TECO on the IBM PC

Nelson H.F. Beebe
Center for Scientific Computing
Department of Mathematics
220 South Physics Building
University of Utah
Salt Lake City, UT 84112
USA
Tel: (801) 581-5254

3 June 1986 — Revision 0.04

Contents

1 What is TECO?

TECO, which stands for **T**ext **E**ditor and **CO**rrector, is a language for text editing originally developed in the early 1960's in the MIT and DEC communities. It has been implemented on virtually all DEC computers since, and now with this implementation, is available on the IBM PC inside the EPSILON text editor developed and marketed by Lugaru Software, Inc. This development is independent of Lugaru Software, and is placed in the public domain.

The most important contribution of TECO to the computing community has almost certainly been the fact that Richard Stallman extended it significantly to provide the underlying support for his wonderful EMACS text editor.[1] The compactness and efficiency of TECO, and the possibility of immediate interpretation for on-line interactive development of editor functionality must surely have been the key to the success of the EMACS project. Now that EMACS is well understood, it has been feasible to re-implement several descendants of EMACS, including EPSILON, using higher-level LISP and C (or C-like) languages for the development support.

2 Why TECO?

There are two fundamental differences between TECO and almost all other text editors:

- TECO is a complete programming language, containing the three necessary constructs—sequential execution, repetitive execution, and conditional execution—which have been mathematically proven sufficient to write any program.

- Because TECO can access the *values* of characters, it can be used for context-sensitive editing which is utterly impossible without this ability.

Although reasonable text editors generally provide some sort of shorthand for repetitive editing tasks through keyboard macro definitions, and some

[1]Richard M. Stallman, *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, in *Interactive Programming Environments*, Eds. D.R. Barstow, H.E. Shrobe, and E. Sandewall, McGraw-Hill (1984), pp. 300-325.

allow commands to be given numeric arguments to be used as repeat counts, this functionality alone is insufficient for many editing tasks, because it lacks conditional execution.

A simple example relevant to FORTRAN programming might be *find the next line in the edit buffer which is longer than 72 characters*. Since the FORTRAN language ignores text beyond column 72, and does not require explicit declarations of variables, long lines which are unlikely to be recognized as such on typical 80-column terminal screens can introduce insidious program bugs which will never be detected by the compiler, and probably not by the programmer. An editor lacking the ability to scan forward in the buffer computing line lengths and stop when the length exceeds the specified limit simply cannot help here; conditional execution is essential.

The previous example did not require the editor to have knowledge of *what* characters were being processed, only the ability to *count* them and determine line lengths.[2] As a second example, therefore, consider the task of automatically changing the case of letters in a portion of text. This might be done by a text replacement function which all editors offer, but it would likely be painful. If the requirements were tightened so that this substitution should happen only on text in certain columns of each line, or on words following a backslash, then it might well become impossible, except by tedious manual deletion-insertion editing of every such instance. Humans should not have to do this, because it is boring, repetitive, and likely to introduce many new errors.

A programmable text editor, like TECO, can handle tasks like this with very little effort, and although it may take a few tries to get the TECO program working correctly, it can perform its job on an arbitrarily large amount of text.

You may of course argue (correctly) that you can write a program for the task in your favorite language, and after a few trips round the compile–link–debug cycle, get it to work on the text. It will almost certainly take *you* much longer to do this than if you could write it in TECO, and except in LISP environments, you probably had to exit and reenter your text editor many times.

[2] Remember that not all computer systems use an actual character as an end-of-line marker.

Even if you are fluent in LISP, and have one of the EMACS descendants built on top of it, you will type many fewer characters to do the job with TECO than with LISP. On the other hand, if you are building up large libraries of editing functions to be used many times, it will be easier to read and understand your LISP code than it will if the code is in TECO, particularly uncommented TECO. I therefore feel that there is a place for both high- and low-level languages for editor extensibility.

The beauty of LISP and TECO is that their uniform syntax makes it possible to implement rudimentary versions of interpreters or compilers for the languages with very little work, compared to that required for a more conventional computer language like C. For example, the first working prototype of this TECO interpreter took two days and two evenings to get to a state where most of the planned functionality was finished and working well enough to be used to help develop itself. This was done without reference to any TECO manuals or other documentation, just sitting creating the code on the fly inside EPSILON on an IBM PC. If this interpreter were now to be moved to another system, it could probably be done in half the time.

3 How TECO Works

Until now, I have intentionally refrained from showing you any examples of what TECO looks like, and I only hinted in the last section that it might be hard to write. Before we go into this, it is important to understand why TECO looks like it does. Remember that the original TECO's were developed to run on very small machines, with perhaps as little as 4K words of memory for the complete editor and its text buffers, in a time when even small computers cost a great deal. *Efficiency* of interpretation was therefore paramount.

The syntax of TECO is such that it can be interpreted in *one* pass, looking at *one* character at a time, and with *zero* lookahead. This essentially means that each text character is assigned a function, and while `C` for *character forward*, `D` for *character delete*, and `S` for *search* are mnemonic enough, the supply of such connections is limited, and the early TECO's ended up with things like `"` for *if* and `O` for *goto*, with perhaps 30 to 50 characters given TECO significance. EMACS TECO grew even more, using up almost all 128 ASCII characters. Several of these act as escape mechanisms to access extended functions defined by multi-letter words, like `FS␣ECHO␣ACTIVE`.[3] You can see, however, that this binding of single characters to functions makes for an efficient implementation—the interpreter just keeps a table of 256 function names which it dispatches through, based on the value of the next input character.

At this point, you may be ready to throw up your hands in despair and say "I'll never learn or remember such nonsense!". Don't give up yet—without expending great effort, you can learn a small and manageable subset of TECO which will handle many editing jobs. Later, if your editing requirements expand, you can learn more. One of the goals of this document is to present TECO to you in such a way as to help you understand how it works and why, and then to guide you to the most useful parts of it to learn first. You will find a tiny subset outlined in the section beginning on page 55.

4 TECO Numbers

Numbers in TECO are integers; floating-point is so rarely useful in text processing that it is simply not provided. If you need to compute a fractional number, you must do it by appropriate scaling to maintain integer values. Numbers are signed values represented in two's-complement notation in one computer "word", which is at least 32 bits wide, even on microcomputer implementations of TECO. This is unlikely to ever pose a restraint on your programming, while 16-bit integers certainly would.

A string of digits 0..9 represents a decimal integer, as usual, and it may have a single optional leading plus or minus sign, or a complement operator, ~. Unlike the C programming language, no significance is attached to leading zeroes. Unlike older TECO's, nothing following a digit string can change its value.[4]

Representation of numbers in other than base 10 is sometimes convenient. Instead of changing a global variable containing the current input radix, a process which turns out to be fraught with danger in EMACS TECO, this version adopts the notation similar to that used in the ADA programming

[3] TECO program examples will be given in typewriter text, and a space in such text will be marked as ␣ so that it is visible.

[4] Older TECO's normally interpret a digit string followed by a period as an octal value, which requires the interpreter to back up and start over in the number evaluation.

language—an optional sign, followed by a number base expressed as a *decimal* integer in the range 2..36, followed by a `#` sign, followed by a "digit" string from the set 0..9, A..Z (or a..z, letter case being insignificant). Unlike ADA, we do not require a trailing `#` sign, since whitespace can always be used in TECO to separate a number from another token with which it might otherwise be confused. For bases larger than 10, letters count as A $\equiv$ 10, B $\equiv$ 11, ..., Z $\equiv$ 36. For example,

```
127  2#1111111  4#1333  8#177  10#127  16#7f  16#7f  32#3u  36#3i
```

are all equivalent in value.

Collection of an integer stops at the first unexpected character, and can never raise an interpreter error, unless the base is out of range. Two rules are applied:

- If only a plus or minus sign has been collected, the number is $+1$ or -1, respectively;
- Otherwise, if no legal digits have been collected, the number is 0.

Thus, we have

$$\begin{array}{rcl} + & \equiv & +1 \\ - & \equiv & -1 \\ 36\# & \equiv & 35\# \equiv \quad \ldots \quad 3\# \equiv \quad 2\# \equiv \quad 0 \end{array}$$

This means that if the interpreter tries to collect a number and finds the first character is, say, a letter, the value zero will be assumed. This is similar to the C `atoi()` function. This peculiar behavior is rare, because collection of a number usually only happens when a decimal digit or sign has been seen.

5 TECO Strings

Strings in TECO in principle may contain all possible characters in the host character set, and may have any length greater than or equal to zero, limited only by available memory. This implementation of TECO in EPSILON's C-like language, EEL, does not permit ASCII `<NUL>` in a string, since that is used as a string terminator by the language; otherwise, all character values from 1 to 255 may appear in a string.

Multi-line strings are therefore permissible, and indeed, very useful. The edit buffer is viewed as a sequence of characters, and lines are distinguishable only because the editor starts a new screen line for text following a line terminator. The line terminator is implementation-dependent. In EMACS TECO, it is `<CR><LF>`, but in this TECO, it is `<LF>`, just like it is in C and the UNIX system. This is transparent when you write TECO inside the editor—if the string looks like it has an embedded newline, then it does. Unlike C, no special representation of certain characters with backslashed letter or octal digit sequences is supported. For example, the command

```
@S/foo
bar/
```

would search for the string "`foo<newline>bar`" in the buffer.

A string object is distinguishable from a number object by virtue of the way it is collected by the interpreter. This is not true in EMACS TECO, which interprets certain integer values as addresses of strings. In EPSILON's EEL language used to implement this TECO, coercion between pointers and integers is forbidden, so objects are given explicit type codes which are checked before using an object.

We therefore have to decide what should be done when an attempt is made to use a string as an integer, or an integer as a string. This *could* be made an error, but it is more useful for it not to be. Instead, a string is coerced to an integer by applying the same rules used to collect an integer which were described in the preceding section, and an integer is coerced to a string by converting it to its decimal representation with no leading zeros, preceded by a minus sign if it is negative.

6 TECO Variables and Arguments

It has been well understood since the earliest uses of computers that the key to code reusability is the possibility of run-time modification of processing through local variables and passed arguments. Also, repetition can sometimes be expressed more easily with recursion than with iteration. TECO has all three of these features.

Recursion is easy—you just do it.

Local and global variables are available in objects called Q-registers, which are simply places to store numbers and text. We shall have much more to say about these later.

Because of the one-pass nature of the interpretation, arguments have to be available before a function code letter is recognized, so numeric arguments are given immediately before the function. There can be zero, one, or two of these. If there are two, they are separated by a comma, and the first is called the *pre-comma argument*, and the second, the *post-comma argument*, or just the *argument*, since that is the one used by functions which take a single argument.

Each function can detect how many arguments it was called with, which of them were given, whether they are strings or numbers, and whether any modifier flags have been set. It is permissible, and common, to have omitted arguments, and for a function to perform differently, depending on the modifier flags and arguments set, as well as on the particular argument values. Examples are `T`, `1T`, and `1,20T`, which type out on the terminal portions of the edit buffer.

A function which expects a string argument, such as the search command, generally reads it immediately following the invoking character. The end of the string is determined either by finding a special terminator character, `<ESC>`[5], or in an alternate form, by finding a delimiter matching the character immediately following the invoking character. Examples are `Sfoo<ESC>` and `@S/foo/`, both of which search for the string `foo` in the edit buffer.

A few functions, however, will also accept string values from the pre- and post-comma arguments, but this is uncommon enough that we shall delay describing how this is achieved.

Both numeric and string arguments are possible with some functions. For example, `-3Sfoo<ESC>` searches backwards in the buffer for the third `foo` before the current position.

Numeric arguments are not restricted to simple integer constants; they can be general numeric expressions involving integers, ASCII character values, buffer positions, and Q-registers, with operators for addition, subtraction, multiplication, and division, as well as bitwise operators for *and*, *or*, *exclusive-or*, and *complement*.

[5]the ASCII escape character, control-[, decimal 27

7 TECO Q-registers

In order to support local and global variables, TECO provides named objects called *Q-registers*. These have very simple names which allow the interpreter to find them by fast table lookup, rather than by having to collect a name string and then search a symbol table for it. The names follow one of the functions

`G`	get Q-register contents into buffer
`:I`	store string value into Q-register
`M`	execute TECO code in Q-register
`Q`	get numeric value from Q-register
`U`	store numeric value into Q-register
`X`	store buffer text into Q-register

and are a letter or digit preceded by zero, one, or two periods. Letter case does not matter; `QA` is the same as `qa`. Examples of legal Q-register references are `Q0`, `Q.0`, `Q..0`, and `QZ`. Some *illegal* references are `Q+`, `Q...`, and `Q*`.

The dotted Q-registers were not present in older TECO's, but were added to EMACS TECO to provide for commonly-used global variables which must be rapidly accessible. Their values are normally not changed by TECO programmers if disaster is to be avoided. For example, EMACS TECO's initialization function is stored in `..L`, and executing it re-initializes the interpreter; if it does not contain valid code, EMACS may crash. In this TECO, the following Q-registers are currently predefined:

`..L`	length of last string inserted, or found by search
`..R`	last regular-expression search string
`..S`	last search string
`..T`	temporary storage for coercions
`..W`	argument flags
`..X`	pre-comma argument
`..Y`	post-comma argument

They may be read, but should not be modified.

Only the undotted Q-registers should be used for local variables. Any which are used by a callable function must be preserved by that function, which is easily achieved by pushing them onto the Q-register stack, and restoring them on return. The function return mechanism will restore them automatically.

A Q-register can contain either a string, or a number, and a type code stored with it tells which of the two it is. All Q-registers are initialized to numeric values of 0 when TECO starts up. To allow a program to distinguish between the two possibilities, the function `FQq` returns -1 if Q-register `q` is a number, and otherwise, it returns the string length. Thus, if we execute `164Uq`, then `FQq` returns -1, while `:Iqfoo<ESC>` followed by `FQq` returns 3.

8 TECO Expressions

TECO is essentially an expression language; every TECO function returns 0, 1, or 2 values which become arguments to the next function executed. It is somewhat unusual in that the expressions are not restricted to numeric values; they can also be strings.

Numeric expressions are constructed from arithmetic and bitwise operators acting on terms which can be numeric variables, constants, or value-returning function references. Some examples using numeric constants are `3*5`, `3+12`, `27-12`, `47/3`, `63&15`, `32^8#37`, `7|8`, `<CTL-^>0&15`, all of which evaluate to `15`. Examples with variables and functions are `Q0*Q..L+3`, `-1,0A-<CTL-^>B`, and `@:S/foo/+1`.

