Emacs Lisp—Introduction

GNU Emacs is a full-featured text editor that contains a complete Lisp system that allows the user to write programs that control the editor.

Dozens of applications have been written in Emacs Lisp—they run inside the editor.

Examples:

Gnus	News reader
Dired	Directory editor
Ediff	Visual interface to diff and patch
Calendar	Calendar application that provides things like days between dates and sunrise/sunset information.

Additionally, GNU Emacs has *modes* that support editing source files for a variety of languages.

A little history[1]

Lisp:

John McCarthy is the father of Lisp.

The name Lisp comes from LISt Processing Language.

Initial ideas for Lisp were formulated in 1956-1958 and some were implemented in FLPL (FORTRAN-based List Processing Language).

The first Lisp implementation, for application to AI problems, was in 1958-1962 at MIT.

Emacs

The first Emacs was a set of macros written in 1976 by Richard Stallman on MIT's ITS (Incompatible Timesharing System) for the TECO editor. (Emacs was an acronym for Editor MACroS.)

Next, a full editor, also called Emacs, was written by Stallman in Lisp for DECSystem-10/20.

Next, James Gosling, then at CMU, developed a UNIX version in C with "Mock Lisp" as the embedded language.

Stallman wrote GNU Emacs as the first step in the GNU project.

---

[1] Don't quote me!

Running Emacs Lisp

On lectura, Emacs can be started by typing `emacs`.

This material is based on GNU Emacs 20.6.1. Use `ESC-x emacs-version` to check the version number.

A convenient way to use Emacs Lisp interactively is with `ESC-x ielm`:

```
*** Welcome to IELM ***  Type (describe-mode)
for help.
ELISP>
```

Lisp expressions

The syntax of Lisp is among the simplest of all programming languages. Function calls have this form:

```
(function expr1 expr2 ... exprN)
```

Examples:

```
ELISP> (+ 3 4)
7

ELISP> (- 9 3)
6

ELISP> (length "abcd")
4

ELISP> (concat "just" "testing")
"justtesting"

ELISP> (capitalize "this string")
"This String"

ELISP> (getenv "HOSTNAME")
"lectura.CS.Arizona.EDU"

ELISP> (dot)
210

ELISP> (dot)
227
```

Just about everything in a Lisp program is a function call.

Lisp expressions, continued

Emacs has extensive built-in documentation. For example, to see the documentation for a function, use `ESC-x describe-function`.

Examples:

`getenv`

```
getenv is a built-in function.

Return the value of environment variable VAR, as
a string.

VAR should be a string.  Value is nil if VAR is
undefined in the environment.

This function consults the variable
''process-environment'' for its value.
```

`capitalize`

```
capitalize is a built-in function.

Convert argument to capitalized form and return
that.  This means that each word's first
character is upper case and the rest is lower
case.

The argument may be a character or string.  The
result has the same type.

The argument object is not altered--the value is
a copy.
```

Lisp expressions, continued

When it makes sense for a function to have an arbitrary number of operands, Lisp typically permits it:

```
ELISP> (+ 1 2 3)
6

ELISP> (- 10 1 2 3 4 5 6 7 8 9 10)
-45

ELISP> (concat "a" "bc" "def" "ghij")
"abcdefghij"
```

Complex expressions are built up by nesting:

```
ELISP> (* (+ 3 4) (- 5 3))
14

ELISP> (substring (concat "abc" "def") 1 3)
"bc"
```

Comparisons

Comparison operations yield `t` if successful and `nil` if not:

```
ELISP> (< 1 2)
t

ELISP> (< 1 0)
nil

ELISP> (= (+ 3 4) (- 10 2 1))
t

ELISP> (string< "abc" "def")
t

ELISP> (string= "abc" "def")
nil

ELISP> (string> "abc" "def")
*** Eval error ***  Symbol's function definition
is void: string>
```

The `not` function inverts `t` and `nil`:

```
ELISP> (not t)
nil

ELISP> (not nil)
t

ELISP> (not (string< "x" "y"))
nil
```

`not` considers everything except `nil` to be `t`:

```
ELISP> (not 0)
nil
```

Variables

Lisp variable names can include many special characters but by convention, variable names are typically limited to alphanumeric characters, underscore, and hypen.

The `setq` function is used to assign a value to a variable:

```
ELISP> (setq x 1)
1

ELISP> (setq y 2)
2

ELISP> x
1

ELISP> y
2

ELISP> (+ x y)
3

ELISP> (setq login-name "whm")
"whm"
```

Note that `setq` returns the value assigned.

It is an error to request the value of an unset variable:

```
ELISP> z
*** Eval error ***  Symbol's value as variable
is void: z
```

Lists

The central element in Lisp programming is the list. Here are some examples of lists:

```
(1 2 3 4)

(x y z)

(+ 3 4)

(car ford)

'("just" a ('test) (((herre) for) example))

(cdr '(1 2 3))
```

Lists can represent program code or data; the meaning is dependent on context.

