THE POP-10 SYSTEM EDITOR
The "PED" Editor.

By: D. J. M. Davies
Date: 25 September 1975
Revised: 1 April 1976

Department of Computer Science,
University of Western Ontario,
LondOn, Ontario, Canada.

This editor and manual are based on the '77-Editor' designed and implemented by Bob Boyer and J Moore at Edinburgh, and described in:

'The 77-Editor'
Bob Boyer, J Moore & Julian Davies
D.C.L. Memo 62, School of Artificial Intelligence, Edinburgh, February 1973.

This implementation owes much in detail to the implementation of the editor in the Edinburgh POP-2 system by Robert Rae, Chief System Programmer there.

INTRODUCTION.

The PED Editor is a text editor built into the POP-1Ø system for editing program and data text. Arbitrary ASCII text can be edited, but the editor is expected to be particularly useful during development of programs for POP-1Ø.

The editor comprises two main components: a 'text buffer' to hold the text being edited, and a set of commands for working on the text in the buffer. There is provision for entering text into the buffer either from the terminal or from files on disc or other devices, for manipulating the text, for inspecting part or all of the buffer on the terminal, for writing part or all of the buffer contents to an output file, and also for compiling POP-1Ø program text directly from the buffer. A very flexible set of editing commands is provided, including commands which have some 'knowledge' of POP-1Ø's syntax.

The Text Buffer is conceptually a character strip with flexible bounds; it may be extended almost indefinitely by inserting text at any point. It contains a sequence of 7-bit ASCII characters. There is also a 'Position Pointer' which is positioned somewhere in the buffer. The position pointer is always positioned between two characters, or before the first character, or after the last. It may be moved around quite freely within the text buffer, either by specifying a numerical displacement (e.g. in character positions) or by a search for a specified piece of text within the buffer. All insertions of text take place at the position pointer (strictly speaking, just before it), and all deletions from the buffer take place forwards or backwards from the pointer's position (possibly after moving the pointer first).

In many respects, the editor is similar to the TECO editor. Its main advantage is that it is within the POP-1Ø system, so that a file can be edited and recompiled many times without leaving POP-1Ø, which means that other programs to be used in conjunction with it do not have to be recompiled each time it is edited. This is made particularly easy by the ability to compile program text directly from the text buffer; a whole program could be created and debugged during a session, keeping the text only in the text buffer, and when correct it can finally be written to a file on disk.

The editor is itself an integral component of the POP-1Ø system, and all the editing facilities are provided through standard functions, operations and other variables. Thus it is quite possible to use the facilities of POP-1Ø (in the ordinary sense) with those of the editor, for instance to define an ordinary POP-1Ø function to perform a particular sequence of editing operations. It is possible, in fact, to define any editing commands you like in terms of those provided as standard, and this flexibility can be put to good use in appropriate occasions.

An important feature of this editor, not featured in most 'stand-alone' text editors, is the UNDO command. All operations on the text (which change its contents) amount to sequences of deletions and insertions. After any text has been inserted or deleted in the buffer, it is possible to reverse the insertion or deletion with an UNDO command. It is therefore easy to recover from mistyped or misguided modifications to the text buffer. The system, of course, has to remember information if an editing operation is subsequently to be UNDONE, which uses store, so the system only stores this information, at any time, for the 6 most recent changes to contents of the buffer. (This limit may be changed if the POP-10 system is reassembled.)

Because it is possible to compile text directly from the buffer, it becomes attractive to use the editor as a debugging tool. The program to be debugged can be read into the buffer, and extra code can be inserted into the version in the buffer for debugging purposes. Unless a FILE or SAVE command is executed, the disk file will be unaffected by this. Another feature of compiling code from the buffer is that if the compiler chokes on faulty syntax, etc., the buffer pointer is left pointing where the compiler reached in the text.

The editing commands are summarised in the file PEDS.HLP, and are described in detail below.

The Edit Environment

The Editor commands are mostly standard POP-10 operations of precedence 2, with a few other functions, variables, and operations of precedence 1. To manipulate text in the editor buffer, therefore, normal POP-10 text is typed to the compiler, which uses these operations, etc.

In order to keep edit commands as short as possible and not conflict with operations and functions defined by the user, almost all edit identifiers are prefixed with the letters "PED", and are usually two letters long without this prefix or five characters long with it, e.g. "IT"/"PEDIT", "MC"/"PEDMC", etc. To save the user having to type PED in front of each edit command, however, a special mode is provided in the compiler which compiles POP-10 text quite normally except that any identifier is automatically prefixed with "PED" if that makes it into an edit command. This mode of compiling is known as "Edit Mode".

POP-10 can be made to compile text from the terminal in Edit Mode, at the top level, by calling the function SETEDIT, or by typing ↑F (which calls SETEDIT). SETEDIT is just like SETPOP in exiting from any programs currently being executed, clearing the stacks, and then restarting the compiler reading from the terminal, only it

restarts the compiler in edit mode. An arbitrary input character repeater can be compiled in edit mode by using the function PEDITFROM, which otherwise is just like COMPILE. If an error occurs while the system is in edit mode, then SETEDIT is called after the error instead of SETPOP. Also if the function POPREADY is called, or a "Ready" break occurs while the system is in in edit mode, then the break itself also compiles from the terminal in edit mode. When the system is in edit mode, the standard prompt on the terminal is "`" instead of ":".

