

```
3 \ DLP80([FOO]) >>
```

will print file [FOO] to the lineprinter 3 times.

A jump to the top

A()

The pointer is reset to the left of the first character in the buffer.

AT top Test

AT()

TRUE is returned if the current position is the top of the buffer, FALSE otherwise.

Z jump to the bottom

Z()

The pointer is reset to the right of the last character in the buffer.

ZT bottom Test

ZT()

TRUE is returned if the current position is the bottom of the buffer, FALSE otherwise. (If the buffer is empty the current position is the top of the buffer and the bottom of the buffer.)

J Jump

J(N)

The pointer is positioned to the right of the Nth character in the buffer. Thus J(Ø) is the same as A(). To find out the number of characters to the left of the pointer use H.

JL Jump Lines

JL(N)

The pointer is positioned at the beginning of the Nth line in the buffer.

F Forward

F(N)

The pointer is moved forward over N characters. (If N is negative, the pointer is moved backwards; N = ∅ is a nonop.)

B Backward

B(N)

The pointer is moved backward over N characters. (If N is negative, the pointer is moved forwards; N = ∅ is a nonop.) This command is useful since unary minus often parses unintuitively in POP-2.

L move Lines

L(N)

If N is positive, the pointer is moved forward over N newlines. If N is not positive, the pointer is moved just to the right of the (N-1)th newline character in the backward direction. (Thus L(∅) repositions the pointer to the beginning of the current line.)

H Here

H()

Returns the number of characters to the left of the current pointer. Thus: H,S'FOO',3D,J; stores the current position on the stack, moves forward to FOO, deletes it, and jumps back to the original position.
(H,SB'Foo',3D,J clean work.)

ZZ very large integer

ZZ()

The value of **ZZ** is a very large integer (as might be returned by going to the bottom of the buffer and typing H()). All edit commands which take integers as character or line positions have the property that a position greater than the number of characters in the buffer is equivalent to the position of the bottom. Thus J ZZ; is equivalent to **Z**; **ZZ** is handy. Wait and see.

M Match

M()

If the item immediately to the right of the pointer is "FUNCTION", "OPERATION", "MACRO", "LAMBDA", "VARS", "COMMENT", "IF", "LOOPIF", "FORALL", "(", "(%", "[", OR "[%" the pointer is moved to the right of the matching closing item (unless it is immediately followed by a ";", in which case the pointer is left immediately after the ";"). If the first item to the right of the pointer is not one of the above, the pointer is moved to the end of the first item. M is thus used to find the end of a function or lambda expression, the matching close or exit for an if, etc. "MFL" is printed and EDSETPOP is called if the matching closing item is not found. Functions that have location arguments use M to determine the second location if LOC2 is ME.

N.B. When we say "forward" in reference to the motion of the position pointer, we mean "to the right and down", i.e. the direction in which one reads English. When we say "in front of" in reference to the location of the position pointer, we mean "at the left end of".

FILE write edited FILE to disc under filename NAME

FILE()

This is the standard way to finish an edit. The contents of the buffer is written to the current track under the filename NAME (initialized by the command IT). If NAME is UNDEF a message to this effect is printed and nothing is done. If there is not enough room on the track, permission to do an EDTIDY is requested. See the command O below. After output the buffer is cleared and NAME is reset to UNDEF.

SAVE SAVE partial edit

SAVE()

The contents of the buffer are written to the current track under the filename in NAME. Then the new file is inserted and the pointer is set to the position it was at before the SAVE command. This command is used to write a partially edited file to disc to protect it from system crashes and then continue editing.

O Output

O(FILENAME) or O(FILENAME, LOC1, LOC2)

If the optional location arguments are not given they are assumed to be Ø and ZZ. The window in the buffer is written to the current track under the filename given as the (first) argument.

If the argument is not a filename, "OFL" is printed, no output occurs, and EDSETPOP is called. If the window (or buffer) is empty, the message 'BUFFER EMPTY' is output, no output occurs, and EDSETPOP is called. (To create an empty file, use DREAD.)

If there is not enough room on the track for the window, the message 'TIDY Y/N:' is output to the console. If you type Y followed by carriage return, an EDTIDY is executed. If you type anything else followed by a carriage return, nothing is output and EDSETPOP is called. If the EDTIDY yields enough room on the track, the window is output. Otherwise, the message 'TIDY Y/N:' is output, etc.