String expressions can be given in function arguments, as in `@S/foo/`, but pre-comma and post-comma arguments can be produced only by function return, the most notable being `:Gq`, `:I*`, `FP`, `X*`, and `n\`, which return strings from a Q-register, the TECO program, the keyboard, the edit buffer, and a number, respectively.

The TECO interpreter is essentially nothing but a loop over the TECO program, with a call to the expression evaluator in each iteration. Because TECO is parsed left to right, one character at a time, with no lookahead, numeric expressions are handled the same way, and TECO arithmetic and bitwise operators therefore have equal precedence and guaranteed left-to-right evaluation. This is unusual in programming languages, with Ken Iverson's APL being the other major established example, although it evaluates from right to left.

When a particular evaluation order must be enforced, parentheses come to the rescue; they cause evaluation of the functions inside the parentheses to be effectively completed before outside operations are acted upon. That way, although `2 + 3 * 5` reduces to `25`, writing it as `2 + (3 * 5)` will produce the value `17` expected from most other programming languages.

Parentheses are more general, however, because they can be combined with the comma operator to cause evaluation of two or more subexpressions,[6] with the values returned by the closing parenthesis being the rightmost two subexpressions. This is analogous to comma expressions in C. What the comma operator does is nothing more than move the current post-comma argument to the pre-comma argument, and empty the post-comma argument. Any pending numeric operation acts only on the post-comma argument, and leaves any pre-comma argument intact. Here are some examples:

`(,)`	returns nothing
`(1,)`	returns `1,`
`(,2)`	returns `,2`
`(1,2)`	returns `1,2`
`1,(2)`	returns `1,2`
`2(1,)`	returns `1,2`
`(1,2,3,4)`	returns `3,4`
`((1,2),(3,4))`	returns `3,4`
`3+(1,2)`	returns `1,5`
`3*(1,2)`	returns `1,6`
`3-(,2)`	returns `,1`
`3*(2)`	returns `6`
`3,@:I*/foo/`	returns `3,foo`

In case you find these confusing, let us walk through the evaluation of `3+(1,2)`. First `3` is recognized and becomes the new post-comma argument. Next, `+` is met and the evaluator is called to get the value of the next expression which will be added to the post-comma argument. It finds the left parenthesis, so it saves the current pre- and post-comma arguments before beginning evaluation of the parenthesized list. Then `1` becomes a new post-comma argument, after which the comma moves it to the pre-comma argument, and `2` becomes the post-comma argument. Finally, the left parenthesis terminates the expression list, the pending `3` is added to the post-comma argument, `2`, giving a new post-comma argument `5`, and the existing

[6] EMACS TECO restricts parenthesized lists to no more than two subexpressions, but it became clear during the development of this TECO that there is no reason to do so.

pre-comma argument is left intact, so the final result is `1,5`. The evaluator was called six times in the process, and the calls inside the parentheses were recursive.

The colon and at-sign modifiers are saved when the open parenthesis is met, then cleared. When the close parenthesis is met, the saved modifiers are merged with their current settings. They will be set on exit from the close parenthesis if they were on when either of the parentheses were met.

9 TECO Language Reference

In this section, we present the definitions of each TECO function, arranged in ASCII lexicographic order. It is assumed that you understand TECO strings and integers, and how arguments are passed. It is not an error to have too many arguments to a function, but too few will raise an error.

For learning TECO, it is more useful to have a list of functions categorized by what they do; you will find that in the next section.

The edit buffer is viewed as a string of characters numbered 1, 2, ..., `Z`, where `Z` is a special TECO function which always returns the number of characters in the buffer. The pointer lies *between* characters, and its position is always given as the number of the character to its left; it is always represented by the function •.[7] The pointer position is conventionally called *point*, and in a screen editor, it would often be marked by a blinking cursor over the character following it, since few terminals have a symbol that can be displayed between characters. • returns the number of characters before *point*, and `Z`−• is the number of characters after *point*.

The position at the start of the buffer may be called either `0` or `B`. Position `0` therefore lies *before* the first character, and position `Z`, *after* the last character. [8]

Most functions operate to the right of *point*. If you move to position `0` (by `0J`) and insert text, it will *precede* other characters in the buffer; text

[7]This is an ordinary ASCII period, but I've made it bigger to distinguish it from sentence-ending periods.

[8]EMACS TECO permits the buffer to be temporarily *narrowed* to restrict the scope of editing operations. In such a case, `B` and `Z` bound this smaller region. `0` is still the same as `B`. All TECO functions, and therefore all EMACS functions built on top of TECO, work in this limited region until the buffer is widened again. Functions like *Save File* obviously must be programmed to widen the region temporarily during the write operation.

inserted at position `Z` will *follow* all other characters in the buffer.

Any TECO function may be given zero, one, or two arguments, and must return zero, one, or two arguments for the next function. The arguments appear immediately before the function, separated by commas. For example

`K`	has no arguments
`1,K`	has only a pre-comma argument
`,1K`	has only a post-comma argument
`1K`	has only a post-comma argument
`1,3K`	has two arguments

Several functions that take two arguments use them to define a range of characters in the edit buffer.

> An argument pair `m,n` means all characters in the buffer from, and including, pointer position `m`, up to, and including, position `n-1`, or equivalently, all characters from number `m+1` to `n`.

This convention makes •,`Z` represent all characters from the current point to the end of the buffer.

Any error condition noted in the following function descriptions always terminates the interpreter, and returns you to top level, which in our case is the EPSILON command loop.

In this document, we use the common notation `<CTL-x>` for an ASCII control character (ordinals 0...31), rather than the alternate notation `^x`, because the caret is also used in TECO, and it would be unclear whether a caret-letter pair, or a control character, was intended.[9]

Almost all functions consume (but possibly ignore) their pre- and post-comma arguments; if they themselves produce no return values, then the next function executed will not receive any arguments, unless these are explicitly written in the program. A few functions pass any existing arguments on unchanged. A function which returns a single argument makes that the new post-comma argument; any pre-existing pre-comma argument is left intact.

[9]This is still a source of confusion for me in EMACS TECO, where I often find it necessary to examine such text with cursor positioning commands in EMACS to see if it is one control character, or a caret-letter pair.

When arguments `m` and `n` are given, they represent *any* valid numeric or string expression, including constants, Q-registers, arithmetic or Boolean expressions, and function return values. A letter `q` immediately following a function name represents any valid Q-register which the function is to operate on. A few functions accept a special asterisk designator, `*`, in place of a Q-register; this signifies that they return a string value for the following function, instead of putting it in a Q-register or the edit buffer.

With these preliminaries out of the way, we can now tabulate the complete TECO function repertoire. For each function, we tell what it does, what errors can occur with it, what return values it produces, and then give one or more commented examples of its use. Some functions have an additional *Conventions* section which summarizes common programming practice in the EMACS TECO community.

`<CTL-I>`

Action: This function is the usual horizontal tab character, ASCII `<HT>`. Except in string arguments, where it is an ordinary character, it acts solely as whitespace to enhance readability.

Conventions: The default action of a tab according to the ANSI X3.4 standard is to position the following character at the output column following the one which is a multiple of 8. Thus, when given at the start of a line, the nine characters `␣␣␣␣␣␣␣␣X` and the two characters `<CTL-I>X` should look identical on a screen or hardcopy output. Tabs therefore make it convenient to line code up in columns, with the side benefit of some data compression. Used carefully, they can add considerably to code readability.

Errors: None.

Return values: Any existing arguments are unchanged.

`<CTL-J>`

Action: This function is the usual newline character, ASCII linefeed, `<LF>`. It serves as whitespace to enhance readability, and may occur as a legitimate character in text strings. It has one other important function: *it discards any existing pre-comma and* *post-comma arguments*. This means that several consecutive newlines are equivalent to a single one, and this can be usefully employed to superimpose a "paragraph" structure on the program.

Relations: The `W` function described later can also be used to discard arguments without introducing a line break.

Conventions: TECO is undeniably cryptic, even more so than computer assembly languages are, so it pays to use line breaks frequently, subject to the need for preserving returned arguments for use by following functions. As in assembly code writing, frequent meaningful commentary at the end of each line should make it possible for the reader to understand the gist of what is happening, without having to consult the actual TECO code on the line.

Errors: None.

Return values: None; any existing arguments are cleared.

`<CTL-M>`

Action: This function is the usual carriage-return character, ASCII `<CR>`. It does exactly what `<CTL-J>` does—supplies white-space for readability, and discards arguments. These two functions do the same thing, so TECO is insensitive to whether either, or both, are used as line terminators in text files.

Errors: None.

Return values: None; any existing arguments are cleared.

`<CTL-R>`

Action: Enter a recursive edit on the edit buffer. Any arguments are discarded. When the edit is exited with `exit-level`, normally bound to `C-X C-Z`, execution resumes with *point* where it was at the exit.

Errors: None.

Examples: <@:S/foo/; <CTL-R>> searches the buffer for the string "foo", exiting the loop when the search fails. At each successful match, a recursive edit is entered. This sort of thing is a common editing requirement—at several places in the edit buffer, do something automatically, and then allow local patching.

<CTL-]>

Action: This is the TECO quote character. Inside an <ESC>-terminated string, before an <ESC>, it makes the <ESC> character part of the string without terminating it. Otherwise, it is assumed that what follows it is a Q-register whose string contents are to be inserted verbatim in the text at that point. No character in the Q-register string can terminate the insertion. If the Q-register contains a number, its value will be converted to a decimal digit string which is then inserted.[10]

Errors: If the character following the <CTL-]> is neither an <ESC> nor a valid Q-register name, then an error is raised.

Return values: None.

Examples: :@I3/hello/␣:@IA/<CTL-]>3,␣there/

This puts the string hello,␣there in Q-register A.

IIhello<CTL-]><ESC><ESC> puts the string Ihello<ESC> in the buffer at *point*.

<CTL-\>

Action: Returns from a function. If issued at top-level, the interpreter simply terminates, just as if end of input had been reached.

Errors: None.

Return values: Any arguments in effect are passed back to the caller.

<CTL-^>

[10] I have found it a frequent irritant that this does not happen automatically in EMACS TECO; it raises an error instead. The :\ function can be used to convert a number to a string.

Action: This function discards its arguments, then consumes the next character following and returns its ASCII ordinal value. This keeps the programmer, and reader, from having to remember numerical values of characters. All 256 byte values (255 in this implementation, <NUL> being excluded) are allowable for the character following the <CTL-^>.

Errors: None.

Return values: Ordinal value of the following character as a post-comma argument. The pre-comma argument is unchanged.

Examples: <CTL-^>AUn QnI loads the value 65 into Q-register n, then inserts that value as a character into the buffer. This does exactly the same thing that @I/A/ would.

␣

Action: This function is the usual space character. Except in string arguments, where it is an ordinary character, it acts solely as whitespace to enhance readability.

Errors: None.

Return values: Any existing arguments are unchanged.

!comment!

Action: Paired exclamation points delimit a comment string. A comment may occur anywhere that whitespace can. However, inside a text string, exclamation point is an ordinary text character. Because TECO syntax is designed to be simple, the scans for the ends of iterations (>) and conditionals (') examine all characters, including those in comments and text strings. If either of these characters is present in a string or comment, it must be balanced by a matching character inside a nearby comment.

There is no way to quote an exclamation point in order to embed it in a comment, although putting a doubled one there (similar to the string quoting convention in many programming languages) will effectively achieve the same thing.

Errors: During a comment scan, it is an error if the end of the program is reached before the closing exclamation point is found.

Return values: Any arguments in effect are passed through unchanged.

Conventions: It is usual for short comments to be appended to each line in a TECO program, with the number of functions per line limited to two or three. Major comment sections usually have comments filling a complete line, and the first character after the opening comment is an asterisk. The following outline is typical:

```
!* blah blah blah blah !
!* more blah blah blah !
xxx     !a comment!
xx xx   !another!
<xx xx> !a loop comment!
```

Examples: A doubled exclamation point inside a comment just makes it the same as two comments: `!blah blah!!blah blah!`.

`< :@S/foo/; !<!  @I/>/ >` searches for the string "`foo`", exiting the loop when the search fails. After each "`foo`" found, it inserts the character `>`. If the `!<!` comment were absent, then the interpreter would incorrectly assume that the `>` inside the following string was the loop terminator.

`QX"E !"!  @I/'/ '` inserts a single quote if Q-register `X` is zero, and otherwise, does nothing. Again, if the double quote in the comment were not present, the quote in slashes would terminate the conditional, and interpretation would incorrectly resume at the slash following the first quote.

`"` Action: A double quote character introduces the TECO conditional, or *if* statement. It must be combined with a following single quote (apostrophe), `'`, which acts as a terminator for the *then* and *else* clauses. The argument to `"` gives the expression to be tested, and the character immediately following the `"` gives the condition for *success* of the conditional, based on the value of the post-comma argument. Any pre-comma argument is discarded.

#: This is the *else* part of a conditional; it may not have arguments, but may be separated from the `"x...'`

then part by any amount of whitespace, newlines, and comments.

A: The ASCII code for an alphabetic character; the values 65. . . 90 and 97. . . 122.

D: The ASCII code for a digit; the values 48. . . 57.

E: Zero.

G: Greater than zero.

L: Less than zero.

N: Non-zero.

U: The ASCII code for an "upper-case" character (anything other than 97. . . 122); this therefore really means "not lower-case".

The *else* part of a conditional is optional. Conditionals may be nested, and the *dangling else* problem common to many programming languages is solved in the standard way—an *else* matches the immediately preceding *then* part. The full syntax of the conditional is

```
n"x !then-part! ... '
 "# !else-part! ... '
```

Since there is no syntax to represent an *else-if* clause, nested conditionals become rather ugly:

```
n"x !then-part! ...
    n"x !first-inner-then-part! ... '
     "# !first-inner-else-part! ... '
 '
 "# !else-part! ...
    n"x !second-inner-then-part! ...
        n"x !deepest-inner-then-part! ... '
         "# !deepest-inner-else-part! ... '
     '
     "# !second-inner-else-part! ... '
 '
```

Conventions: In order to make nested conditionals at least marginally readable, extreme care should be taken in their formatting.

The form shown above keeps short *then* and *else* parts on one line, and splits longer ones across multiple lines in such a way that the single and double quotes of matching parts line up in the same column. Good commenting helps immensely.

Errors: An error is raised if the closing single quote matching an opening double quote cannot be found. It is an error if an *else* part occurs without a matching preceding *then* part.

Return values: Any arguments in effect at the end of a branch of the conditional are passed through unchanged to the following function.