Once "in" the editor, that is, in edit mode, life is just the same as before except that edit commands may be given in their unprefixed forms, and errors and breaks will keep the system in edit mode. To leave edit mode, simply type ↑G or SETPOP(); .

You may enter and leave edit mode quite freely without changing the buffer which contains the text. If, in the middle of editing, you wish to leave edit mode, simply type ↑G, and type ↑F later to return. This manoeuvre is sometimes desirable, even though you can execute arbitrary POP-1Ø text in edit mode; you may wish to access a variable whose identifier is the same as an unprefixed edit command. Accidental CTRL G's do not hurt anything; just type ↑F or .SETEDIT; again.

Unlike POPEDIT, when 'in the editor', one is not restricted to typing edit commands. If you wish to output a file, compile a function (either as a command you are going to use repeatedly or to test a new component of your program), experiment with acceptable POP-1Ø 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 VC; the editor will type out the current line. If you type IN'FOO'; the text 'FOO' is inserted into the buffer, If you type SB BAR; the editor searches backwards for the function definition of BAR (if BAR is a function).

The prefixed versions of edit commands are available outside edit mode. That is, you can use the operations PEDVC, PEDIN and PEDSB at any time just as you would use VC, IN and SB in edit mode.

The Buffer and Position Pointer

When you wish to edit a file, you type

```
IT[ filespec ];
```

where [filespec] is the name of the file to be edited. This inserts the characters of the file into "the buffer", which is conceptually an elastic character strip (but actually a structure composed of POP-1Ø records which refer to blocks on disc or to user-typed character strips which have been inserted). You may then freely modify the contents of the buffer, and when you are satisfied with it, write it back to a disc file.

There is a "pointer", printed as "↑", which marks the current position in the buffer. It is here that insertions and deletions occur. However, you are free to move the pointer at will throughout the buffer. This is true however long the file is, and however many pages it contains: you have access to the whole file all the time. Commands that move the pointer include the search commands (which position the pointer before or after a specified unit of text in the buffer) and explicit Move commmands (such as 3MC; which moves the pointer 3 characters forwards, and -4ML; which moves the pointer 4 lines backwards). You are also able to discover how many characters you are from the start of the buffer (with CP - Current Position) so as to "remember" a position and return or refer to it later.

You can print text from anywhere in the buffer, or even go to other places in the buffer 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. The usual output command at the end of editing a file is FILE, which writes the buffer to a file named by the filename in the variable NAME: NAME is set to contain the file's name by the IT command, but it can be assigned to directly.

Errors

When an error occurs, the normal POP-10 error handling routines are called. Normally, this means that SYSERR is called to print a message giving the error number and culprit, and then a POPREADY break is entered. At the start of the break you can type ? to get an explanation of the error, or : to get a listing of the calling sequence.

However, if the error is an editor error, and the variable PEDSHTERR has been set non-zero, then the error message from SYSERR is abbreviated, and no break is called. This is to reduce the amount of tiresome printing from routine editor errors. The message might look like this:

```
?PEDSFL   168
culprit  ABC
setedit
!
```

This is a search-fail error, error 168, but the mnemonic PEDSFL is enough for the experienced user. The mnemonics are given in Appendix A. If you have PEDSHTERR set, type .POPXPLNER; to get an explanation of the most recent error. PEDSHTERR is initially false, to give a full error message and break on every error. Note that PEDSHTERR has this effect for editor errors, irrespective of whether or not you are actually in edit mode.

Syntax of Edit Commands

Since edit commands are operations, their syntax is very free-form. You may adopt whatever style is most convenient for you.

Since they are operators, it is not necessary to include dots or parentheses to get them executed. However, you may give parentheses if you wish (but an operation may not directly follow a period). That is, typing VC(); is exactly the same as typing VC;.

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

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

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

It has been found that the simplest sane syntax is to type the commands in the order they are to be evaluated, with no dots or parentheses, but with all arguments and commands separated by commas. Terminate the sequence of commands with ";" or "=>" as usual in POP-10.

If a numeric argument directly precedes a command, it is not necessary to interpose a space or comma. Thus to move to the next line in the buffer, it is possible to type any of the following.

```
1ML;
ML 1;
1,ML;
ML(1);
etc.
```

(to exhibit a few of the variations)

If you realise that you are typing the same sequence of commands over and over again, define a function which executes them and call it instead. The function may have formal parameters and take arguments if this is desired.

For example, assume you wish to change some of the SUBSCR'S in a file to SUBSCRC'S, but you do not wish to write a fully automatic edit function to decide which occurences. The following command operation will make such an exchange and then print the line:

```
OPERATION 2 FOO;
XF("SUBSCR",'SUBSCRC'); VC;
END;
```

The following command is equivalent but more efficient:

```
OPERATION 2 FOO;
"SUBSCR"SEF; 'C@IN VC;
END;
```

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

The following command strings are other ways to make that exchange throughout the file, verifying every occurence:

```
JA; <?"SUBSCR"XFT 'SUBSCRC@ THEN VC?>
    or
JA; <?"SUBSCR"SEFT THEN 'C@IN VC?>
```

Experiment with the syntax until you get comfortable. Use VC 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;. (It takes two UNDO's to undo an exchange.)