`ielm` assumes that lists are to be evaluated:

```
ELISP> (setq x (1 2 3 4))
*** Eval error ***  Invalid function: 1
```

Quoting a list suppresses evaluation:

```
ELISP> (setq x '(1 2 3 4))
(1 2 3 4)

ELISP> (setq complex '(1 2 (a b c (A B) d f) 3))
(1 2
   (a b c
      (A B)
      d f)
   3)
```

Lists, continued

Lists are thought of as having a head and tail.

The `car` function yields the head of a list:

```
ELISP> (setq x '(1 2 3 4))
(1 2 3 4)

ELISP> (car x)
1
```

The `cdr`[2] (say "could-er") function produces the tail of a list:

```
ELISP> (cdr x)
(2 3 4)

ELISP> (cdr (cdr x))
(3 4)

ELISP> (car (cdr (cdr x)))
3

ELISP> (car (cdr '(x y z)))
y
```

The `cons` function creates a list from a head and a tail:

```
ELISP> (cons 1 '(a b c))
(1 a b c)

ELISP> (cons '(a b c) '(1 2 3))
((a b c) 1 2 3)
```

---

[2] The names "car" and "cdr" are said to have originated with the inital Lisp implementation, on an IBM 7090. "CAR" stands for Contents of Address part of Register and "CDR" stands for Contents of Decrement part of Register.

Lists, continued

In Lisp, the empty list is called *nil* and can be named with `()` or `nil`:

```
ELISP> ()
nil

ELISP> '(() a ())
(nil a nil)

ELISP> (cons 1 nil)
(1)

ELISP> (cons nil nil)
(nil)

ELISP> (cdr '(1))
nil
```

Lists, continued

The `length` function returns the number of top-level elements in a list:

```
ELISP> (setq x '(4 3 2 1 0))
(4 3 2 1 0)

ELISP> (length x)
5

ELISP> (length '(a b c (1 2 3) d))
5
```

The `nth` function returns the Nth element of a list:

```
ELISP> x
(4 3 2 1 0)

ELISP> (nth (car x) x)
0
```

The `append` function concatenates lists:

```
ELISP> (append x '(a b) x '(1 2 3 4))
(4 3 2 1 0 a b 4 3 2 1 0 1 2 3 4)
```

The `reverse` function reverses lists:

```
ELISP> (reverse '(1 2 3))
(3 2 1)
```

Lists can be compared with `equal`:

```
ELISP> (equal '(1 2 3) (cons 1 '(2 3)))
t
```

Functions

The *special function* `defun` is used define functions. The general form is this:

```
(defun name arguments expr1 expr2 ... exprN)
```

The result of *`exprN`* is the return value of the function.

A function to calculate the area of a rectangle:

```
(defun area (width height)
  (* width height))
```

Usage:

```
ELISP> (setq a (area (+ 3 7) 5))
50
```

`defun` is called a special function because it doesn't evaluate all its arguments.

A function can be defined interactively in ielm but the more common thing to do is to create a file. By convention, Emacs Lisp source files have the suffix `.el`. (E-L)

A source file can be loaded with `ESC-x load-file`, or the current buffer can be evaluated with `eval-current-buffer`.

Functions, continued

Consider a function `linelen` that computes the distance between two points represented as two-element lists:

```
ELISP> (linelen '(0 0) '(1 1))
1.4142135623730951

ELISP> (linelen '(0 0) '(3 4))
5.0
```

Definition:

```
(defun linelen (p1 p2)
    (setq x1 (car p1))
    (setq y1 (car (cdr p1)))
    (setq x2 (nth 0 p2))  ; for variety...
    (setq y2 (nth 1 p2))

    (setq xdiff (- x2 x1))
    (setq ydiff (- y2 y1))
    (setq len (sqrt (+
                  (* xdiff xdiff)
                  (* ydiff ydiff))))
    len
    )
```

Problem: The `setq`s have modifed variables at the top level:

```
ELISP> x1
0

ELISP> y2
4

ELISP> xdiff
3
```

The `let` function

The special function `let` allows specification of variables that exist for a limited time.

The general form is this:

```
(let (varExpr1 varExpr2 ...) expr1 expr2 ...)
```

Each *`varExpr`* is either a variable or a list containing a variable and an initializing expression. The value of the `let` is the value of *`exprN`*.