If you wish to output part of the buffer through an arbitrary character consumer, then use the command VC. VC outputs characters through CUCHAROUT.

VC does not output a term.

S Search

S(SEARCH-ARG) or S(SEARCH-ARG,N)

If not present, N is assumed to be 1. If SEARCH-ARG is an integer the second argument must be supplied. S searches in the forward direction for the beginning of the text denoted by SEARCH-ARG. The relationship between the argument type and the text found is as follows:

(1) If SEARCH-ARG is a character string, the Nth occurrence of the string is found.

(2) If SEARCH-ARG is a text item, the Nth occurrence of the item is found (i.e. occurences as substrings of other items are not counted).

(3) If SEARCH-ARG is a function object, the Nth occurrence of the item "FUNCTION" immediately followed by the name of the function object is found. (i.e. if FOO is a function which was defined using "FUNCTION" or has not yet been given a value (and is thus [FOO . UNDEF]), the text 'FUNCTION FOO' is found. This works even if FOO has been SPEC'D but does not work if FOO has been defined using assignment. If SEARCH-ARG is an operator or macro, the appropriate text is found.)

If successful, the pointer is left at the beginning of the matched text. If not, the pointer is left unchanged and "SFL" is printed, and subsequent edit commands on the same line are ignored; EDSETPOP is called.

ST Search and Test

same as for S

Same as S except that TRUE is left on the stack if the search succeeds and FALSE is left on the stack otherwise. "SFL" is not printed; EDSETPOP is not called.

SB Search Backwards

same as for S

Same as S except that the search proceeds in the backward direction from the current position.

SBT Search Backwards and Test

same as for S

Same as for SB except that TRUE is left on the stack if the search succeeds and FALSE is left otherwise (as for ST).

SE Search for End

same as for S

Same as for S except that the pointer is left immediately after the matched text if the search succeeds. If SEARCH-ARG is a function object, the pointer is left immediately after the matching "END".

SET Search for End and Test

same as for S

Same as for SE except truth values are returned.

SEB Search for End Backwards

same as for S

Same as for SE except that the search proceeds in the backward direction.

SEBT Search for End Backwards and Test

same as for S

Same as for SEB except that truth values are left on the stack.

SS Search Search-arg

SS()

The value of this operator is the last object searched for. Thus SE'FOO',SEB SS; will first search for 'FOO' forwards and then backwards. The use of SS is to save having to type the object of a search again when the wrong occurrence of it was found.

N.B. The way we search for a number is to "print" the number into a character strip, to search through the buffer for a string of characters matching the strip, and having found a candidate to check that the candidate really parses as the number searched for. Consequently, searching for the number 9 will not find 2:1001 or 8:11.

TIDY edit dTIDY

TIDY()

TIDY is the edit version of EASYFILE's DTIDY. When using the editor it must be used instead of DTIDY to insure the integrity of the buffer. Since the buffer refers to sectors on a disc track when a file is inserted, it is essential that information in the buffer be updated if the sectors are shifted down to tidy the track. TIDY does this, both to the buffer itself, and all of the GRABbed objects on the EDGRABLIST list. If TIDY is interrupted (for example, by CTRL G) your disc track is okay, but the buffer may be ruined. Using DTIDY is the most effective way to randomly rearrange your file. Typically you executed TIDY just prior to outputing the buffer when it is necessary to make room on your track. The command O will request permission to TIDY if necessary.

EDUNDOINIT **ED**it UNDO ring buffer **INIT**ialization

EDUNDOINIT(N)

EDUNDOINIT is a function, not an operator. It constructs a circular list (ring buffer) with N elements in it, which is used to hold sufficient information to undo the last N modifications. When the editor is compiled, the ring is initialized to size 2. You may reset it with this function. Note that large values of N mean that the system will not garbage collect chunks of the buffer for a long time, causing your store size to increase (since the last N insertions or deletions must be kept in case you UNDO them).

UNDO UNDO the last modification

UNDO()

The last un-UNDOne insertion or deletion is undone. That is, the text is restored to its configuration just prior to the last still effective insertion or deletion (provided the modification was not made by UNDO itself, in which case the previous modification is UNDOne). The pointer is always left to the right of the text inserted or deleted. It is not restored to its position prior to the UNDOne command. Note that EXCHANGE commands require two UNDO commands to undo. The normal use of UNDO is to recover from mistakes soon after they are made (rather than allow a backtracking search through all possible modifications). For example, if you typed DL 4; when you meant L 4;, then UNDO will restore the deleted lines.