Argument Types for Edit Commands

Edit commands take a variety of arguments depending on the requirements, however, all arguments must be POP-10 items of one sort or another. The commands always conform to general rules of POP-10 syntax. The type of argument sometimes affects the precise action of the command. These relationships will be described for each command. Some commands will supply default arguments if applied when the stack is empty. This facility should be used with caution, however, since stray items on the stack may be gobbled up with unexpected consequences.

The conventions for specifying argument types are as follows.

(1) STRINGS - items typed in using string quotes, or made with INITC.
(2) TEXT-ITEMS - items which are POP-10 words, or (positive) numbers
(3) INTEGERS - POP-10 integers
(4) FILE-NAMES - POP-10 list structures
(5) REPEATER - POP-10 character repeater functions
(6) FUNCTION OBJECTS - either any POP-10 function, operation or macro with a name in its FNPROPS, or any pair with a word in

its front and UNDEF in its back.
(7) GRABBED OBJECTS - items returned by the GW command
(8) An INSERT ITEM - a string, text-item, filename, repeater, or grabbed object: these are permissible arguments of IN and IT.
(9) A SEARCH ITEM - a string, text-item, or function object: these are permissible arguments for search commands.

The idea behind Function Objects is as follows. To find the text defining a function FOO, one can type: FOO SF;. The variable FOO will either contain a function FOO, in which case the name of the function can be obtained by SF from its FNPROPS, or else FOO was previously undefined, in which case the system will initialise it with the pair [FOO . UNDEF], and SF can extract the desired name from this pair. This is discussed further in the section on search commands.

The difference between strings and text-items is that strings represent a particular sequence of characters in the buffer, irrespective of context, but text-item arguments represent a sequence of characters in the buffer in such a context that the itemiser would construct the given text item while reading the text in the buffer. For example, searching for the string 'X1' will find the substring X1 in the string COORX1Y1; searching for the word "X1" would reject that substring as a match, and would only find a string 'X1' if it is properly delimited as the word X1 in the text: e.g. in VARS X1 Y1; or SQRT(X1+3).

Another difference between words and cstrips as search items is in the treatment of lower-case characters. In a cstrip, the upper case letters and the characters "@" "[" "\" "]" "↑" and "←" are treated as equivalent to their lower-case forms; in a search for a word, the characters must match exactly because the search routine reads an item out of the buffer and checks that it equals (=) the search item.

Similarly, searching for the number 123 would find only occurences of that integer, and not the substring in the texts X123 or 1234.Ø. Similar rules apply during insertion of text-items; the characters required are inserted, but a space will be put at either end if necessary. Inserting a string will insert only exactly the characters in the string. When a search is made for a number, the number is printed with PR, and the search function looks for the sequence of characters made by PR (less the initial space). Thus, searching for 123 will not find the text 2:1111Ø11 (but it will find the text 8:123).

Windows in the Buffer

Several commands take a pair of arguments called Window Specifiers, which specify a "Window" in the buffer. A "Window" is simply some part of the text in the buffer (possibly of zero length).

The first window specifier may be either an integer, or else a search item. An integer argument is taken as the CP count of characters to the start of the window; a search item is searched for in both directions in turn to find the window start (SF first, then SB if that failed).

The second window specifier may be an integer, a search item, or the special item PEDME. If an integer, it defines the CP count at the end of the window, and must be >= to the CP count at the start of the window. If it is a search item, it is searched for forwards from the start of the window, and its End is taken as the end of the window (SEF). Otherwise, the item PEDME signifies that the end of the window is at the Matching End of the text starting at the start of the window, as defined by the command MM.

The commands which use windows are: PCOMP, OP, VW, DW, GW and PEDAPPW. These commands, and certain others which use a window internally, will print the warning message '%buffer empty' if applied when the buffer is empty.

As examples of the use of window specifiers, you may compile the function FOO from the buffer by typing

```
PCOMP(FOO,ME);
```

or you can compile the whole buffer by typing

```
PCOMP(Ø,ZZ);
```

You can type out the definition of the function FOO by typing

```
VW(FOO,ME);
```

or write it into a file with

```
OP([filename],FOO,ME);
```

You can grab the text of the function definition with GW(FOO,ME) which deletes the text and returns a 'grabbed object' which can be inserted elsewhere (possibly in another file).

Compilation

PCOMP

Formats:

```
PCOMP E Ø => ()
PCOMP E filename => ()
PCOMP E window-specs => ()
```

PCOMP compiles POP-10 text from the buffer, either a window from the current buffer, or after first filling the buffer from the specified file.

PCOMP(Ø) compiles the function, operation or macro definition which the pointer is curry poi7Iwg too. If the pointer is not currently pointing inside such a definition, PCOMP compiles the last preceding definition and following text until the poifter is reached (or further if necessary to reach the end of an enclosing syntactic structure). If there are no preceding function, operation, or macro definitions in the buffer, then text is compiled from the start of the buffer up to the pointer.
If PCOMP is applied with the stack empty, then Ø is taken as default argument.