Examples:

```
QL"A FTQL␣contains␣a␣letter<ESC>'
QL"D FTQL␣contains␣a␣digit<ESC>'
QL"L FTQL␣contains␣a␣negative␣number<ESC>'
QL"E FTQL␣contains␣zero<ESC>'
QL"G FTQL␣contains␣a␣positive␣number<ESC>'
```

This makes five tests to tell you something about the contents of Q-register `L`. By rewriting it with nested conditionals, we can make it somewhat faster, though harder to read:

```
QL"A FTQL␣contains␣a␣letter<ESC>'
  "# QL"D FTQL␣contains␣a␣digit<ESC>'
  '
QL"L FTQL␣contains␣a␣negative␣number<ESC>'
  "# QL"E FTQL␣contains␣zero<ESC>'
       "# FTQL␣contains␣a␣positive␣number<ESC>'
  '
```

% Action: Percent discards any arguments, then increments the number in the following Q-register by one, and returns the result as a post-comma argument value for the next function. If the Q-register contains a string, it is coerced to a numeric value before incrementing. After incrementing, the Q-register always contains a number.

Relations: The same effect on the Q-register can be obtained by writing `Qq+1Uq`, but that is longer, slower, and does not pass on a value. Incrementing by one is such a common operation that TECO and many computer architectures provide it as an efficient primitive.

Errors: Q-register name is invalid.

Return values: The incremented Q-register contents becomes the new post-comma argument, and the pre-comma argument is cleared.

Examples: Since the TECO remaining loop iteration count is not available to the program, one of the commonest uses of the increment function is to bump a separate loop counter. The program <CTL-^>AUL 26<QLI %L> initializes Q-register L to the ordinal value of the letter A, then loops 26 times inserting the character whose value is in Q-register L. The increment function executes once each loop iteration, with the result that the alphabet string ABC...XYZ is inserted into the buffer.

& Action: Ampersand is the bitwise logical *and* operator. It returns the *and* of its left and right operands.

Errors: There is no argument.

Return values: The *and* result.

Examples: 3&4 returns 0; 3&7 returns 3.

(Action: Save any colon and at-sign modifiers, and any pre- and post-comma arguments, then clear both modifiers and arguments.

Parentheses are used for expression grouping to enforce a particular operator precedence, since TECO arithmetic and bitwise operators otherwise have equal precedence and left-to-right evaluation.

They are also used for grouping a sequence of comma-separated expressions which are evaluated in order left to right.

Errors: None.

Return values: None.

Examples: See the) command, and the discussion in the section on TECO expressions.

) **Action**: Merge the current colon and at-sign modifiers and arguments with those saved by the previous matching open parenthesis.

Errors: None.

Return values: The function values returned are from the last two comma-separated expressions.

Examples: `(1,2)`, `(1,(2))`, and `(2,(1,))` all return `1,2`. `3+(1,2)` returns `1,5`. `(1,2,3,4)` returns `3,4`. `@:(1,2)S/foo/`, `@:(2)S/foo/`, and `2@:S/foo/` all search for the second following occurrence of the string "`foo`".

* **Action**: Asterisk multiplies the current post-comma argument with the value of the following expression and returns the result. Integer overflow returns a wrapped result, without error.

Errors: There is no argument.

Return values: The integer product.

Examples: $3 * 4$ returns 12; $3 * (4 * 5)$ returns 60.

\+ **Action**: Plus adds the current post-comma argument to the value of the following expression and returns the result. Integer overflow returns a wrapped result, without error.

Errors: There is no argument.

Return values: The integer sum.

Examples: $3 + 4$ returns 7; $3 + (4 * 5)$ returns 23.

, **Action**: Comma separates the pre- and post-comma arguments. It discards any existing pre-comma argument, and moves the current post-comma argument, if any, to become the new pre-comma argument.

Errors: None.

Return values: A new pre-comma argument, if there is one.

Examples: `27,` makes 27 the new pre-comma argument.

– **Action**: Minus subtracts the value of the following expression from the current post-comma argument and returns the result. Integer overflow returns a wrapped result, without error.

Errors: There is no argument.

Return values: The integer difference.

Examples: $3 - 4$ returns 7; $3 - (4 * 5)$ returns -17.

• Action: This function (an ordinary period character) returns the value of *point* in the buffer. This always lies in the range `B`...`Z`.

Relations: • is a *read-only* value; if you want to put *point* somewhere, use the jump (`J`) function.

Errors: None.

Return values: The value of *point*.

Examples: `•U4` stores the value of *point* in Q-register `4`.

/ Action: Slash divides the current post-comma argument by the value of the following expression and returns the result.

Errors: There is no argument, or the following expression is zero.

Return values: The integer quotient.

Examples: 3/4 returns 0; 60/3 returns 20; `100*•/Z` returns the percent of the edit buffer to the left of *point*.

`0-9` Action: A string of digits define a decimal number. Based numbers take the form of a decimal base in 2...36, followed by `#`, followed by a string of "digits" in that base, where `A` $\equiv$ `a` $\equiv$ 10, `B` $\equiv$ `b` $\equiv$ 11,..., `Z` $\equiv$ `z` $\equiv$ 36.

Errors: None.

Return values: The converted number.

Examples: `127`, `2#1111111`, `4#1333`, `8#177`, `10#127`, `16#7f`, `16#7f`, `32#3u`, and `36#3i` are all equivalent values.

: Action: Colon is one of two optional modifier functions (the other being `@`) which affect the behavior of the following function. The two modifiers may appear in either order. The effect of the : modifier is different for each function, as described in the other entries of this section.

Errors: None.

Return values: None. Existing arguments are passed through.

; Action: Semicolon is used for loop exiting. Without an argument, it uses the result returned by the last search function (see `R` and `S` below) as an implicit argument. It does nothing if its argument is negative, but otherwise causes execution to resume at the function following the close of the iteration (`>`) at this level. With a colon modifier, exit occurs if the argument is negative. With an `@` modifier, exit occurs if the argument is zero. With both modifiers, exit occurs if the argument is non-zero.

Errors: It is illegal to execution this function outside an iteration.

Return values: None.

Examples: `<@:S/foo/; <CTL-R> >` loops through the edit buffer searching for the string "`foo`". Each time it is found, *point* is left following the string, and a recursive edit level is entered, permitting changes to be made to the edit buffer. When the recursive edit is exited, the searching resumes from wherever you left *point*. The loop terminates when the search fails, because `S` then returns zero, which is the condition for ; to exit.

< Action: Left angle bracket begins an iteration loop whose end is marked by a *balanced* matching `>`. The iteration count is given by the post-comma argument to `<`; if it is not greater than zero, the loop is not executed at all. Any pre-comma argument is discarded.

The iteration count is stored internally in the interpreter for decrementing and testing when the next iteration begins. It is not available to the program in the loop body, and therefore if it was defined by a Q-register, that Q-register may be reused in the loop without affecting the iterations.

An omitted argument is equivalent to infinity—the loop will execute indefinitely until an error or loop exit (;) function terminates it.

Loops may be nested to a depth of at least 10; this limit is defined by stack sizes in the interpreter which are trivial to change.

If the body of the loop contains unbalanced left or right angle brackets in text strings, these must be balanced by the insertion of appropriate comments containing matching brackets; see the `!comment!` description above.

Errors: It is illegal for this function not to be matched by a right angle bracket, but this error may not be detected until it is necessary to find the loop exit to execute a ; exit function, or until the end of the program is reached.

Return values: None.

Examples: Because of this bracket balancing act, it is possible to have more than one non-comment right angle bracket which matches a left angle bracket. For example, 99<Q0"E @I/foo/ !<!>'"# @I/bar/!<!'> will insert 99 instances of "foo" into the buffer if Q-register 0 is zero, and otherwise will insert 99 instances of "bar".

= Action: Equal sign types out its arguments to the screen, converting them to a number if necessary. If there are two arguments, they are separated by a comma. It is useful in debugging for displaying numbers during the execution of a program.

Relations: The FT function can be used to display a string expression.

Errors: There must be at least one argument.

Return values: None.

Examples: <@:S#foo#; •= > loops through the buffer searching for the string "foo", exiting when the search finally fails. At each successful match, the value of *point* is displayed.

0UC <:S<CTL-]>N<ESC>; %C> FTcount␣=␣<ESC> QC= counts the number of occurrences of the string contained in Q-register N, then types the string count␣=␣n, when n will be the computed count value.

`>` **Action:** Right angle bracket is the end-of-loop function. Arguments are discarded. With no modifiers, the iteration count is decremented, and then control transfers back to the matching start-of-loop function (`<`), where the iteration count will be tested. An `@` modifier suppresses the decrement operation; since this effect can be achieved by starting the loop with no argument, giving it an infinite iteration count, it is rarely used. However, with multiple loop ends as illustrated in the example for `<`, it might be useful in a finite loop.

A loop exit function (`;`) transfers control to the character following the `>`.

Errors: None.

Return values: None.

Examples: See the `<` function examples, as well as several others in this section.

`@` **Action:** At-sign is one of two modifier functions (the other being `:`) which affect the behavior of the following function. The two modifiers may appear in either order. Functions which take string arguments interpret the `@` modifier to mean that the first character after the function code (or after the Q-register target designator, if there is one) should be used as the string delimiter, instead of <ESC>.

Errors: None.

Return values: None. Existing arguments are passed through.

`A` **Action:** `A` examines values of characters in the edit buffer. With a single argument, `n`, `nA` returns the ASCII ordinal value of the character `n` characters to the *right* of *point*. `0A` is the value of the character immediately to the left of the pointer. *Point* does not move.

With two arguments, `m,nA`, works the same as `nA`, except that if the referenced character is out of the buffer, it returns `m`, instead of raising an error.

Relations: The `:G` function can be used to obtain the value of a character in a string in a Q-register.

Errors: If $\bullet + n - 1$ is not in the range `B,Z`, the referenced character is out of the buffer, and an error is raised.

Return values: The character value, or `m` if it is outside the buffer.

Examples: `<•−Z; 1A−<CTL-^>X"E <CTL-R> '>`

Loop through the buffer searching (inefficiently) for the letter `X`, entering a recursive edit level each time an instance is found. Loop exit happens when *point* reaches the end of the buffer; this test is made before the use of `1A` because that would raise an error on the last iteration.

`<-1,1A"L 0;' 1A−<CTL-^>X"E <CTL-R> '>`

This is equivalent to the first example. Instead of testing explicitly for end-of-buffer, it uses the return value -1 to achieve the same result.

`<•−Z; (1A−<CTL-^>A)*(<CTL-^>Z−1A):"L 1AU0 Q0+32U0 D Q0I' C>`

Like the previous examples, this loops through the buffer until the end is reached. Each time an upper-case letter is found, it is extracted and deleted, incremented by 32, and then reinserted, becoming the corresponding lower-case letter.

`B` Action: `B` returns the value of *point* at the beginning of the buffer. It is always identical to zero.

Errors: None.

Return values: Zero.

Examples: `BJ` jumps to the beginning of the buffer.

`C` Action: With an argument `n`, `nC` moves the pointer right ($\mathtt{n} > 0$) or left ($\mathtt{n} < 0$) characters in the edit buffer. No movement happens if the argument is zero, or if the movement would be outside the buffer. If there is no argument, it moves right one character, so `C` works like `1C`.

Errors: Without a colon modifier, it is an error to attempt to move outside the buffer.

Return values: Without a colon modifier, none. With a colon modifier, −1 if the move succeeds, and 0 if it would fail. These are the same values returned by the `:R` and `:S` search functions.

Examples: `•US @:S/foo/"L −Q..LC'"# QSJ'` searches for the next occurrence of the string "`foo`", then backs up over it, placing *point* before the string. If the search fails, *point* is left unchanged, because the *else* part of the conditional will be executed, and it jumps back to the start position. Q-register `..L` contains the length of the last string found by a search function, or inserted in the buffer; it is precisely an application such as this that makes `..L` useful.

`D` Action: With an argument `n`, `nD` deletes `n` characters right ($n > 0$) or left ($n < 0$) of the pointer in the edit buffer. No deletion happens if the argument is zero. It is *not* an error if the argument count would take the deletion outside the buffer; `999999D` will probably delete the remainder of the buffer, unless it is enormous. Deleted text is lost forever; saving of deletions must be explicitly programmed—the `FX`, `K`, and `X` functions are good for that.

Relations: The kill function, `K`, can be used both to kill a character range in the edit buffer, to kill to beginning-of-line and end-of-line, and to kill several lines. The copy function, `X`, can be used to copy text without deleting it from the edit buffer. `FX` combines the actions of `X` and `K` to move a block of text out of the edit buffer and into a Q-register.

Errors: None.

Return values: None.

Examples: `BJ <@:S/foo/; -Q..LD>` will delete all instances of the string "`foo`" in the buffer, leaving *point* at the end of the buffer. Q-register `..L` contains the length of the last string found by a search function. One could have changed `-Q..L` to `-3D` in this example, but that would not work if the search string had unknown length, which would be the case if it had a Q-register string insertion.

`F` **Action:** `F` is a dispatch character for a small family of functions which just do not fit in the limited single-character TECO repertoire.

Errors: None.

Return values: None.

Examples: Read on.

`FI` **Action:** `FI` reads one input character from the keyboard and returns its value. The read blocks until input is available. With a colon modifier, the character is read and returned, but stuffed back into the input buffer where it will be read again by the next keyboard read.

Relations: `FP` can be used to read a string from the keyboard. The `FT` function can be used to type a prompt string if this is required.

Errors: None.

Return values: The ASCII ordinal value of the character read.

Examples: `<@:S/foo/; FTQuit␣?<ESC> ␣FI-<CTL-^>Q@;'>`

This searches the edit buffer for the string "`foo`", then prompts for input and reads a single character. If the character is `Q`, the loop is exited; otherwise, the search continues until end-of-buffer is reached.

`FK` **Action:** `FK` returns minus the length of the last string inserted into the buffer with `I`, `G`, or `\`, or found by an `R` or `S` search function. If the search was backwards, then `FK` returns the positive length; that way, `FKC` will always go to the other end of the matching string, no matter what the search direction was. In this implementation, this length is stored in Q-register `..L`, so `FK` is equivalent to `-Q..L`, or `Q..L` after a backward search.

Errors: None.

Return values: Minus the string length.

Examples: `<:@S/foo/; FKC <CTL-R> >` searches the buffer for the string "`foo`", backs up over it, and enters a recursive edit.

`FPprompt<ESC>`

Action: FP issues a prompt to the screen, then waits for a string to be entered from the keyboard. In the EPSILON implementation, the string input is terminated by an unquoted newline. The read blocks until input is available.