V Verify current line

V()

Prints out the entire line containing the current pointer. The pointer is printed as "↑". The poimter is not moved.

VL Verify Lines

VL(M,N)

Prints the N-M lines starting at the beginning of the line M from the current line and ending at the beginning of the line N from the current line. Thus, to print the two lines above the current position, the current line, and the one below it, type VL(-2,2). The pointer is printed as "↑" if encountered. The pointer is not moved.

VC Verify Characters

VC(LOC1, LOC2)

Prints the characters in the window. The current position, if encountered, is printed as "↑" and is left unchanged. CUCHAROUT may be redefined by the user to cause the window to be output to any character consumer. If CUCHAROUT is not CHAROUT, "↑" is not printed (isn't that neat).

VMAC Verify with MACro expansion

VMAC(LOC1, LOC2)

This command is like VC except that as it prints the window it prints the expansions of any macros. It does not modify the contents of the buffer. VMAC uses the compile command C. Thus, the first time you use VMAC, you also bring in C if it is not already in (see the section on CORE REQUIREMENTS).

CORE REQUIREMENTS

When initially compiled, the editor requires about 9 blocks. This includes buffer space for files inserted, but not for other insertions made. The insertion of a character strip costs 6 words plus the size of the strip. The insertion of a file costs (in words) three times the number of sectors in the file.

There are some features of the editor described in this document which are not compiled initially. Instead, they are trapped and compiled the first time they are used. Thereafter they exist in core. These features are those that deal with function objects, items, text matching, and compiling from the buffer with C. Thus, it is possible to avoid these features and not pay for them. (That is, you can search for strings, insert strings, move arbitrarily, and delete or exchange strings with the initially compiled package. However, as soon as you, say, search for an item an additional function is compiled.) When all the facilities described are compiled, the editor requires about 13 blocks. (Note: two functions are compiled each time they are used and then thrown away:EDUNDOINIT, and TIDY.)

To cancel all identifiers associated with the editor, and regain the space, DCOMP the file [CANCEL EDIT] on track 77.

SUMMARY OF EDIT COMMANDS

COMMAND	ESSENCE	CLASS
A	jump to the top	MOVE
AT	top Test	MOVE
B	move Backward	MOVE
C	Compile	COMPILE
CC	Current Character	GRAB
D	Delete characters	DELETE
DC	Delete characters in window	DELETE
DL	Delete Lines	DELETE
DS	Delete through Search-arg	DELETE
DSB	Delete through Search-arg Back.	DELETE
DAZ	kill buffer	DELETE
ED	enter EDitor	EDIT ENVIRON.
EDITFROM	compile EDIT commands FROM	COMPILE
EDUNDOINIT	EDit UNDO ring INIT	UNDO
F	move Forward	MOVE
FILE	output edited FILE	OUTPUT
GRAB	GRAB and delete	GRAB
H	Here	MOVE
I	Insert	INSERT
IC	Insert Character	INSERT
IT	begin new edIT	INSERT
J	Jump	MOVE
JL	Jump Lines	MOVE
L	move Lines	MOVE
M	Match	MOVE
ME	Matching End	LOCATION SPEC.
MKS	MaKe Strip	GRAB
NAME	NAME of file being edited	INSERT
NI	Next Item	GRAB
O	Output	OUTPUT
R	Read characters from console	INSERT
S	Search	SEARCH
ST	Search Test	SEARCH
SB	Search Backward	SEARCH
SBT	Search Backward and Test	SEARCH
SE	Search for End	SEARCH
SET	Search for End and Test	SEARCH
SEB	Search for End Backward	SEARCH
SEBT	Search for End Backward and Test	SEARCH
SS	Search Search-arg	SEARCH
SAVE	SAVE partial edit	OUTPUT
TIDY	edit dTIDY	TIDY
UNDO	UNDO last modification	UNDO
V	Verify current line	VERIFY
VC	Verify characters in window	VERIFY
VL	Verify Lines	VERIFY
X	eXchange	EXCHANGE
XB	eXchange Backward	EXCHANGE
XT	eXchange and Test	EXCHANGE
XBT	eXchange Backward and Test	EXCHANGE
Z	go to bottom	MOVE
ZT	bottom Test	MOVE
ZZ	2097151	MOVE
<<	test and repeat	MACROS
<*	count, test and repeat	MACROS
>>	close repeat	MACROS
VMAC	Verify with MACro expansion	VERIFY