PCOMP(FILENAME) saves the current buffer (with GW), inserts the specified file into the buffer, and compiles it from there.

PCOMP(WINDOW-SPECS) compiles the text in the specified window of the buffer.

If compilation is successful in PCOMP, then the buffer contents and pointer position are always restored to what they were beforehand. However, if an error occurs during compiling, then the pointer is positioned just after the character last read by the compiler. If a file was being compiled, then the old buffer position and contents are lost. If the error is due to a syntax error, you will generally find that the pointer is only a few characters after the bug, so VC; or -2V8 2; (for example) will display the context of the error. You are still in the editor (if you were before), so fixing the error is easy: edit the text in the buffer, and then type

PCOMP(FOO,ZZ);

where FOO is the function definition containing the error. This will recompile the changed definition and the rest of the file following it. If another error is discovered, this procedure can be repeated.

PCOMP makes it very easy to debug POP-10 programs in this way. When a file all compiles from the buffer, you might give a SAVE command to write a copy in disk, while keeping it in the buffer. Further bugs may appear when you attempt to run the program, so you can edit any function definition in the buffer and recompile it from there with PCOMP(Ø);. When you are satisfied with the program, you can say FILE; to write it back to disk.

PCOMP compiles text in ordinary mode, even though your terminal is in edit mode. That is, unprefixed edit commands in the buffer are not recognised as edit commands. To compile a window from the buffer in edit mode, you have to say:

PEDAPPW(window-specs,PEDITFROM);

PEDITFROM

Format:
PEDITFROM E <character repeater> => ()
PEDITFROM E <filename> => ()

PEDITFROM is a function, not an operation. It compiles the character stream delivered by the repeater function, in Edit Mode. Thus, non-prefixed edit commands met in the stream are prefixed with PED. Apart from compiling in edit mode, it is just like COMPILE. To compile a file of your own standard edit commands, execute

PEDITFROM([filespec]);

To compile edit commands in a file with COMPILE or in INIT.POP, you must give all edit commands in prefixed form.

The variable POPCREP is local to PEDITFROM (as it is to COMPILE) and contains the repeater function argument; this may be useful after an error to determine where the error occured. PEDITFROM does not change whether your terminal is in Edit Mode.

SETEDIT

Format:
SETEDIT E () => !!

SETEDIT is a function, not an operation. It causes all currently executing functions to be abandoned (as if by a JUMPOUT function), and restarts the compiler, compiling from the terminal in Edit Mode.

SETEDIT clears the stacks and system workspace, and sets POPEDMODE to TRUE. Then it applies the function POPEDFN, whose default value is IDENTFN but which can be changed by the user. If POPEDFN returns (as it usually will), then the compiler is restarted, compiling from the terminal in Edit Mode. The compiler actually reads the input repeater in PEDCHARIN, which defaults to CHARIN. It is possible to replace PEDCHARIN by another repeater function, but obviously it should take its characters from CHARIN, possibly after manipulating them in some way. (This facility can be useful in implementing a 'transcription' facility, which records all proceedings at the terminal in a disc file.)

The prompt is set to "! " by SETEDIT. If a ready break is called, then POPREADY will also compile in Edit Mode, giving the prompt "n!! " to indicate this (where <n> is the break level).

Inserting

IT edIT file

Format:
IT E <filename> => ()

IT is an operation to start editing a file. It is applied to the filename, a list, and the file is inserted into the buffer. This is the standard way to begin an edit.

An error is given if the buffer is not empty (you should either write the buffer out, or clear it with .RESET;). If the buffer is empty, the file is inserted, and its filename is saved in NAME for future reference; the buffer pointer is left at the top of the buffer (which is different from the other insert commands). When you have finished editing, it is usual to write the new file with the commands FILE or SAVE, which write the buffer to a file named as in NAME: you can assign a new filename to NAME if you wish the new file to have a different name.

Technically it is possible to insert any Insert Item with IT. IT is equivalent to IN, except that it first checks that the the buffer is empty, and afterwards puts the pointer at the top of the buffer. However, IT prints a warning '%not a file' if its argument is not a filename.

E.g.:

```
[MYPROG.POP]IT;
```

IN INsert item

Format:
IN E <insert item> => ()

IN inserts the given Insert Item into the buffer just before the position pointer. That is, text is inserted at the pointer, and the pointer is left at the end of the new characters. The text inserted depends on the insert item.

(1) If the insert item is a string, the characters of the string are inserted. (If the string is of zero length, then IN is a no-op, and does not make an UNDO entry either.)
(2) if the argument is a word, the characters of the word are inserted, with spaces before or after as necessary to delimit it properly.
(3) if the argument is a number, then characters of its decimal representation are inserted, with spaces before or after as necessary to delimit it. The characters will be those that PR prints for that number.

(4) if the item is a list, it is assumed to be a filename, and that file is inserted. Furthermore, if the buffer is empty, then the filename is also put into NAME.
(5) If the item is a function, it is assumed to be a character repeater, and all the characters it produces are inserted (with IC).
(6) If the item is a grabbed object (made by GW), then the characters in the object are inserted, provided that the object has not been inserted previously anywhere. If the grabbed object has already been inserted, then an error is given.