Relations: FI can be used to read a single character. :Gq returns a string or substring from a Q-register. :I* returns a string from the TECO program. X* returns a string from the edit buffer. n\ returns a digit string.

Errors: None.

Return values: The input string.

Examples: `FPPattern:␣<ESC>U1` prompts for a pattern string and returns it in Q-register 1.

FQq Action: FQ is used to distinguish a number from a string in a Q-register, and to obtain string lengths.

Errors: Invalid Q-register.

Return values: −1 if the Q-register contains a number, and otherwise the length of the string in the Q-register (which could be 0).

Examples: `3U0 FQ0=` types −1 to the screen; `@:I1/foo/ FQ1=` types 3.

FTstring<ESC>

Action: FT types its string argument to the screen. The = function will type a numeric argument.

Relations: The T function can be used to type portions of the edit buffer to the screen. FP can be used to prompt for, and return, string input. =, FT, and T are all useful for debugging output.

Errors: No following string found.

Return values: None.

Examples: `•−Z"E FTEmpty␣buffer<ESC>'` will tell you if the buffer is empty.

FX Action: FX combines the kill and copy functions efficiently to produce one which moves text out of the edit buffer into a Q-register or argument string. `nFXq` is equivalent to `nXq nK`, and `m,nFXq` is equivalent to `m,nXq m,nK`.

An `@` modifier makes the text append to whatever is in the Q-register (it will be coerced to a string first if necessary), just like it does for the `X` function.

Errors: Invalid Q-register name. Character range out of buffer.

Return values: None.

`G` Action: `Gq` gets text from a Q-register `q` into the edit buffer, or with a colon modifier, returns the text as an argument. If the Q-register contains a number, its decimal string representation is returned.

With a post-comma argument and colon modifier, `n:Gq` returns as a *number* the character after position `n` in the Q-register (counting $0, 1, 2, \ldots, length - 1$).

With two arguments, `m,nGq` returns characters from positions `m`...(`n` − 1) in the Q-register as a string value. Without the colon modifier, the string is inserted into the buffer; with it, the string becomes the new post-comma argument. If the Q-register contains a number, the substring range is ignored and the entire converted string is returned, since a substring of an arbitrary number is unlikely to be meaningful.

Relations: The `:I` function can put text from the TECO program into a Q-register, and the `X` function will put text from the edit buffer into a Q-register. `n\` converts a number to a digit string.

Errors: Invalid Q-register or character position or substring range out of bounds.

Return values: None if no modifiers. With a colon modifier, returns the text instead of inserting it into the buffer.

`H` Action: `H` is shorthand for the whole buffer; it is equivalent to the range specification `B,Z`.

Errors: None.

Return values: `B,Z`.

Examples: `HK` is a quick way to delete the whole edit buffer.

`I` Action: With no argument, `I` inserts the following `<ESC>`-terminated string into the buffer at *point*, moving *point* to the end of the inserted text. With an `@` modifier, the string is delimited instead by the character following the `I`.

With a single numeric argument, `nI` inserts that character into the buffer, and with two arguments, `m,nI` inserts `m` copies of the character whose ordinal is `n`. It does this much more efficiently than a loop, `m<nI>`, would do. For numbers out of the range 0...255, the higher bits are simply truncated; no error is raised.

The length of the inserted string is saved in Q-register `..L` and is available there, or as its negative from the function `FK`, for later use.

With a colon modifier, a Q-register name must follow, as `Iq`, and insertion happens into that register, instead of into the buffer. The length of the inserted string is *not* saved. Single and double arguments, and the `@` modifier work as before, and in the latter case, the delimiter character follows the Q-register name.

Instead of a Q-register name, an asterisk may be used, which means return the string as an argument, instead of putting it in a Q-register or the edit buffer.

Relations: `:Gq` returns a string or substring from a Q-register. `X*` returns a string from the edit buffer. `n\` returns a digit string. After storing their returned value in a Q-register with `Uq`, a subsequent `@I/<CTL-^>q/` or `Gq` can insert them in to the edit buffer.

Errors: Invalid Q-register name. Unterminated string.

Return values: None, unless `:I*` is used, in which case, the string.

Examples: `Ifoo<ESC>` inserts "`foo`" into the buffer after *point*. `@I/foo/` does the same thing. `@I/foo/ FKC` inserts, then backs up over the string, leaving *point* in front of it.

`32I` inserts a space (ASCII space = 32) into the buffer. `80,<CTL-^>*I` inserts 80 asterisks.

`32:IS` puts a single space into Q-register `S`, as does `@:IS/␣/`. `80,<CTL-^>*:IA` inserts 80 asterisks into Q-register A.

`@:I*/foo/` returns "`foo`" as a string value, so `@:I*/foo/UY` and `@:IY/foo/` both put the string in Q-register `Y`.

`J` **Action**: `J` jumps to the buffer position determined by the argument value. It is an error if this position is outside the buffer. A colon modifier suppresses the error and returns −1 if the jump is successful, and 0 if it is not; *point* does not move in the latter case. If there is no argument, `B` is assumed, so *point* moves to the beginning of the buffer.

Errors: Position is out of bounds, and there is no colon modifier.

Return values: None without a colon modifier. With one, −1 if the jump succeeds, and 0 if it fails.

Examples: `J`, `0J`, and `BJ` all jump to the beginning of the buffer, and `ZJ` jumps to the end. •+7`J` moves forward 7 characters.

`K` **Action**: `K` is the kill function. With a single argument, the kill acts on lines. `nK` kills from point to, and including, the `n`th newline following. An omitted argument is equivalent to an argument of 1. If the argument is negative or zero, then it backs up over $1-$`n` newlines, spaces forward over the last one, and kills from there to *point*. Thus `0K` kills to beginning of line.

There is an implicit newline at the beginning of the edit buffer.

A colon modifier causes it to move backwards over the last newline before the kill, so `:K` kills to end-of-line, but does not join the current line to the following one.

The text range selected with one argument is the same as that for the `L`, `T`, and `X` functions.

With two arguments, `m,nK` kills characters in that buffer range, and leaves point at `m`. No text is killed if the range is empty, that is, `m` $\geq$ `n`, and *point* does not move.

It is *not* an error if a single argument count would take the deletion outside the buffer; `999999K` will probably delete the remainder of the buffer, unless it is enormous. Deleted text is lost forever; saving of deletions must be explicitly programmed—the `FX` and `X` functions are good for that.

Relations: The `nD` function deletes `n` characters, so it is equivalent to `•,•+nK` when $n > 0$, and to `•+n,•K` when $n \leq 0$.

Errors: None.

Return values: None.

Examples: `0K` kills to beginning-of-line, `:K` to end-of-line. `0L 1K` kills the entire current line, as does `:L -K`. `HK` is the standard way to delete the entire buffer. `B,•K` kills from buffer to *point*, and `•,ZK` kills from point to end of buffer.

`L` Action: `L` provides line-oriented movement in the buffer. With a single argument, `nL` moves forward over `n` newlines, leaving *point* at the beginning of the line. An omitted argument is equivalent to an argument of 1. If the argument is negative or zero, then it backs up over $1 - n$ newlines, then spaces forward over the last one, again leaving *point* at the beginning of the line.

There is an implicit newline at the beginning of the edit buffer.

With a colon modifier, it backs up over the last newline.

The text range selected with one argument is the same as that for the `K`, `T`, and `X` functions.

With two arguments, `m,nL` is equivalent to $m + n - \bullet$`J`. There is no good reason why this should be so, except history.

Errors: Character range out of buffer.

Return values: None.

Examples: `0L` moves to the beginning of current line, `-L` to the beginning of the previous line, and `L` to the beginning of the next line.

`:L` moves to the end of the current line, `0:L` to the end of the previous line, `-:L` to the end of the second previous line, `1:L` to the end of the following line.

`M` Action: `M` is the function caller. It expects either a following Q-register, or with an `@` modifier, a function name as a delimited string. In the first case, the Q-register should contain a string representing a TECO program to be executed. The function return, `<CTL-\>`, may set the arguments for the next function. The TECO interpreter is invoked recursively on this program, but it inherits the entire environment of edit buffers and Q-registers. If it uses any Q-registers, it is good practice to save them on the stack with the `[` function; the function return will automatically restore them.

If the Q-register has a number in it, then it is taken to be a key number which is used to dispatch to the function normally bound to it; there is no return value. Key numbers are just the ASCII ordinal value of the character, with 128 added for a control key, and 256 added for a meta key. Thus, `C-A` is either 1 (`<CTL-A>`) or 129, `M-A` is 321 (256 + 65), and `C-M-A` is 452 (256 + 128 + 65).

In the second case, `@M/foo/` calls the EPSILON function `foo()`. It is not possible to pass arguments to such a function, but many of the interesting ones, and all of the `command` functions, take none anyway. A numeric argument to `@M` may become an iteration count for the function, but most EPSILON functions rely on the top-level command loop to call them multiple times if an iteration count is specified. There is no return value, and no error indication if the EPSILON function does not exist—`@M/foo/` then becomes a no-op.

Since EMACS uses TECO to write virtually all of its functions, `M` is used heavily. In order to support named functions (which this TECO does not), it needs a lookup mechanism to return the TECO code associated with a name; the function to perform this is normally in Q-register `.M`. A named function call then usually looks like `M(M.Mfoo)stringarg<ESC>`, which says *run the function in* `.M` *to find the code for foo, and return it as a string value for the outer* `M` *to execute, passing it the following string argument.* This TECO enjoys the benefit of being written on top of a *compiled* high-level language, EEL, instead of the other way around, so it is unlikely that invocation of functions written in TECO will be of much interest.

Nevertheless, the mechanism is there.

Errors: Invalid Q-register, or error in the invoked function.

Return values: None in the case of a key number dispatch or EPSILON function call. Otherwise, the values set by the function return.

Examples: `<CTL-^><CTL-I>U0 M0` will call the EPSILON function normally bound to horizontal tab, which is usually some indenting function. `@M/indent-previous/` would call the function which is usually bound to tab.

`Q` Action: Return the value in the following Q-register as a string or number. A Q-register name is a digit or letter preceded by zero, one, or two periods. The dotted Q-registers are reserved for internal uses; they may be read, but should not be written into.

If the string is used in a numeric expression, it is coerced to a number, but the Q-register remains a string.

Relations: `Q` provides a way of getting values *out* of Q-registers into arguments or expressions; `G` and `<CTL-^>` get their contents into the edit buffer. `:I`, `U`, and `X` allow storing values *into* Q-registers.

Errors: Invalid Q-register name.

Return values: The Q-register contents.

Examples: `Q0U1` copies Q-register 0 to Q-register 1. This is designed to work for both strings as well as numbers. Another way to do this is `:I1<CTL-^>0<ESC>`, which may be less efficient. `1` + `Q0U0` increments a number in Q-register 0, but this is more efficiently done with `%0`.

`R` Action: `R` is the regular-expression search function, which provides a powerful pattern matching search facility. Regular expressions supported by EPSILON are strings with the following contents:

- Any character except the special characters below matches itself.

- % followed by a character matches that character; it is used to quote the characters listed below to protect them from any special interpretation.
- ^ at the start of a complete pattern string restricts matches to start at the beginning of a line. A ^ anywhere else in the pattern is an ordinary character.
- $ at the end of a complete pattern string restricts matches to finish at the end of a line. $ elsewhere in a pattern is an ordinary character.
- • matches any character except newline.
- `[s]` for a string of characters, `s`, matches any character in `s`. `[^s]` matches any character *not* in `s`. In `s`, `a-b` stands for all characters from `a` to `b`, inclusive. The `]` character can only appear as the first character in `s`.
- `(s)`, where `s` is any pattern, matches what `s` matches.
- `s*` matches *zero or more* occurrences of one of the above patterns, `s`.
- `s+` matches *one or more* occurrences of one of the above patterns, `s`.
- `s|t` matches either of the patterns `s` or `t`.
- `st`, a pattern `s` followed by a pattern `t` matches a match for `s` followed by a match for `t`.
- A `!` in a pattern makes the search consider it to be the end of the pattern; that is, `foo!bar` matches only a `foobar` in the buffer, but *point* is left after `foo`.

`R` is followed by an `<ESC>`-terminated string; with an `@` modifier, the string is delimited by the character following `R`.

With no modifiers, a failing search raises an error. With a colon modifier, the search returns -1 on success, and 0 on failure.

The length of the last successful match is preserved in Q-register `..L`, and its negative is available from the function `FK`. Since *point* is left following the matched string, this provides a way to find its beginning.

The last regular-expression search string is saved in Q-register ..R, and will be substituted for a null string given to a R function.

With an argument n, it finds the nth match forward ($n > 0$) or backward ($n < 0$) in the edit buffer. It is a no-op if the argument is 0.

There is no way to limit the search; it will proceed to the beginning or end of the buffer.

Relations: The S function provides normal string searching.

Errors: Incomplete string. With no colon modifier, failing search.

Return values: None without a colon modifier. With a colon, -1 on success, and 0 on failure.

Examples: <@:R/[A-Z][a-z0-9]*/; <CTL-R>> searches the buffer for the next capitalized word (upper-case letter followed by lower-case letter or digit), entering a recursive edit after each match, and exiting the loop when the match fails.

@:R/[0-9]%•[0-9]*[EDed][+-][0-9]+/ searches for the next floating-point number string in the buffer. Alternate patterns would be required to find all such forms allowed by most programming languages.

S Action: S the normal search function. It is normally followed by an <ESC>-terminated string; with an @ modifier, the string is delimited by the character following S.

With no modifiers, a failing search raises an error. With a colon modifier, the search returns -1 on success, and 0 on failure.

The length of the last string found is preserved in Q-register ..L, and its negative is available from the function FK. Since *point* is left following the matched string, this provides a way to find its beginning.

The last normal search string is saved in Q-register ..S, and will be substituted for a null string given to a S function.

With an argument n, it finds the nth match forward ($n > 0$) or backward ($n < 0$) in the edit buffer. It is a no-op if the argument is 0.

There is no way to limit the search; it will proceed to the beginning or end of the buffer.

Relations: The `R` function provides regular-expression pattern matching.

Errors: Incomplete string. With no colon modifier, failing search.

Return values: None without a colon modifier. With a colon, −1 on success, and 0 on failure.

Examples: `<:@S/foo/; FKD @i/bar/>` replaces all instances of `foo` in the edit buffer by `bar`.

`T` Action: `T` types text from the edit buffer to the screen. No matter what the selected text range is, *point* does not move.

With a single argument, `nT` types the text from point up to the `n`th following newline. An omitted argument is equivalent to an argument of 1. If the argument is negative or zero, then it backs up over $1 -$ `n` newlines, spaces forward over the last one, and types from there to *point*.