SUMMARY OF EDIT COMMANDS

COMMAND	ESSENCE	CLASS
A	jump to the top	MOVE
AT	top Test	MOVE
B	move Backward	MOVE
C	Compile	COMPILE
CC	Current Character	GRAB
D	Delete characters	DELETE
DC	Delete characters in window	DELETE
DL	Delete Lines	DELETE
DS	Delete through Search-arg	DELETE
DSB	Delete through Search-arg Back.	DELETE
DAZ	kill buffer	DELETE
ED	enter EDitor	EDIT ENVIRON.
EDITFROM	compile EDIT commands FROM	COMPILE
EDUNDOINIT	EDit UNDO ring INIT	UNDO
F	move Forward	MOVE
FILE	output edited FILE	OUTPUT
GRAB	GRAB and delete	GRAB
H	Here	MOVE
I	Insert	INSERT
IC	Insert Character	INSERT
IT	begin new edIT	INSERT
J	Jump	MOVE
JL	Jump Lines	MOVE
L	move Lines	MOVE
M	Match	MOVE
ME	Matching End	LOCATION SPEC.
MKS	MaKe Strip	GRAB
NAME	NAME of file being edited	INSERT
NI	Next Item	GRAB
O	Output	OUTPUT
R	Read characters from console	INSERT
S	Search	SEARCH
ST	Search Test	SEARCH
SB	Search Backward	SEARCH
SBT	Search Backward and Test	SEARCH
SE	Search for End	SEARCH
SET	Search for End and Test	SEARCH
SEB	Search for End Backward	SEARCH
SEBT	Search for End Backward and Test	SEARCH
SS	Search Search-arg	SEARCH
SAVE	SAVE partial edit	OUTPUT
TIDY	edit dTIDY	TIDY
UNDO	UNDO last modification	UNDO
V	Verify current line	VERIFY
VC	Verify characters in window	VERIFY
VL	Verify Lines	VERIFY
X	eXchange	EXCHANGE
XB	eXchange Backward	EXCHANGE
XT	eXchange and Test	EXCHANGE
XBT	eXchange Backward and Test	EXCHANGE
Z	go to bottom	MOVE
ZT	bottom Test	MOVE
ZZ	2097151	MOVE
<<	test and repeat	MACROS
<*	count, test and repeat	MACROS
>>	close repeat	MACROS
VMAC	Verify with MACro expansion	VERIFY

EDIT ERROR MESSAGES

BUFFER EMPTY
Attempt to output an empty buffer or window.

BUFFER NOT EMPTY
Attempt to start a new edit (with IT) before the buffer is cleared of the previous file. Do a DAZ or write it out. Its name is still in NAME if it was inserted with IT.

IFL
Attempt to insert a file not found on any track in DTRS.

MFL
Closing item (for M) not found in buffer.

NAME = UNDEF
Attempt to FILE or SAVE with NAME = UNDEF. You did not begin the edit with IT. Use O to output and name the file.

OFL
Attempt to output file under an illegal EASYFILE file name.

SFL
Search failed. Either you looked in the wrong direction, you searched for too many occurences, or it's not in the buffer. This error often occurs when you are at the bottom and search forward. It can also occur in DS commands, EXCHANGE commands, and commands which take location specifiers as arguments. When a search fails the pointer has not moved. If a DS prints SFL, it failed to find the target and deletes nothing. The last object searched for is in SS. Thus, if you searched for a string in the wrong direction you can give the correct command and use the argument SS to save typing the string again.

TIDY Y/N:
Permission to do a TIDY requested. If you type Y (yes), a TIDY is done. It will mean you cannot UNDO any previous modifications. If the message comes up again, it means there is just not enough room on the track. If you type N (no) the output command is aborted, no harm done.

↑:
This means you are at the bottom of the buffer. The most common mistake is failure to appreciate that after an insert (with I) you are at the end of the inserted text or file.

ACKNOWLEDGEMENTS

The authors would like to thank D. Bobrow and A. Sloman for their suggestions and support. Without their help the editor would not have been as natural and powerful a debugging tool as it is.

TRACK AND FILE

The editor is on track 77. It may be compiled by typing DCOMP([EDIT]);. When compilation is complete, a list of changes and additions (with dates) is output. You may kill this with CTRL O or CTRL G. You are not in the editor until you type ED. To get out of the editor, hit CTRL G. You need not recompile it to enter it again; use ED again.