If the buffer is empty, then IN always changes the value of NAME. If the argument is a filename, then it is copied to NAME; if the insert item is anything else, then NAME is cleared. IN never changes NAME unless the buffer is empty.

E.g.:
```
'A-3*X'IN VC;
IN "XX", IN "-", IN 34;
POPMESS([INSOS FILE.TXT])IN;
[LIB:RANDOM.LIB]IN;
```

IC Insert Character

Format:
IC E <character code> => ()

The character represented by the code is inserted into the buffer just before the pointer. If IC is applied to TERMIN, it has no effect.

IC is designed to be efficient in its use of store. If it is applied repeatedly in succession, the characters are stored in order with a minimum of overhead. It can therefore be used as an output repeater if desired, e.g.:

```
NONOP IC -> CUCHAROUT;
```

When IC has been applied repeatedly in this manner, one application of UNDO removes all the characters inserted. IN uses IC repeatedly in this way if its insert item is a character repeater.

E.g., 32IC; inserts a space

IR Insert Read

Format:
IR E () => ()

IR is a command which reads characters from the terminal and

inserts them into the buffer at the pointer. It stops reading when either ↑Z or <ESC> is typed. The terminal prompt is set to a null string while it is reading. IR is used to insert large blocks of typed text. Characters are inserted as they are typed (line by line), so ↑G or ↑F will abort the IR but do not lose the text already typed.

One application of UNDO will remove all the characters typed to IR. IR actually reads characters from PEDCHARIN, which is normally CHARIN.

E.g.:

```
IR;
..text...
↑Z
```

Moves

JA, JZ Jump to top, Jump to bottom

Format:

JA, JZ all E () => ()

JA moves the pointer to the top of the buffer, before the first character if any. JZ moves the pointer to the bottom of the buffer, after the last character if any.

AT, ZT top-Test, bottom-Test

Format:

AT, ZT all E () => <truthvalue>

AT and ZT are operations of precedence 1, which return TRUE respectively if the pointer is at the Top ("A") or Bottom ("Z") of the buffer, or otherwise FALSE in each case.

An efficient way to test whether the buffer is empty is with

```
AT AND ZT
```

which is a true conditional only when the buffer is empty.

JC Jump to Character

Format:

JC E <integer> => ()

JC takes an integer argument N, and puts the pointer just after the N'th character in the buffer. It is equivalent to JA if N is zero or negative, or to JZ if N is larger than the number of characters in the bufLs:Z'{plly the argument of JC will be a count obtained from CP at an earlier time, and JC will then restore the pointer to the same position again (provided that no insertions or deletions have subsequently been made in the buffer before that position).

CP Current Position

Format:

```
CP E () => <positive integer>
```

CP is an operation of precedence 1, and it returns the number of characters between the top of the buffer and the pointer. It returns zero if the pointer is at the top.

ZZ

ZZ is a protected variable which contains a very large positive integer. It is commonly used to supply a window specifier item for the end of the buffer, since JC(ZZ); will be equivalent to JZ;.

JL Jump to Line

Format:

```
JL E <line number> => ()
```

JL takes an integer argument N, and puts the pointer at the start of the N'th line. JL(1); is equivalent to JA; as is JL with a zero or negative argument. JL is equivalent to JZ if its argument is larger than the number of lines in the file. Lines are regarded as terminated by line-feed characters, so JL goes to the top of the buffer and searches forwards for N-1 line-feeds. Note that formfeeds, vertical tabs and carriage returns are ignored in counting lines.

MC Move Characters

Format:

```
MC E <integer> => ()
```

MC takes a signed integer and moves the pointer forwards or backwards by that number of characters (relatively). ØMC; is a

no-op. MC moves the pointer forwards if its argument is positive, otherwise backwards. If the argument would place the pointer outside the buffer, then the pointer is left at the top or bottom of the buffer depending on the direction of movement.

ML Move Lines

Format:
ML E <integer> => ()

ML takes a signed integer and moves the pointer forwards or backwards by that number of lines. ØML; positions the pointer at the start of the current line; 1ML; positions it at the start of the next line forwards, etc. Like MC, ML leaves the pointer at the top or bottom of the buffer if the magnitude of the argument is too large.

MM Move to Matching end

Format:
MM E () => ()

If the item immediately following the pointer (in the buffer) is a number or string constant or a word other than those listed below, the pointer is moved to the end of that item, except that if that item is immediately followed by a semicolon, then MM leaves the pointer just after that semicolon.

If the item fllowing the pointer is one of the following 'open bracket' words, then the pointer is moved to just after the corresponding matching 'closing bracket' (unless that is immediately foX!{;+# by a semicolon, when the pointer is moved to just after the semicolon). If the matching closing item cannot be found because of incorrect nesting of brackets of that type, then an error is given and the pointer is not moved.

If there is no item between the pointer and the end of the buffer, then MM leaves the pointer at the bottom of the buffer.

opening bracket	matching closing bracket
CANCEL	;
COMMENT	;
FORALL	CLOSE or EXIT
IF	CLOSE or EXIT
LOOPIF	CLOSE or EXIT
UNLESS	CLOSE or EXIT
VARS	;
LAMBDA	END
FUNCTION	END
OPERATION	END
MACRO	END
SECTION	ENDSECTION
"	"
(	)
(%	%)
[	]
[%	%]

Note: recognition of closing brackets depends on the syntactic properties of the words (borrowing compiler routines); cancelling a closing bracket word will make it unrecognisable. Also MM simply skips two items if it meets '"', even if the second of these is not actually another word quote.