The text range selected with one argument is the same as that for the `K`, `L`, and `X` functions.

There is an implicit newline at the beginning of the edit buffer for the `K`, `L`, and `X` functions, but you cannot find it with the search commands or the `A` command.

With a colon modifier, it backs up over the last newline.

With two arguments, `m,nT` types that range of characters.

Errors: Character range out of buffer.

Return values: None.

Examples: `0T` types the text to the beginning of current line, `-T` types text to the beginning of the previous line, and `T` types text to the beginning of the next line.

`:T` types text to the end of the current line, `0:T` types text to the end of the previous line, `-:T` types text to the end of the second previous line, `1:T` types text to the end of the following line.

`U` **Action**: `U` stores into a Q-register. With a single argument, `nUq` puts the value `n` into Q-register `q`. Normally, `n` is a numeric value. However, it can also be a string value set in the argument by `Q` or `:I*`. No value is returned.

With two arguments, `m,nUq` stores the post-comma argument, and passes the pre-comma argument out as a return value. That way, `m,nUq Up` does what `nUq mUp` would do. This is how an argument pair can be recovered and stored for later use.

Relations: Text can also be inserted into Q-registers with `:I` and `X`, and both numbers and strings can be popped into a Q-register with `]`, or restored by the return, `<CTL-\>`, in a function.

`G` and `I` can get Q-register contents into the edit buffer.

`=` and `FT` can type Q-registers to the screen.

Errors: Invalid Q-register, or no argument.

Return values: None with a single argument; otherwise the pre-comma argument.

Examples: `65U0` puts the number 65 in Q-register 0. `HUZ UB` puts `B` in Q-register `B`, and `Z` in Q-register `Z`.

`W` **Action**: `W` simply discards whatever arguments exist. It is used when returned arguments from a function are not wanted for the next one, and both are written on the same line of the program. It is often used in EMACS TECO to discard function returns after `M(M.M␣foo)`, but is rarely needed in ordinary TECO.

Relations: Newline (both `<CTL-J>` and `<CTL-M>`) also discards arguments.

Errors: None.

Return values: None.

Examples: `HWK` kills from *point* through the following newline, instead of the whole buffer, since `W` discards the pair `B,Z` returned by `H`, and `K` without an argument kills one line. `@:S/foo/ WC` searches for "`foo`", then advances *point* one character; if the

W were omitted, the returned value -1 from a successful search would move *point* left one character, placing it between the two o's.

X Action: X copies (think, *Xfers*) text from the edit buffer into a Q-register, or returns it as a string argument, if an asterisk is given in place of a Q-register name. *Point* does not move. With a single argument, it acts on lines. nX selects from *point* to the n^{th} newline following. An omitted argument is equivalent to an argument of 1. If the argument is negative or zero, then it backs up over $1 - n$ newlines, spaces forward over the last one, and copies from there to *point*. Thus 0X copies text from the beginning of the current line to *point*.

There is an implicit newline at the beginning of the edit buffer.

A colon modifier causes it to move backwards over the last newline before the copy, so :X copies to end-of-line.

An @ modifier causes the selected text to be *appended* to the text currently in the Q-register. If that is not already a string, it is converted to one.

The text range selected with one argument is the same as that for the K, T, and X functions.

With two arguments, m,nXq copies the text range m,n into the Q-register q. If the range is empty, a null string is placed in the Q-register.

Relations: :Gq returns a string or substring from a Q-register. :I* returns a string from the TECO program. n\ returns a digit string.

Errors: Invalid Q-register name. Character range out of buffer.

Return values: None unless X* is used, in which case the selected string is returned.

Examples: 0Xq copies from the beginning of the current line to *point*, :Xq copies from *point* to the end of the current line. 0L 1Xq copies the entire current line, including the newline. HXq is the standard way to copy the entire buffer. B,•Xq copies from buffer to *point*, and •,ZXq copies from *point* to end of buffer.

`Z` **Action:** `Z` returns the end of buffer position, equivalent to the number of characters in the edit buffer.

Errors: None.

Return values: The end of buffer position.

Examples: `ZJ` jumps to the end of the buffer.

`[` **Action:** `[` pushes the following Q-register onto the Q-register stack. Any argument is then stored in that Q-register. That is, `n[A` is equivalent to `[A nUA`, but faster. This operation is useful for temporary storage of a Q-register; it can be recovered with the pop function, `]`.

There is only one Q-register stack for all Q-registers, and for all recursive levels of the TECO interpreter. This permits a caller to push arguments onto the stack which are popped by a called function, though this usage is extremely rare, since TECO programmers are accustomed to managing with no more than two arguments at one time.

The Q-register stack is unusual in that it records the name as well as the value of the Q-register. The pop function, `]`, discards the name and returns the value, but the function return, `<CTL-\>`, restores each value pushed since function entry to its original Q-register.

Errors: Stack overflow. Invalid Q-register name.

Return values: None.

Examples: `1[N 5<@I/v/ GN @I/,␣/> -D ]N` saves Q-register `N`, initializes it to 1, then inserts `v1,␣v2,␣v3,␣v4,␣v5,` into the buffer, and restores Q-register `N` to its original value.

`\` **Action:** The backslash function converts numbers to strings in the buffer or returned argument, and vice versa.

With no arguments, it converts the digit string in the buffer beginning at *point* to a number, moving *point* past it, and returns the converted numeric value. *Point* does not move if it is not at a number, and 0 is then returned.

With a single argument, the backslash function converts the argument to a decimal string and inserts it in the buffer, leaving *point* after it.

With two arguments, `m,n\` converts the `n` to a string blank-padded (`m` > 0) or zero-padded (`m` < 0) on the left in a field of width `abs(m)` characters, leaving *point* after it. If `m` is not big enough, it will be ignored and the string inserted with as many digits as are necessary.

When text is inserted in the buffer, the length of the insertion is stored in Q-register `..L`, where it is available, together with its negative returned by `FK`, for use.

With a colon modifier and one or two arguments, the converted number is returned as a string value, instead of being inserted in the buffer.

Relations: `:Gq` returns a string or substring from a Q-register. `:I*` returns a string from the TECO program. `X*` returns a string from the edit buffer.

Errors: None.

Return values: With no arguments, the value of the digit string in the edit buffer. Otherwise, none.

Examples: With *point* before the text `3.14159`, `\N C \F` puts the number 3 in Q-register `N`, and 14159 in Q-register `F`, leaving *point* after the 9.

`\U0 FKD Q0*Q0\` replaces a number in the buffer by its square.

`1UN 999<@I/read␣F/ -3,QN\ 10I %N>` inserts the lines

```
read F001
read F002
...
read F999
```

into the buffer.

] Action: `]q` pops a value from the Q-register stack into Q-register q. See the description of the push function, `[`, for details.

Errors: Stack underflow. Invalid Q-register name.

Return values: None.

Examples: `[A]B` is another way to do `QAUB`, that is, copy the contents of Q-register `A` to Q-register `B`.

ˆ **Action**: Caret is the bitwise logical *exclusive or* operator. It returns the *exclusive or* of its left and right operands.

Errors: There is no argument.

Return values: The *exclusive or* result.

Examples: 3ˆ4 returns 7; 3ˆ3 returns 0.

| **Action**: Vertical bar is the bitwise logical *or* operator. It returns the *or* of its left and right operands.

Errors: There is no argument.

Return values: The *or* result.

Examples: 3|4 returns 7; 3|3 returns 3.

~ **Action**: Tilde is the bitwise logical *complement* unary operator. It returns the one's complement of its right operand.

Errors: None.

Return values: The one's complement result.

Examples: ~0 returns -1 on a two's complement machine.

10 TECO Function Guide

This section is devoted to a summary of TECO functions by what they do. Only brief descriptions are given in the table entries; you will have to consult the language reference section beginning on page 11 for the details.

10.1 Modifiers

`@` Set the `@` modifier for the next function.

: Set the colon modifier for the next function. The : and `@` modifiers can occur in any order.

10.2 Numbers

Digit strings form decimal numbers. Based numbers are represented by a decimal base in 2…36, followed by a `#` sign, followed by a "digit" string, where "digits" are taken from the set 0…9, A…Z, and letter case is *not* significant. Examples are `255`, `8#377`, and `16#ff`.

10.3 ASCII

`<CTL-^>` Ordinal value of the next ASCII character following the `<CTL-^>`.

`nA` The ordinal value of the character `n` characters to the right of *point*. `0A` is the character to the left of *point*.

`m,nA` Like `nA`, except that it returns `m` if the character address is out of the buffer.

`FI` Next input character from keyboard.

`:FI` Next input character from keyboard, but the character is left in the input stream where it will be read again.

10.4 Bitwise Operators

All bitwise and arithmetic operators have equal precedence and are evaluated from left to right; use parentheses to obtain other evaluation orders. String arguments are implicitly coerced to integers.

`&` Binary bitwise *and*.

`^` Binary bitwise *exclusive or*.

`|` Binary bitwise *or*.

$\sim$ Unary bitwise *complement*.

10.5 Arithmetic Operators

All bitwise and arithmetic operators have equal precedence and are evaluated from left to right; use parentheses to obtain other evaluation orders. String arguments are implicitly coerced to integers.

+ Binary addition. Unary plus is treated like binary plus by supplying an implicit zero operand.

− Binary subtraction. Unary minus is treated like binary minus by supplying an implicit zero operand.

`*` Binary multiplication.

`/` Binary division.

10.6 Expressions

All TECO functions may be regarded as expressions, and a function can return 0, 1, or 2 string or number results which can be used in another expression. A parenthesized list of comma-separated expressions returns up to two values from the last two (rightmost) evaluations.

10.7 Arguments

A TECO function expects 0, 1, or 2 arguments. Arguments are separated by the comma operator, and are called the pre- and post-comma arguments. In this TECO, the arguments and `@` and `:` modifier flags are stored in double-dotted Q-registers. It should rarely be necessary to reference them as such.

`..W` The flags. Bit value 1 is set when there is a post-comma argument. Bit value 2 is set for a pre-comma argument. Bit value 4 is set for an `@` modifier, and bit value 8 is set for a colon modifier. For example, if all are set, the value will be $1 + 2 + 4 + 8 \equiv 15$.

`..X` The pre-comma argument.

`..Y` The post-comma argument.

10.8 Position

• Current position in the edit buffer. When • is `n`, it lies *between* characters `n` and `n+1`, where the first character is numbered `1`.

`B` Value is 0—the beginning of the edit buffer.

`H` Equivalent to `B,Z`, a range representing the *whole* buffer.

`Z` Value is the number of characters in the edit buffer.

10.9 Moving Point

`nJ` Move *point* to position `n` in the edit buffer.

`n:J` Like `nJ`, but return -1 on success and 0 if the movement would be out of the buffer.

`nC` Move *point* `n` positions left (`n` $<$ 0), or right (`n` $>$ 0).

`n:C` Like `nC`, but return -1 on success and 0 if the movement would be out of the buffer.

`nL` Move *point* to beginning of $\mathtt{n}^{th}$ line after current line. `0L` moves to beginning of current line, `-1L` to beginning of previous line.

`n:L` Like `nL`, but move back over newline in direction of start position. `:L` moves to end of current line.

`m,nL` Same as `m` + `n`−•`J`.

`\` Move *point* right over number and return its value. Both decimal and based integers are supported.

10.10 Copying Text

`QpUq and sUp` Copy the contents of Q-register `p`, or string argument `s`, to Q-register `q`.

`nXq` Copy text in edit buffer from *point* to the $\mathtt{n}^{th}$ following newline into Q-register `q`.

`n:Xq` Like `nXq`, but move back over newline in direction of start position before copying.

`n@Xq` Append text in edit buffer from *point* to the `n`th following newline to text in Q-register `q`.

`n@:Xq` Like `nXq`, but move back over newline in direction of start position before appending.

`n\` Insert printed decimal representation of number `n` into edit buffer.

`n:\` Return printed decimal representation of number `n` as a string argument.

`m,n\` Insert printed decimal representation of number `n` into edit buffer right-adjusted in a field of width abs(`m`) characters. The field is blank filled if `m` > 0, and zero filled if `m` < 0. If the field is too small, it will be expanded and fill will be suppressed.

`m,n:\` Return printed decimal representation of number `n` as a string argument. The string is right-adjusted in a field of width abs(`m`) characters. The field is blank filled if `m` > 0, and zero filled if `m` < 0. If the field is too small, it will be expanded and fill will be suppressed.

10.11 Deleting Text

TECO deletion commands do not save deleted text anywhere unless this is explicitly programmed.

`nD` Starting at *point*, delete `n` characters left (`n` < 0) or right (`n` > 0).

`nK` Kill from *point* through the `n`th following newline.

`n:K` Like `nK`, but move back over newline in direction of start position before killing.

`m,nK` Kill range of characters in positions `m`$\ldots$`n` $-$ `1`.

`nFXq` Kill from *point* through the `n`th following newline, saving the killed text in Q-register `q`.

`n:FXq` Like `nFXq`, but move back over newline in direction of start position before killing, saving the killed text in Q-register `q`.

`m,nFXq` Kill range of characters in positions `m`...`n` − `1`, saving the killed text in Q-register `q`.

`n@FXq` Kill from *point* through the $\mathtt{n}^{th}$ following newline, appending the killed text to Q-register `q`.

`n@:FXq` Like `n@FXq`, but move back over newline in direction of start position before killing, appending the killed text to Q-register `q`.

`m,n@FXq` Kill range of characters in positions `m`...`n` − `1`, appending the killed text to Q-register `q`.

10.12 Inserting Text

`Gq` Insert text from Q-register `q` in the edit buffer. If the Q-register contains a number, it is coerced to a string and inserted.

`m,nGq` Insert substring of text from Q-register `q` in the edit buffer.

`:Gq` Return text from Q-register `q` as a string argument.

`m,n:Gq` Return substring of text from Q-register `q` as a string argument.

`nI` Insert ASCII character with ordinal value `n`.

`m,nI` Insert `m` copies of ASCII character with ordinal value `n`.

`I and @I` Insert text from TECO program in the edit buffer.

`:Iq` Insert text from TECO program in Q-register `q`.

`:I*` Return text from TECO program as a string argument.

`n\` Insert printed decimal representation of number `n` into edit buffer.

`n:\` Return printed decimal representation of number `n` as a string argument.

`m,n\` Insert printed decimal representation of number `n` into edit buffer right-adjusted in a field of width $abs(\mathtt{m})$ characters. The field is blank filled if $\mathtt{m} > 0$, and zero filled if $\mathtt{m} < 0$. If the field is too small, it will be expanded and fill will be suppressed.