MMT Move to Matching end and Test

Format:
MMT E () => <truthvalue>

MMT has the same effect as MM except that it returns a truth-value. If it can find the correctly nested matching end of text in the buffer, then it moves the pointer there and returns TRUE. If, however, the text in the buffer is incorrectly structured, with a missing closing bracket, then MMT leaves the pointer unmoved and returns FALSE. MMT also returns FALSE if it moves the pointer to the end of the buffer without meeting any items. (In this case, both MM and MMT leave the pointer at the end of the buffer.)

Note that MMT may still give rise to errors in the itemiser if, for example, a bracket decorator appears out of context, or a number contains two decimal points.

Searching

SF Search Forwards (for start of item)
SB Search Backwards (for start of item)
SEF Search for End Forwards
SEB Search for End Backwards

Format:
SF, SB, SEF, SEB all E <search item?>, <count?> => ()

These commands search forwards or backwards for the specified piece of text, starting from the current position of the buffer pointer. If the text is found in the buffer, then the pointer is moved to the start or end of it, as specified. If the text is not found, the pointer is not moved, and a 'search fail' error is called.

If the top item on the stack, on entry, is an integer, it is taken as the count argument. It must be a positive integer. Otherwise, the count argument defaults to 1. The search command will search for the N'th matching piece of text in succession if the count is N.

If a search item is supplied as argument, it must be one of the cases listed below, and it defines the text to be sought. Otherwise, if the stack is empty, the current content of the variable PEDSS is used as the search item. When a search item is supplied, it is saved in SS for future reference. SS may be assigned to directly.

The search item may be one of the following:

(1) a STRING: the buffer is scanned for a matching group of characters, except that upper and lower case alphabetic characters are regarded as equivalent (in their corresponding pairs). All characters in the string are significant.

(2) a WORD: the buffer is scanned for a matching group of characters in the buffer which parses as the given word item. This means that upper and lower case characters are regarded as distinct, unlike the string searching case.

(3) a NUMBER: the number is 'printed' with PR to determine a sequence of characters to search for, and that sequence is sought. Then the characters are read back, taking account of leading zeros, etc., to check that the text in the buffer parses as the same number.
Negative numbers cannot be sought, because the itemiser cannot construct negative numbers for the check. A search for 2 will not find 2.Ø , because these are not equal.

(4) a Function Object:
A Function Object is either
(a) a function item with a name, or
(b) a pair whose front is a word, and whose back is

UNDEF.

In case (a), the name of the function is determined from its FNPROPS, and IDENTPROPS is applied to it to discover whether it is an operation or macro. In case (b), the front of the pair is taken as the name of the function, and it is assumed to be a normal function, not an operation or macro.

In either case, one of the key-words "FUNCTION", "OPERATION" or "MACRO" is sought as appropriate, and when found, the following text items in the buffer are inspected (with CI). The text in the buffer is deemed to match if it conforms to one of the following cases:

```
FUNCTION <name>                    or
MACRO <name>                       or
OPERATION <any item> <name>
```

Note that this procedure still works in case (a) if the function item has been 'bugged', because BUG copies the FNPROPS of the original function to the new closure function that it creates.

In the case of searching for the end of a function definition, the pointer is left at the Matching End of the entire function definition.

A search failure commonly results from searching in the wrong direction, for example: searching forwards when already at the bottom of the buffer. In such cases, it is easy to try again by typing e.g. SB; or 2SEB; etc. as required. The stack is always empty just after an error, so it is safe to implicitly use the value of SS, which will have been set by the previous search command.

SFT	Search Forwards and Test
SBT	Search Backwards and Test
SEFT	Search for End Forwards and Test
SEBT	Search for End Backwards and Test

Format:

```
SFT, SBT, SEFT, SEBT all E <search item?>, <count?>
                              => <truthvalue>
```

These four commands exactly parallel the previous four search commands in the way they search for a piece of text in the buffer. If the search is successful, they position the pointer in the same way as the previous commands, respectively, and also return TRUE. If, on the other hand, the search fails, then they leave the pointer unmoved and return FALSE instead of calling an error.

These versions of the search commands are commonly used in 'automatic' editing programs for determining whether the editing procedure has anything left to do. For example, they are useful as the conditionals of iterative expressions, such as:

<? SFT'string' THEN edit actions ?>

Verification

All verifying commands send output through CUCHAROUT to the current output stream. When this is CHAROUT, sending the characters to the terminal, then the characters sent are slightly modified as described below. Otherwise, if the current output is to any other device, the characters sent are exactly those in the selected portion of the editor buffer, without modification. This enables the contents of the buffer to be written on some device other than disk, by placing an output character repeater function in CUCHAROUT and then using a verification command. However, it will be necessarq to explicitly close that output stream, since the verification commands do not close the output stream when they have finished.

When the current output is to the terminal (through CHAROUT), the character stream is transformed in three respects.

(a) a 'marker' character is printed at the position of the pointer if that is within the text being verified. The 'marker' is initially "↑", but can be changed to any other ASCII code with

POPMESS([PEDMARK <ASCII character code>])

A useful alternative marker is code 1Ø, which is line-feed. No marker is printed if the code is set to zero.

(b) upper or lower case letters and special characters may be flagged if the terminal does not have dual-case hardware. If the terminal is in TTY LC mode, then all characters are sent as they appear in the buffer (except for translation of control characters to arrow format). However, if lower-case mode is turned off, then the treatment of letters and special characters depends on the value of the variable PEDEU. EU is initially zero, and in this case lower-case range characters are each converted to their Upper-case equivalents and prefixed with a single quote. If EU is positive, then lower-case characters are replaced by their Upper case counterparts, but Upper-case characters from the buffer are prefixed with single quotes. If EU is negative, then all alphabetic-range characters are printed unchanged, even though the terminal is not in lower-case mode.

(c) a form-feed or vertical tab character is printed as two line-feeds, to save time and paper.

Verification commands never move the position pointer.

VL Verify Lines

Format:

VL E <integer> => ()

VL takes a signed integer as argument, and verifies that number of lines forwards or backwards from the current position. <n>VL; will print out the same characters as <n>DL; will delete.

VB Verify Block of lines

Format:
VB E n1, n2 => ()

VB takes two integers as arguments, defining a block of complete lines to be printed out. (n2-n1) lines will be printed out, starting with the line that ML(n1); would move the pointer to. VB(-3,3); is equivalent to -3VL,3VL; but VB may also print a block of lines not including the current pointer position, e.g. VB(2,ZZ); .

VC Verify Current line

Format:
VC E () => ()

VC is equivalent to VB(%Ø,1%), and takes no arguments. It is often convenient to include VC as the last command in a string of edit commands. It types out the characters of the line which the pointer is currently pointing to.

VW Verify Window

Format: VW E <window specifiers> => ()

VW takes a pair of window specifiers, and prints the characters in the specified window. VW(Ø,ZZ) will print out the entire contents of the buffer. VW(FOO,ME) will print out the definition of the function FOO.

VMAC Verify window with MACro expansion

Format:
VMAC E <window specifiers> => ()

VMAC verifies the contents of a window rather like VW, but any macros in the text are expanded. VMAC prints what the compiler sees after macro expansion. Note that VMAC will only recognise words as macros if they have been compiled before VMAC is used or

if they are standard macros. (Remember some standard macros such as BUG and UNBUG do not make any replacement text at all.) This command is valuable when debugging programs which use macro facilities extensively. Unfortunately the original indentation and spacing information in the file is lost, so VMAC will not print the text very neatly. It tries to insert newlines in the correct places.

Deleting Text

DC Delete Characters
DL Delete Lines

Format:
DC, DL all E <integer> => ()

DC deletes the specified number of characters from just before or just after the pointer. DL deletes the specified number of lines forwards or backwards from the pointer position. If the argument is positive, then N characters or lines are deleted from after the pointer, or text is deleted to the end of the buffer if this is reached sooner. If the argument is negative, then !N! characters or lines are deleted from before the pointer, or to the start of the buffer if this reached sooner.
ØDC; is a no-op.

DAZ

Format:
DAZ E () => ()

DAZ deletes from "A" to "Z", i.e.: the whole buffer.

DW Delete Window

Format:
DW E <window specifiers> => ()

DW deletes the contents of the specified window. The pointer is left at the point of the deletion.

GW Grab Window

Format:
GW E <window specifiers> => <grabbed object>

GW deletes the contents of a window like DW, but it returns a "grabbed object" on the stack, which may be INserted in the buffer at a later time. The grabbed object may only be inserted once, and if the GW is UNDOne that counts as an insertion of the grabbed object.

If a grabbed object is saved somewhere, and the core image is saved with POPMESS-SAVE and later restored, then the grabbed object will no longer be usable if the text in it was originally inserted from a disc file. In this case, the I/O channel being used for that inserted file is no longer connected to the file after the core image has been RESTOREd, and the characters of the grabbed object cannot be accessed. An error 173 (buffer contents using unassigned channel) will be caused when an attempt is made to access those characters.

DSF Delete: Search Forwards
DSB Delete: Search Backwards

Format: DSF, DSB all <search item?>, <count> => ()

DSF and DSB take arguments like SEF and SB respectively, and delete from the current position forwards to the end or backwards to the start of the specified search item (if it is found).

"AND"DSF; will delete forwards from the pointer position through to the end of the next occurence of the word "<and" in the buffer.

Exchange commands

XF eXchange Forwards
XB eXchange Backwards

Format:
XF, XB all E <search item?>, <count?>, <insert item> => ()

These commands search for a search item, like a search command, and then delete the text found, and insert the Insert item supplied in its place.

The Insert Item is treated exactly as in the IN command, and is supplied on the top of the stack. The remaining items on the stack (if any) then define the search string and count exactly as for the search commands. A search fail error is given if the search item cannot be found. Otherwise, the pointer is left at the end of the inserted item, as for IN.

Normally two UNDO's are necessary to reverse an exchange, the first to remove the inserted item, the second to reinsert the deleted search item. However, it is sometimes convenient to give the insert item '' - an empty string - which has the effect simply of deleting the search item; in this case, only one UNDO is required.