`m,n:\` Return printed decimal representation of number `n` as a string argument. The string is right-adjusted in a field of width $abs(\mathtt{m})$ characters. The field is blank filled if $\mathtt{m} > 0$, and zero filled if $\mathtt{m} < 0$. If the field is too small, it will be expanded and fill will be suppressed.

10.13 Coercions

This TECO automatically coerces strings to integers and vice-versa as required by context. For example, inserting a number into the buffer by `Gq` will convert the number to a decimal string, and adding a constant to a Q-register which contains a string will implicitly coerce the string to an integer. Q-register contents are *not* modified by coercions.

It is often useful to perform explicit coercions. You can convert an integer to a decimal string by:

`Gq` Insert number from Q-register `q` as a decimal string in the edit buffer.

`:Gq` Return number from Q-register `q` as a decimal argument string.

`nI` Insert ASCII character with ordinal value `n`.

`m,nI` Insert `m` copies of ASCII character with ordinal value `n`.

`n\` Insert printed decimal representation of number `n` into edit buffer.

`n:\` Return printed decimal representation of number `n` as a string argument.

`m,n\` Insert printed decimal representation of number `n` into edit buffer right-adjusted in a field of width $abs(\mathtt{m})$ characters. The field is blank filled if $\mathtt{m} > 0$, and zero filled if $\mathtt{m} < 0$. If the field is too small, it will be expanded and fill will be suppressed.

`m,n:\` Return printed decimal representation of number `n` as a string argument. The string is right-adjusted in a field of width *abs*(`m`) characters. The field is blank filled if `m` > 0, and zero filled if `m` < 0. If the field is too small, it will be expanded and fill will be suppressed.

You can convert a string to a number by:

`Qq+0` Convert string in Q-register `q` to a number and return it as a numeric post-comma argument.

`\q` Convert digit string in edit buffer to a number, returning it as a numeric value in Q-register `q`. Decimal and based integers are recognized.

`:\` Convert digit string in edit buffer to a number, returning it as a numeric post-comma argument. Decimal and based integers are recognized.

10.14 Conditionals

`n"x` Begin the *then* part of a conditional. The code up to the next matching apostrophe is executed if the argument `n` satisfies the condition defined by the `x`, one of the letters `A` (alphabetic), `D` (digit), `E` (equal to zero), `G` (greater than zero), `L` (less than zero), `N` (non-zero), or `U` (not lowercase letter).

`"#` Begin the *else* part of a conditional. The code up to the next matching apostrophe is executed if the condition on the *then* part was not satisfied.

`'` End the *then* or *else* part of a conditional.

10.15 Functions

`<CTL-\>` Return from a function, automatically restoring any Q-registers saved on the push-down list.

`Mq` Call the TECO function contained as a string in Q-register `q`. If the Q-register contains a number, that number is a key number, and the EPSILON function bound to that key is executed. Key numbers are just the ASCII ordinal value of the character, with 128 added for a control key, and 256 added for a meta key. Thus, `C-A` is either 1 (`<CTL-A>`) or 129, `M-A` is 321 (256 + 65), and `C-M-A` is 449 (256 + 128 + 65).

`@M/epsfun/` Call the EPSILON function `epsfun()`.

10.16 Keyboard Input

`FI` Return ordinal value of next input character from keyboard.

`:FI` Return ordinal value of next input character from keyboard, but leave the character in the input stream where it will be read again.

`FPprompt<ESC> and @FP/prompt/` Type the prompt, then read and return a string from the keyboard.

10.17 Looping

`;` Exit the innermost enclosing loop if the last search failed.

`n;` Exit the innermost enclosing loop if $n \geq 0$.

`n:;` Exit the innermost enclosing loop if $n < 0$.

`n@;` Exit the innermost enclosing loop if $n = 0$.

`n@:;` Exit the innermost enclosing loop if $n \neq 0$.

`n<` Begin a loop to be executed `n` times. The loop is not executed at all if $n \leq 0$. The loop iteration counter is not available to the program.

`<` Begin an infinite loop.

`>` End a loop. The iteration count is decremented and control is transferred to the matching loop beginning.

`@>` End a loop. Control is transferred to the matching loop beginning *without* decrementing the iteration count.

10.18 Q-registers

A Q-register is a TECO variable named by a letter or digit, preceded by 0, 1, or 2 periods. The dotted Q-registers are reserved for internal uses. Q-registers, and their push-down list, are global—functions should preserve any they use by pushing them with `[`; they will be restored automatically on return. Q-registers start out initialized to numeric zero when TECO is first loaded; they are never re-initialized by the interpreter, so they survive across an entire edit session. That makes them particularly useful for saving pieces of text that you need to insert in various places. The kill ring is not a good place to store text because it is volatile.

Q-registers can hold either strings or numbers, and the values will be coerced from the one to the other as required by context. The contents of the Q-registers are not modified by this coercion.

`%q` Increment number in Q-register `q` by 1.

`[q` Push Q-register `q` onto push-down list.

`]q` Pop Q-register `q` from push-down list.

`FQq` Return -1 if Q-register `q` contains a number, and otherwise, the length of the contained string.

`Gq` Get the string or numeric value `n` from Q-register `q`.

`:Iqstring<ESC>` Insert "`string`" in Q-register `q`.

`@:Iq/string/` Insert "`string`" in Q-register `q`.

`n:Iq` Insert ASCII character with ordinal value `n` as a *string* in Q-register `q`.

`m,n:Iq` Insert a string of `m` ASCII characters with ordinal value `n` in Q-register `q`.

`Qq` Return the contents of Q-register `q`.

`nUq` Put the string or numeric value `n` in Q-register `q`.

`nXq` Copy text in edit buffer from *point* to the `n`th following newline into Q-register `q`.

`n:Xq` Like `nXq`, but move back over newline in direction of start position before copying.

`n@Xq` Append text in edit buffer from *point* to the $\mathtt{n}^{th}$ following newline to text in Q-register `q`.

`n@:Xq` Like `nXq`, but move back over newline in direction of start position before appending.

`\q` Convert digit string in edit buffer to a number, returning it as a numeric value in Q-register `q`. Decimal and based integers are recognized.

`:\` Convert digit string in edit buffer to a number, returning it as a numeric post-comma argument. Decimal and based integers are recognized.

10.19 Quitting

`<CTL-G>` Abort and return to top-level EPSILON command loop.

`<CTL-\>` Return from function, or to EPSILON top-level if at TECO top-level already.

10.20 Quoting

`<CTL-]>` Quote the next `<ESC>` character in a TECO string, or insert the contents of the following Q-register.

10.21 Searching

A failing search is an error unless a colon modifier is given, in which case the search returns -1 on success, and 0 on failure. After a successful search, *point* is left following the matching string if the search was forward, or before it, if the search was backward. The length of the last matching string on a successful search is saved in Q-register `..L`, and its negative[11] is available with `FK`, so that *point* can be easily positioned on either side of the match. The search return value is also remembered for use by the loop exit function (`;`).

[11] positive, if the search was backwards

`nRregexp<ESC>` Regular-expression search for the $\mathtt{n}^{th}$ next match.

`n@R/regexp/` Regular-expression search for the $\mathtt{n}^{th}$ next match.

`nSstring<ESC>` Normal string search for the $\mathtt{n}^{th}$ next match.

`n@S/string/` Normal string search for the $\mathtt{n}^{th}$ next match.

10.22 Substrings

`n:Gq` Return character `n` of string in Q-register `q` as a number.

`m,nGq` Insert characters in positions `m`...`n-1` of string in Q-register `q` in buffer. If the Q-register contains a number, the arguments are ignored and the whole number string is inserted.

`m,n:Gq` Return characters in positions `m`...`n-1` of string in Q-register `q` as a string argument. If the Q-register contains a number, the arguments are ignored and the whole number string is returned.

`m,nXq` Copy characters in positions `m`...`n-1` of edit buffer into Q-register `q`.

`m,n@Xq` Append characters in positions `m`...`n-1` of edit buffer to text in Q-register `q`.

10.23 Typing Text

`n= and m,n=` Type (and discard) the arguments.

`nF(=) and m,nF(=)` Type the arguments but do not discard them. not yet implemented

`nT` Type from *point* through the $\mathtt{n}^{th}$ following newline.

`n:T` Like `nT`, but move back over newline in direction of start position before typing.

`m,nT` Type range of characters in positions $\mathtt{m} \ldots \mathtt{n} - \mathtt{1}$.

`FTstring<ESC> and @FT/string/` Type the string.

`FPprompt<ESC> and @FP/prompt/` Type the prompt, then read and return a string from the keyboard.

11 Tiny TECO

I promised in an early chapter that it is possible to learn a small part of TECO and still do useful work. Here is a stepwise guide to an ever-expanding subset of the language. You should consult the language reference section beginning on page 11 for the details and examples; the summary here is intended to be brief, so as not to overwhelm you.

A preliminary word of advice—you are going to make mistakes, and some of them will be whoppers that totally trash your edit buffer. Just two little characters, HK, will empty the buffer irrevocably. *Therefore*, before you execute a TECO program, *save* the edit buffer! If you have a disaster, a quick **revert-file** will put things right back to their state at the time of the save.

To begin with, you need to know eight basic functions:

•	Current value of *point*.
`nC`	Move *point* `n` characters forward.
`nD`	Delete `n` characters.
`Istring<ESC>`	Insert text.
`nJ`	Move *point* to absolute position `n`.
`m,nK`	Kill range of characters.
`Sstring<ESC>`	Search for a string.
`Z`	Current buffer size.

Next, you can augment this set with line-oriented commands:

`nL`	Move to beginning of `n`th line from *point*.
`nK`	Kill from point to beginning of `n`th following line.

Then you can add looping:

`n;`	Exit loop if `n` ≥ 0.
`n<`	Begin `n`-iteration loop.
`>`	End loop.

With the addition of conditionals, you have the basic requirements of any programming language (sequential, repetitive, and conditional execution):

`n"x`	Begin the *then* part of a conditional. The code up to the next matching apostrophe is executed if the argument `n` satisfies the condition defined by the `x`, one of the letters `A` (alphabetic), `D` (digit), `E` (equal to zero), `G` (greater than zero), `L` (less than zero), `N` (non-zero), or `U` (not lower-case letter).
`"#`	Begin the *else* part of a conditional. The code up to the next matching apostrophe is executed if the condition on the *then* part was not satisfied.
`'`	End the *then* or *else* part of a conditional.

The next step is to add Q-registers, so you can store results. First you need number storage and retrieval:

`nUq`	Put number `n` in Q-register `q`.
`Qq`	Get the number from Q-register `q`.

After awhile, you will probably want to store text in Q-registers too:

`:Iqstring<ESC>`	Put `string` in Q-register `q`.
`Gq`	Get `string` from Q-register `q` into edit buffer.
`m,nXq`	Copy text characters `m` + 1 . . . `n` from edit buffer into Q-register `q`.

Lastly, you might need to convert numbers to strings, and vice versa:

`n\`	Insert printed decimal representation of number `n` into edit buffer.
`\Uq`	Convert digit string in the buffer to a number stored in Q-register `q`.

You have now learned 23 function variants and should be able to do a lot with them. Since many commands allow colon and `@` modifiers, if you learn what they do for just this subset, you can probably triple your TECO repertoire.

The next time you have an editing job which does not seem to be handled by this small set that you know, you can consult the function guide section beginning on page 43. The table of contents and the index should also help you find things.

12 The Minibuffer

EMACS TECO is accessible from inside EMACS in a small buffer called the *minibuffer*, which is displayed when the function `^R Minibuffer`, normally bound to the key M-Altmode, is invoked. It appears as a small four-line window at the top of the screen, and its contents are saved on the command ring, which itself is stored in Q-vector `.N`. The minibuffer can be edited as normal text, except that `<ESC>` is made a self-inserting character, rather than bound to its usual `^R Prefix Meta` function. Two features of the EMACS minibuffer are annoying—its initial size is set at four lines, which is often too small, and it does not have a default file attached to it like a normal edit buffer, so saving its contents requires a few more keystrokes than would otherwise be necessary.

In this implementation of TECO on top of EPSILON, the minibuffer is a normal buffer named `*MINIBUFFER*`, and it can have a filename attached to it if the code is to be saved.

The minibuffer can be selected by any of the standard buffer switching commands, but as in EMACS, the command `minibuffer` is bound to M-Altmode (Alt-`<ESC>` in EPSILON terminology, since the IBM PC keyboard has a meta-key labeled *ALT*). Instead of doing a buffer switch, this enters a recursive editing mode, with the minibuffer replacing the current window. When `exit-level`, bound to `C-X C-Z`, is run, the recursive editing level is exited, and the original buffer reappears in the window.

For debugging convenience, if the last two characters in the minibuffer are `<ESC>`'s, when `minibuffer` exits, it runs `step-minibuffer` which uses the TECO debugger described in the next section to examine the program execution.

TECO editing mode is automatically selected when the minibuffer is the current buffer, or whenever a file with extension `.TEC` is visited. The only key binding change it makes is that `<ESC>` is bound to `normal-character` instead of `alt-prefix`. Since the *ALT* key is still bound to the latter, no inconvenience is incurred.

Once a TECO program in the minibuffer is ready for execution, you select the buffer which is to be the TECO edit buffer, then execute the program with `execute-minibuffer`, normally bound to `C-X <ESC>`.

The TECO interpreter checks for a user abort before executing each command, so a run-away program can always be stopped by the `abort` command bound to `C-G`.

13 The TECO Debugger

The TECO debugger is run by the command `step-minibuffer`, normally bound to `C-X <ESC>`. It just invokes the function `teco_interpreter()` with the debug step flag set. In this mode, the interpreter calls the function `teco_debugger()` before executing each command when the single step flag is set, and turns on the trace flag whenever the quiet count is non-positive.

The debugger waits for input of a single character which is interpreted before returning. If the character is a "?", it displays a help message showing the available commands:

```
HELP or H or ?: help
CTL-L:          refresh window
CTL-R:          recursive edit of minibuffer
G:              go without single stepping
N:              step silently until newline
Q:              quit
V:              view Q-register
nnn:            step next nnn functions silently
other:          continue single stepping
```

The `G` response turns off all debug tracing, and allows the program to be executed to completion, as if it had been invoked by `execute-minibuffer` in the first place.

The `N` response suppresses tracing for the rest of the current line in the program. When a newline is reached by the interpreter, tracing is restored.

The `Q` response terminates the interpreter completely, returning control to the top-level EPSILON command loop. The `abort` function on `C-G` will do the same thing.

A decimal digit string defines a quiet count which is decremented each time a TECO function is invoked; while the count is positive, debug tracing is suppressed.

The `<CTL-R>` response enters a recursive edit on the minibuffer. It does this simply by invoking the function `minibuffer`. Since this is a normal EPSILON editing mode, other buffers can be visited too, but `exit-level` should be run from the minibuffer. When exit occurs, control returns to the debugger, which then runs `save-minibuffer`, which in turn copies the possibly-changed minibuffer contents back into the interpreter string, and control returns to the interpreter, which resumes execution wherever you left *point* when the minibuffer exit occurred. This allows changes made to the minibuffer to take immediate effect, giving a run-time patch capability.

The patching has one flaw which you need to be aware of—if execution resumes in the middle of a loop, and the position of the loop beginning, `<`, has changed because of edits, then when the loop-end function, `>`, is run, it will jump back to the saved position of the `<`, which might no longer be there. The execution thread will then no longer be what you think it is. To try to catch this error, `>` checks the interpreter string and aborts if it does not find the expected `<`.

14 The TECO Implementation in Epsilon

In this section, we discuss some of the features of the implementation of TECO under EPSILON. Although some critical remarks may be made, EPSILON and EEL are products of high quality and I commend Lugaru Software for a fine job. Considering the dismal history on the IBM PC of compiler and assembler quality from numerous vendors, including IBM and MICROSOFT, it is gratifying to have a compiler for a language as complex as C which actually works!

14.1 Epsilon Support

EPSILON provides all of the underlying support for buffer, file, and window management, and runs the main command loop which interprets keyboard characters. Most EPSILON functions have been patterned after capabilities of EMACS, and most are written in EEL, a C-like language which must be compiled, but once compiled, can be dynamically loaded into the editor. Since EPSILON supports a process buffer in which one can run operating system commands, in practice, this means that one can edit a function written in EEL, save it, compile it in the process buffer, load it, and test it, all without leaving the editor. This is quite similar to EMACS, which offers a *Tcompile* function to compile a TECO program in the current region, after which it is immediately available for execution. By using the minibuffer in both editors, TECO code can be developed and interpreted immediately.

EPSILON also provides the basic editing primitives of insertion, deletion, replacement, and searching, and these are used directly by the TECO interpreter. The EPSILON model of buffer selection is unusual and not C-like, but nevertheless straightforward.

A buffer is selected for editing by assigning a buffer name pointer to the global variable `bufname`, and *point* is updated by assigning a new value to the global variable `point`. The function `size()` returns the current buffer size, and all access to the buffer is via function calls, such as `character(n)` to obtain the $\mathtt{n}^{th}$ character, or `insert(c)` and `stuff(s)` to insert single characters or strings. This implementation means that EPSILON can support paging of large files, even ones which exceed the limited address space of the IBM PC.

Updating of *point* inside these functions is pretty much the same as EMACS TECO does, so programming functions for the TECO interpreter is not fraught with mental remappings of *where are we now?*.

14.2 The Epsilon Debug Facility

EPSILON has a limited source-level debugger for EEL functions. It displays the function execution line by line, allowing stepping into, or over, function calls, suspending stepping until the function completes, changing the debug window size, and entering a recursive edit.

The EEL debugger has two serious lacks, however. The first is that there is no facility for examining, and possibly changing, variable values, and the second is that there is no call traceback display. These drawbacks seriously limit the speed at which debugging can be carried out, because one still has to go back to the source code and insert debug output statements, then recompile, reload, and re-execute it.

14.3 TECO Object Representation

One of the first considerations that must be addressed in designing a TECO interpreter is the representation of objects which could be either strings or numbers. Since these are what Q-registers hold, we shall call them Q-objects.

On the DEC-20, the machine on which EMACS was originally developed, the computer word contains 36 bits, and until the advent of extended addressing with later models of the family, a word address required 18 bits, giving a total address space of $2^{18} = 262144$ words, or about 1.1Mbytes. Characters are normally stored as 7-bit ASCII, five per word, with the left-over low-order bit set to zero. The architecture provides for efficient addressing of bytes of any size from 1 to 36 bits, and a one-word byte pointer consists of position and size fields in the high-order 18 bits, and the word address in the low-order 18 bits. A character pointer and an integer can therefore both occupy the same storage, and the particular bit patterns in the pointer give it an integer value which is near the most negative integer, $-2^{35} \equiv -34359738368$.

EMACS TECO allocates one word for each Q-object, and when necessary, determines from the high-order bits whether it is a string pointer or integer. This means, for example, that copying one Q-register into another, such as `QpUq`, can be efficiently implemented by copying a single word, without regard for its type. Arithmetic performed on a Q-object which is actually a string pointer just uses the pointer value, but the result is unlikely to be a valid pointer. No check is made in arithmetic expression evaluations for such pointer arithmetic.

The copying of Q-objects which might be pointers has an important ramification—after `QpUq`, both Q-registers point to the *same* string. If one subsequently assigns one of them a numeric value, as by `3Uq`, the interpreter cannot free the string storage pointed to by the previous contents of Q-register `q`, because there could be, and in this case, *is*, another reference to it. EMACS TECO therefore must have a *garbage collector*, just like LISP.

The INTEL iAPX architecture on which the IBM PC is based has a byte-addressable segmented memory space in which an address is a composite object consisting of a 16-bit segment, and a 16-bit offset. On the 8088, 8086, and 80186 processors, a complete address is formed by multiplying the segment value by 16 and adding to the offset, giving a 20-bit address space of $2^{20} = 1048576$ bytes. On the more powerful 80286 and 80386 processors in virtual address mode (instead of the real address mode compatible with the smaller processors), the segment is no longer a number to be added to the address, but instead an index into a table which provides a base address to which the offset portion is added. On the 80286, this constructs a 24-bit physical memory address, and on the 80386, a 32-bit address. Since all but the 80386 have only 16-bit registers, and all lack an instruction to perform arithmetic on a segment-offset pair,[12] the developers of EPSILON and EEL chose to make pointers and 32-bit integers incommensurate, and coercion between them is forbidden by the language.

The effect of this design decision is that since we cannot perform pointer to integer coercions, Q-objects cannot be represented by single 32-bit integers. Instead, they become a record structure containing a type flag, an integer, and a pointer. In EEL, this is

```
struct Q_object
```

[12]This is probably the greatest weakness of the iAPX architecture, for it takes about 30 instructions to add a constant to a pointer!

```
{
  int type; /* QT_NUMBER or QT_STRING */
  int n;    /* value if type == QT_NUMBER */
  char *s;  /* value if type == QT_STRING */
  int name; /* Q-register name (for function return restore) */
};
```

Use of a `union` could overlay the integer and pointer, but we have not done so. Throughout the interpreter, therefore, when Q-objects are manipulated, it is necessary to check the type flags to determine what code branch must be executed. Of course, much of this is encapsulated in preprocessor macro definitions to simplify the coding and hide the grubby details.

In this TECO interpreter, both Q-registers and stacks are implemented as Q-objects. This makes it simple to support looping, push and pop commands, and recursive invocation of the interpreter and EPSILON, all the while maintaining a global set of Q-registers and Q-register stack.

14.4 TECO Garbage Collection

The string assignment problem of the last section now brings us to the issue of *garbage collection*. The core of most LISP and TECO interpreters is written at a low level, usually assembly language, where the programmer has complete control over the environment and how things get implemented. Garbage collection can be designed in from the start.

With EEL, we have a C run-time environment, where standard library functions `malloc()` and `free()` can be used for dynamic memory management. However, we do not have access to the internals of this management to implement something like a garbage collector. Of course, one could initially allocate a large portion of free memory and then write private allocation and deallocation routines which could handle garbage collection, but it seems like overkill to do this.

Instead, the choice made in this TECO interpreter is that string assignment is done by making a fresh copy of the string, rather than by straight pointer assignment. We are then guaranteed that no string has more than one pointer pointing to it, and therefore, if a number is subsequently assigned to the Q-object, the string space can be deallocated without fear of a dangling pointer somewhere. Although this takes more care than would be necessary if we really did have a garbage collector, there is in fact only a small number of places where such extra bookkeeping need be done, so it is not burdensome.

One can of course regret that the standard C library does not provide for garbage collection, because careless programming can lead to growth in allocated memory. Indeed, one of the popular command shells in the UNIX system for a long time had such a bug, and would grow during a login session to the point where it finally died for lack of memory, and logged out the user. We try to be more careful.

14.5 Outline of the Interpreter

When the compiled TECO interpreter is first loaded into EPSILON, the function `when_loading()` is executed; it loads in the various compiled files that make up the interpreter, then invokes the function `teco_startup()`.[13] That function initializes the Q-registers, flags, mark ring, and function dispatch table, and then calls `teco_init()` which initializes stacks and clears the arguments. `teco_init()` is also called by `execute-minibuffer` and `step-minibuffer`. This initialization is not part of `teco_interpreter()`, so that recursive invocation of the interpreter preserves stacks, arguments, and Q-registers.

With these initializations complete, the user can run `execute-minibuffer` and `step-minibuffer`. The former is the simplest—it checks first for the existence of the minibuffer, then calls `teco_init()`, `save_minibuffer()`, and finally `teco_interpreter()`.

The intermediate function, `save_minibuffer()`, is introduced partly because of efficiency concerns. Because EPSILON permits buffer access only through function calls, it is advisable to copy the minibuffer into a string which can be accessed in-line. There is the happy side benefit that since `teco_interpreter()` operates on a string, rather than a buffer, it is easy to invoke it recursively on a new string, providing straightforward support for the TECO function-call function, `M`.

`teco_interpreter()` is the core of it all, but is surprisingly simple. It loops over the string into which `save_minibuffer()` copied the minibuffer,

[13] We use `foo()` to mean an EEL function with zero or more arguments, and `foo` for an EEL command, which is not permitted to have arguments. Although commands are otherwise normal integer functions, they rarely return values. Another distinction between them is that command names are recognized for completion by `extended-command`, while function names are not. Both can be invoked as extended commands.

using a string index which is globally available. At each iteration, it just calls `teco_eval()` to evaluate the next function. `teco_eval()` checks for a user abort, then skips over whitespace and comments (another efficiency measure), checks the debugger flags and calls `teco_debugger` if necessary, and finally, dispatches through a function table to one of up to 256 functions based on the value of the current minibuffer character.

When a binary operator, like +, is found, the dispatch calls the function `teco_binop()` which saves the current post-comma argument for the left operand and calls `teco_eval()` to evaluate the right operand. This results in a recursive entry to `teco_eval()`.

Similarly, when a left parenthesis is met, the dispatch calls `teco_leftparen()` which saves the current flags and arguments, then loops testing for a right parenthesis which terminates the loop, and otherwise calls `teco_eval()`, which again is recursive. At loop exit, the current flags and arguments are merged with the values saved on entry. Thus, parenthesized expression lists can be nested arbitrarily deep, and any number of comma-separated expressions can be present in any list.

It is important for the string to be an exact copy of what is in the minibuffer, because then the string index corresponds to *point* in the minibuffer, and viewing the minibuffer during the course of execution of the TECO program will always show the cursor tracking the execution. Even if this were not the case, it is not feasible to compact the input, because compression of white space and comments requires a complete TECO interpretation, and comments which are present to balance conditionals and loops cannot be removed anyway.

All functions called advance the global string index at least one position, except for the loop end function, which backs it up. The index movement is in general different for each function, so it cannot be handled by `teco_eval()` itself. The functions which handle a single whitespace character or comment have a loop which gobbles such text, rather than suffering the function call overhead which would be occasioned by an immediate return after handling one character or one comment.

Because of this simple structure of the interpreter, and general independence of the individual functions from each other, adding a new TECO function just involves writing the EEL code to support it, then installing it in the function dispatch table in `teco_startup()`.

14.6 Additional Support Functions

While EPSILON has most of the basic editing functionality of EMACS, aside from TECO, it lacks several functions which I have come to rely on, and have therefore added. These are not strictly part of the TECO interpreter, although those that access Q-registers obviously need some of that support.

In order to make their presence known, it is well to document them here. We use the same format as for the TECO language reference, but omit the returned values and error entries, since EPSILON commands do not normally return anything, and errors simply revert to the top-level EPSILON command loop.

`after`

Action: Prompt for a string, then display each matching line followed by the *arg* lines after the match. An omitted argument is equivalent to an argument of 0.

Relations: `before` and `occur`

Key Binding: None.

`before`

Action: Prompt for a string, then display each matching line preceded by the *arg* lines before the match. An omitted argument is equivalent to an argument of 0.

Relations: `after` and `occur`

Key Binding: None.

`blank-trim-lines`

Action: Trim trailing blanks (spaces and tabs) in the entire buffer. The number of characters deleted is printed in the echo area.

Key Binding: None.

`check-line-length`

Action: Find the next line in the buffer longer than *arg* characters and position *point* after it. An omitted argument is equivalent to an argument of 72.

Key Binding: None.

`execute-minibuffer`

Action: Execute the TECO code in the minibuffer.

Relations: `minibuffer` and `step-minibuffer`.

Key Binding: `C-X <ESC>`

`get-q-register`

Action: Get the contents of a Q-register into the edit buffer at *point*. The Q-register name is prompted for from the keyboard.

Relations: `put-q-register` and `view-q-register`

Key Binding: `C-X G`, `C-X g`

`how-many`

Action: Prompt for a string, then display a count of how many lines in the buffer (starting from *point*) contain that string. With an argument, the string is a regular-expression instead of a normal search string.

Relations: `occur`

Key Binding: None.

`insert-file`

Action: Prompt for a filename, then read the file into the current buffer at *point*. *Point* is left at the beginning of the inserted text, and mark at the end. The insertion may be undone either by killing the region, or by `undo`.

Key Binding: None.

`mark-whole-buffer`

Action: Put *mark* at the end of the buffer, and *point* at the beginning.

Key Binding: `C-X H`, `C-X h`

`minibuffer`

Action: Create a minibuffer if necessary, giving it the fixed name `*MINIBUFFER*`, select it, set TECO editing mode, then call `recursive-edit` to allow you to edit text in the buffer. On exit from the recursive edit with `exit-level` on `C-X C-Z`, it checks the last two characters in the minibuffer to see if they are `<ESC>`, which requests that `step-minibuffer` should be called before returning, then reselects the original buffer, and calls `step-minibuffer` if wanted.

Relations: `execute-minibuffer` and `step-minibuffer`

Key Binding: `A-<ESC>`

`occur`

Action: Prompt for a search string, then display *arg* lines before and after each line in the buffer which matches that string. An omitted argument is equivalent to 0.

Relations: `after`, `before`, and `how-many`

Key Binding: None.