A common error with an exchange command is to delete the correct text, but insert the wrong item in its place. In this case, a single UNDO will remove the wrong insert item, after which you can insert a replacement.

XFT eXchange Forwards a
XBT eXchange Backwards and Test

Format:
XFT, XBT all E <search item?>, <cnunt?>, <insert item>
=> <truthvalue>

These commands parallel the previous exchange commands, but return TRUE if the exchange is performed, or FALSE otherwise. In the latter case, the pointer will not have been moved.

These commands are used similarly to the Search and Test commands, for example as the conditions of iterative expressions. For example,

```
<?"FNCTION"XFT'FUNCTION' THEN VC?>
```

will replace all occurences of "FNCTION" by "FUNCTION", verifying in each case.

Output

OP OutPut

Formats:
OP E <filename> => ()
OP E <filename>, <window specifiers> => ()

OP is a command for writing part of the buffer to a specified file on disk. It takes a filename argument (a list) and optionally a window specification. The window of the whole buffer is implied if no window is specified. The contents of the window is then written to the specified file, with an OUTBAK repeater, and the output file is closed.

If no window was specified, then OP also applies PEDRESET, which clears the buffer and the UNDO list. Note that this may destroy a file if the buffer was empty.

The normal way to write the complete buffer to disk after editing a file is with the FILE command described below. If output is required through a nonstandard character consumer, the consumer should be put into CUCHAROUT, and a verifq command used to print characters into it.

FILE FILE edited file on disk

Format:

```
FILE E () => ()
```

FILE is the usual command to terminate an editing session. It uses OP to write the entire buffer to the file whose name is in the variable PEDNAME (presumably put there by IT earlier).

FILE gives an error if NAME is zero, or if the buffer is empty. Otherwise, the buffer is written out and cleared, and the UNDO list reset.

SAVE SAVE file (during editing)

Format:

```
SAVE E () => ()
```

SAVE is useful during editing to write the current version of the file to disk, while also keeping it in the buffer. It remembers the current pointer position CP, and does a FILE, and then re-inserts the file afresh with IT and restores the pointer position.

This has the effect of clearing the UNDO list, as the price of cleaning up the buffer chain of records. If you wish to be able to undo back through a SAVE, you can get this effect by the command:

```
OP(NAME,Ø,ZZ);
```

but be sure that NAME is set and the buffer is not empty. This use of OP will not clear the buffer since the window is given explicitly.

Miscellaneous Commands

PEDMKS MaKe String

Format:
PEDMKS E <window specs> => <cstrip>

The PEDMKS command takes a pair of window specifiers, like GW, and returns a string of the characters in that window. The window is NOT deleted, and the string may be inserted anywhere else as many times as desired, or used for other purposes.

UNDO undo edit action

Format:
UNDO E () => ()

The UNDO command will reverse an individual insertion or deletion in the buffer. To save memory, the command remembers only up to six elementary undo-able actions back at any time, and an error is given if you attempt to UNDO too many.

There is one other constraint on its use: if GW is used and the 'grabbed object' is reinserted somewhere, the GW cannot be UNDONE, and an attempt to undo it will cause an error.

The PEDRESET function clears the UNDO list to contain nothing; it is called by FILE and SAVE.

PEDRESET

Format:
PEDRESET E () => ()

PEDRESET is a function which resets the editor. It clears the buffer and the UNDO list.

POPMESS-PEDMARK

POPMESS([PEDMARK <character code>]);
changes the 'marker' character printed by verification commands at the position of the pointer.

PEDAPPW APPLY function to Window

Format:

PEDAPPW E <window specifiers>, <function> => ()

PEDAPPW is a function; it takes an argument function and a window specification. It then creates an input character repeater reading from that window, and applies its argument function to that repeater function. As the repeater function is applied, it moves the pointer to the character in the buffer just returned.

For example PEDAPPW(Ø,ZZ,COMPILE) will compile the contents of the buffer like PCOMP(Ø,ZZ). This function is used to implement PEDVMAC, PCOMP, and similar facilities.

The edit commands are summarised in the file PEDS.HLP, which can be printed by the HELP PEDS command.

APPENDIX A

PED ERROR CODES

error	code	meaning
16Ø	PEDWNF	Window bound Not Found
161	PEDIWA	Illegal Window bound Argument
162	PEDIIA	Illegal Insert Arg: not string, word, number, file or grabbed object
163	PEDNDF	Not a Disk File in PEDIN
164	PEDBNE	Buffer Not Empty in PEDIT
165	PEDRGO	Reinserting Grabbed Object, already inserted or undone
166	PEDIMA	Illegal Move Argument; not an integer
167	PEDSFL	Search Fail; item not found, position unchanged
168	PEDISA	Illegal Search Arg: not string, word, number or function object
169	PEDFBB	Filing Bad Buffer: buffer empty, or no NAME
17Ø	PEDUNF	UNDO Fail; nothing to undo
171	PEDMMF	MM Fail; text not properly nested
172	PEDIBA	Illegal BVAL Arg: assigning non-pos.integer
173	PEDUAC	buffer contents using unassigned channel probably due to POPMESS-SAVE/RESTORE

INDEX