`put-q-register`

Action: Copy the text between *mark* and *point* into a Q-register. The Q-register name is prompted for from the keyboard.

Relations: `get-q-register` and `view-q-register`

Key Binding: `C-X X`, `C-X x`

`set-pop-mark`

Action: EPSILON maintains only a single *mark*, which is inadequate. This function maintains a ring of marks unique to each edit buffer; the ring size is set at compile time to the value in the header file, `teco.h`, currently 12. Without an argument, this function sets the mark at *point*, and pushes it onto the ring. With an argument, it pops the ring into *point*. A ring instead of a stack is used so that underflow and overflow is impossible, and repeated pops will finally bring *point* back to where it started, preserving all the marks on the ring.

Key Binding: <CTL-@> (this replaces the binding to the EPSILON `set-mark` function).

`set-teco-trace`

Action: Without an argument, this function toggles the TECO trace flag. An explicit argument sets it (zero ≡ off, non-zero ≡ on). The trace flag gives additional output when TECO is single stepped.

Relations: `step-minibuffer`

Key Binding: None.

`show-buffer`

Action: Prompt for a buffer name and then display it.

Key Binding: None.

`show-columns`

Action: Type out a column number display.

Key Binding: `M-=`

`step-minibuffer`

Action: Initiate single stepping of the TECO interpretation of the minibuffer. For details, see the earlier section on the debugger.

Relations: `execute-minibuffer`, `minibuffer`, and `set-teco-trace`

Key Binding: None.

`teco-mode`

Action: Set up for editing TECO. At present, the only action is to rebind <ESC> to `normal-character`, since that character is often needed. It is invoked automatically when a file with extension `.TEC` is edited.

Key Binding: None.

`undo`

Action: Undo the last function which ran the `save-for-undo` command. The latter makes a copy of the current buffer in an undo buffer named `*UNDO*`, and `undo` exchanges the two buffers. Without an argument, it asks for confirmation. The exchange can be undone by another call to `undo`.

Unfortunately, no EPSILON functions call this one, only some of the functions described here. The convention in EMACS is that all functions which make radical changes to the buffer run `save-for-undo` before they begin. EMACS saves only the changed region of the buffer, which is fast. This version saves the entire buffer, which may be slow if the file is big. This could initiate swapping to disk, but thanks to EPSILON's design, this only forces you to wait momentarily; you cannot run out of memory.

Key Binding: None.

`view-minibuffer`

Action: This function runs `show-buffer` on the minibuffer.

Key Binding: None.

`view-q-register`

Action: Prompt for a Q-register name, then display its contents in the echo area. If it contains a long string, only the first 50 characters are shown. If it contains a number, it is displayed in decimal, octal, and hexadecimal.

Relations: `get-q-register` and `put-q-register`

Key Binding: `C-A-Q`, `C-A-q`, `C-A-Y`, `C-A-y`

14.7 Epsilon Limitations

EPSILON has several limitations which restrict the degree of closeness of this implementation of TECO to EMACS TECO. Here is a list of some of them.

- There is no bounded search function. This is a serious lack, because its simulation is necessarily inefficient.

- Commands cannot take string arguments which replace the string value they normally obtain from the keyboard. This too is a serious restriction. If each command which prompts for input then called a normal function to do the actual work, the problem could be eliminated.

- There is a function, `try_calling()`, which can be used to call a function defined by a string argument, but there is no way to test if the call was successful, or if the function even exists. There is no analogous routine to lookup the value of a global variable, or test for its existence. This makes it impossible for TECO programs to create variables dynamically, and it is impossible to implement a function like the EMACS `Edit Options` command which displays in a temporary buffer all variables matching a specified string pattern, allowing their values to be edited and modified. EPSILON is clearly capable of this, since it supports dynamic loading and binding of functions and variables; the primitives are just not available to the programmer in EEL.

- There is a source token length limit in EEL of about 250 characters, which is much too small for string tokens which can legitimately span several lines—an example is the debugger help message.

- EPSILON has no concept of a narrowed region, which is a very valuable feature of EMACS. It is used to limit the scope of editing to a portion of the buffer, such as a replacement of a string that occurs dozens of times but for which replacement must happen only in part of the buffer. The only way to do this at present is to move the region to a new buffer, perform the edits, and move the region back.

- There is no traceback display in the EEL debugger, so you cannot tell where you came from.

- It is not possible to display values of variables in the EEL debugger.

- The `say()` and `sayput()` primitives type only in the one-line echo area. This drastically limits typeout. If you want to see a larger amount of text with *more* processing (pause at end of screen), you have to put it in a temporary buffer and then run `show-buffer` on it. For some purposes, this is satisfactory, but for others, it is not. For example, `occur` can potentially display a lot of text, but in this implementation, it must find *all* of that text first before the user gets to see any of it. In EMACS, it is possible to abort the typeout at any time as the text is being found. I would very much like to see a set of EEL functions, `start_more_display()`, `more_display()`, and `end_more_display()` to handle this. The first and last would not require arguments; `more_display()` would be called with a text string to be displayed.

- Although EPSILON search and display functions are very fast, insertion and deletion are not. There is an objectionable delay if you do something like `C-U 80 *` to insert 80 asterisks in the buffer. A command like the EMACS `Flush Lines`, which deletes all lines in the buffer which contain a match with its string argument, when implemented in EEL or TECO can be *very* slow. This clearly is evidence of the EPSILON iteration count handling (the `normal-character` command gets called 80 times, instead of a string of 80 characters being created and inserted in one operation), and of the text buffer implementation, which is probably a contiguous block of text which has to be moved for every insertion or deletion. EMACS TECO uses this organization too, but maintains a gap where insertion is currently happening, so that buffer movement is minimized. The DEC-20 is fast enough that massive deletions in an edit buffer of about 1 Mbyte do not noticeably slow performance; the IBM PC is not. Several other EMACS implementations have used linked lists of lines which makes insertion and deletion fast, and allows an easy implementation of a function to undo changes of single lines. The linked list implementation does suffer a startup overhead for the list construction, which can be very noticeable on large files. I have experienced startup delays of several *minutes* on a VAX-750 with CCA EMACS on editing a multi-megabyte file.

- Richard Stallman in his article about EMACS which was cited in the introduction makes a point about the value of dynamic variable binding in addition to argument binding. He gives the example of three functions A, B, and C, where A calls B and B calls C. If A dynamically creates a new variable, C can access it without any changes being required in B or its arguments. If the variable already existed globally, or in some calling ancestor of A, the new value would hide the old one, until A exited. With EEL, we can achieve part of this—A can create a global or buffer-specific variable which B is ignorant of and C can find, but it cannot hide any previously-existing variable. The C language allows three levels of variable encapsulation—local to a function, local to all functions in a source file, and global to all functions. EEL has only the first and third of these.

It appears difficult to introduce any kind of dynamic scoping in a C-like language; indeed, dynamic scoping has been largely deprecated in COMMON LISP in favor of lexical scoping. Nevertheless, there are certainly occasions where it will be missed.

15 The TAGS Package

Although keeping a document or program in a single large file during writing and development is convenient for editing and searching, it is an invitation to disaster, particularly under an operating system which does not support multiple file generations like the DEC-20 does. However, splitting the work up into multiple files makes it hard to constantly jump back and forth between them.

The TAGS package was developed in EMACS to deal with this. It uses a special file of keywords, or tags, which additionally contains the name of the file, and if possible, the approximate position the tag is found in the file. A `goto-tag` command bound to a convenient key prompts for a tag, then searches for it in the relatively small tags file kept in a permanent edit buffer, extracts the filename, visits it in a new buffer if necessary, positions to the tag, and finally selects the buffer in the current window. This is marginally less convenient than an incremental search, since the prompt waits for a complete string, but the response can be quite rapid. Having the approximate tag position available speeds the search even further.

The tags file is created outside the editor by a separate program, which has some notion of how to deal with files containing assorted programming languages and document formatters. The DEC-20 version is written entirely in assembly language.

For EPSILON, I decided to take a different approach. Initially, I wrote a small program, `ctags.c`, which recognizes macro definitions and function headers and outputs a tags file containing the complete line for each such match, along with the file name and byte position within the file. However, it seemed wise to make this format similar to the output of the UNIX EGREP utility which is available on many systems, and for which several public domain versions have been published in personal computer magazines. EGREP outputs lines of the form

```
filename:matchingline
filename:linenumber:matchingline
```

in response to a regular expression given on the command line. For example, regular expressions that might be suitable for identifying routine headers in FORTRAN programs could be

```
^␣␣␣␣␣␣[␣]*BLOCK[␣]*DATA
^␣␣␣␣␣␣[␣]*CHARACTER[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*COMPLEX[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*DOUBLE[␣]*PRECISION[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*INTEGER[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*LOGICAL[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*PROGRAM
^␣␣␣␣␣␣[␣]*REAL[␣]*FUNCTION
^␣␣␣␣␣␣[␣]*SUBROUTINE[␣]*
```

It does not matter much if a few extra lines get matched and inserted in the tags file this way, and it is clearly much easier to write a collection of regular expressions to identify tag lines in files, rather than to write a separate program for each type. It also makes it simple for the user to add new regular expressions to pick up extra tags which might not have been envisioned by the author of the tags program.

The TAGS package for EPSILON recognizes lines of the form

```
filename:matchingline
filename:linenum:matchingline
filename:linenum:pagenum:matchingline
filename:bytenum:linenum:pagenum:matchingline
```

where none of the colon-separated fields, except the last, may contain blanks. In fact, any number of digit fields can be present; if there are three of them, like `ctags.c` outputs, then the first is assumed to be a byte position. Otherwise, they are ignored and a brute force search in made in the file.

Here is a complete list of the functions available in the TAGS package:

`tags-add`

Action: Add the current line to the `*TAGS*` buffer. The `*TAGS*` buffer does not usually have a file connected with it, so the first time you execute this command, it will prompt for a file name to be connected to the buffer. The changes are not automatically saved, but when you exit from the editor, you will be asked if you want to save the new tags file.

Key Binding: `M-+`

`tags-apropos`

Action: Display a list of all tag lines containing the string which is prompted for. Since filenames are also present on these lines, this command can be used to find what files contain a particular tag.

Key Binding: `M-,`

`tags-goto`

Action: Prompt for a tag and find it. If no tags file has been loaded into the `*TAGS*` buffer yet, then it will ring the bell and prompt for a tags file name first. If for some reason you need to change tag files, you can just kill the `*TAGS*` buffer before invoking this function. If necessary, a new buffer for the file containing a match for the tag will be created. The buffer is selected in the current window, with *point* at the beginning of the line containing the tag.

The tag name is remembered, so that if you subsequently give a null string in response to the tag prompt, the saved name will be used.

If no argument is given, the tag search will be forward from the current point in the `*TAGS*` buffer. If this search fails, it will be restarted from the beginning of the buffer, unless the search began there.

If a buffer switch does not happen, then *mark* is set where the function was invoked.

With a zero argument, the tags search is forced to start at the beginning of the `*TAGS*` buffer.

With a non-zero argument, the tags search goes backward from the current point, and if it reaches the beginning of the `*TAGS*`, continues from the end of the buffer, unless the search began there.

Key Binding: `M-•`

`tags-query-replace`

Action: Run a `query-replace` on all files named in the `*TAGS*` buffer. Each `query-replace` works only on the current buffer, so if you type a command to quit, or complete the substitutions in the remainder of the buffer, only the current file will have been processed. The original and replacement strings are remembered, so if you invoke this function again and respond to the prompt with a null string, `query-replace` will continue in the next file found in the `*TAGS*` buffer. When there are no more files left, it will ring the bell and return.

Relations: `query-replace`, `tags-search`

Key Binding: None.

`tags-search`

Action: Prompt for a search string and search for it through all the files in the tags file. The search string is remembered, so a null string entered on a second invocation will use the last string and continue the search, switching files if necessary. You can continue to use `normal-search`, usually bound to `C-S`, in the current buffer.

With an argument, use `re-search` for a regular-expression search instead.

Relations: `tags-query-replace`

Key Binding: `C-M-S`

16 Epsilon and EMACS TECO Differences

This implementation of TECO is a much smaller one than EMACS TECO. The latter needs to provide hooks for host operating system communication, file handling, memory allocation, terminal control, screen updating, and so on, since it is effectively the operating system seen by EMACS. This TECO has all of that available inside EPSILON, so it need not duplicate that functionality.

The design of EMACS TECO has been used as a guide for this TECO, but certain implementation differences have been inevitable. In most cases, they should not be noticeable. Most of the changes have been made to provide desirable enhancements to functionality. If you are already used to programming in EMACS TECO, it useful to have a summary of these differences. If you are not, then you can ignore the rest of this section.

- Two adjacent numeric values in EMACS TECO are added; this is probably a hold-over from DDT. In this TECO, the first is discarded.
- In EMACS TECO, the bitwise *exclusive or* operator is on `#`, while this TECO puts it on `^`, reserving `#` for the more useful based numbers.
- In EMACS TECO, the bitwise *or* operator is on `<CTL-A>`, while this TECO puts it on `|`.
- Labels and goto are not implemented; in more than ten thousand lines of EMACS TECO programming, I have found them unnecessary.
- In EMACS TECO, a string is represented by a number with a certain recognizable pattern in the high-order bits, which is actually a hardware string address. Arithmetic on it manipulates the address, and not the string itself. In this TECO, strings and numbers are distinct, but a string is coerced to an integer by applying the same rules used to collect an integer which were described on page 5 , and an integer is coerced to a string by converting it to its decimal representation with no leading zeros, preceded by a minus sign if it is negative.

- In EMACS TECO, `nR` is equivalent to `-nC`; it moves the position left `n` characters, and is therefore really a redundant function. In this TECO, it runs the regular-expression search function, which is much more useful.

- The search command, `S`, in EMACS TECO allows specification of multiple search targets separated by `<CTL-O>` characters in the string. This TECO does not support this, because the facility is available in a much more powerful form in the regular-expression search function, `R`.

- The EMACS TECO debug facility provides for stepping only over complete lines of commands. This TECO steps by single commands as well as by complete lines.

- EMACS TECO supports Q-registers named by `<ESC>`-delimited arbitrary string arguments given in place of the usual Q-register name in the `G`, `:I`, `M`, `Q`, `U`, and `X` commands. This TECO does not.

- Double-dotted Q-registers are reserved for predefined internal uses which are different than those of EMACS TECO. They may be read, but should never be modified.

- Empty text ranges for the `m,nK`, `m,nT`, and `m,nX` commands raise an error in EMACS TECO. They select null strings in this TECO.

Index