Example:

```
(defun f (x y)
    (let ((xsq (* x x)) (y2 (+ y y)) sum)
        (setq sum (+ xsq y2))
        (format "xsq = %d, y2 = %d, sum = %d"
            xsq y2 sum)
        )
    )
```

Execution:

```
ELISP> (setq sum "old sum")
"old sum"

ELISP> (f 1 2)
"xsq = 1, y2 = 4, sum = 5"

ELISP> sum
"old sum"

ELISP> xsq
*** Eval error ***  Symbol's value as variable
is void: xsq
```

The `let` function, continued

`linelen` rewritten with `let`:

```
(defun linelen (p1 p2)
    (let ((x1 (car p1))
          (y1 (car (cdr p1)))
          (x2 (nth 0 p2))
          (y2 (nth 1 p2))
          xdiff ydiff)

         (setq xdiff (- x2 x1))
         (setq ydiff (- y2 y1))
         (sqrt (+
                (* xdiff xdiff)
                (* ydiff ydiff)))
        )
    )
```

Original version:

```
(defun linelen (p1 p2)
    (setq x1 (car p1))
    (setq y1 (car (cdr p1)))
    (setq x2 (nth 0 p2))  ; for variety...
    (setq y2 (nth 1 p2))

    (setq xdiff (- x2 x1))
    (setq ydiff (- y2 y1))
    (setq len (sqrt (+
                    (* xdiff xdiff)
                    (* ydiff ydiff))))
    len
    )
```

The `while` function *(revised)*

The special function `while` provides a means to iterate.

The general form is this:

```
(while test-expr expr1 ... exprN)
```

`test-expr` is evaluated and if it yields a non-nil value, *`expr1`* through *`exprN`* are evaluated. It continues until *`test-expr`* yields `nil`.

A loop to sum the numbers in a list:

```
(defun sumnums (L)
    (let ((sum 0))
        (while (not (equal L ()))
            (setq sum (+ sum (car L)))
            (setq L (cdr L)))
        sum)
    )
```

Usage:

```
ELISP> (sumnums '(1 2 3))
6

ELISP> (setq L '(10 20 30))
(10 20 30)

ELISP> (sumnums L)
60

ELISP> L
(10 20 30)
```

The `list` function

The `list` function is used to create a list from one or more values.

```
ELISP> (setq x "x")
"x"

ELISP> (setq y 5)
5

ELISP> (setq L (list x y x))
("x" 5 "x")
```

Contrast with quoting:

```
ELISP> (setq L2 '(x y x))
(x y x)
```

In combination:

```
ELISP> (setq L3 (list (car L) (cdr L2) 7))
("x"
 (y x)
 7)


ELISP> (list L L2 L3)
(("x" 5 "x")
 (x y x)
 ("x"
  (y x)
  7))
```

The `cond` function

The special function `cond` provides for conditional execution of expressions. The general form is this:

```
(cond clause1 clause2 ... clauseN)
```

Each clause is of the form:

```
(test-expr expr1 expr2 ... exprN)
```

Each clause is processed in turn, first evaluating *`test-expr`*. If it yields a non-nil value then *`expr1`* through *`exprN`* are executed. The value of the last expression is the value of the `cond`.

If the *`test-expr`* for a clause produces `nil`, then the next clause is evaluated in the same way.

Example:

```
(defun cond-ex1 (N)
    (cond
        ((= N 0) "N is zero")
        ((> N 100) "N > 100")
        ((= (mod N 2) 0) "N is even")
        (t "None of the above"))
    )

ELISP> (cond-ex1 10)
"N is even"
ELISP> (cond-ex1 1000)
"N > 100"
ELISP> (cond-ex1 7)
"None of the above"
```

The `cond` function, continued

Imagine a function `(divide L N)` that produces a two-element list of the number of integers in `L` that are smaller and larger than `N`, respectively.

```
ELISP> (divide '(1 2 3 4 5 6) 4)
(3 2)

ELISP> (divide nil 0)
(0 0)
```

Implementation:

```
(defun divide(L N)
    (let ((smaller 0) (bigger 0))
        (while L
            (setq elem (car L))
            (setq L (cdr L))
            (cond
                ((< elem N)
                    (setq smaller (1+ smaller)))
                ((> elem N)
                    (setq bigger (1+ bigger)))
                )
            )
        (list smaller bigger)
        )
    )
```

Problem: Modify `divide` to produce two lists of numbers.

Simple editor functions

Emacs Lisp has hundreds of functions that interact with Emacs' editing facilities in some way.

There are several editor-specific Lisp objects: buffer, window, process, keymap, marker, and more.

A *buffer* is Lisp object that holds text. Here are examples of some of the many functions that interact with buffers:

`(buffer-name)` returns the name of the current buffer:

```
ELISP> (buffer-name)
"*ielm*"
```

`(buffer-size)` returns the number of characters in the current buffer:

```
ELISP> (buffer-size)
2882
ELISP> (buffer-size)
2908
```

`(insert` *`expr1 ... exprN`*`)` inserts the values of *`expr1`* through *`exprN`* into the current buffer:

```
ELISP> (insert "just" (+ 17 15) "inserted")
nil
ELISP> just inserted
```

Simple editor functions, continued

Here is a function that returns the name and size of the current buffer:

```
(defun bufinfo ()
    (concat "The buffer is "
        (buffer-name) " and has "
        (buffer-size) " bytes"))
```

Usage:

```
ELISP> (bufinfo)
"The buffer is *ielm* and has 3305 bytes"
```

`bufinfo` is satisfactory when called in `ielm` mode but here is a version that's suitable as a general purpose Emacs command:

```
(defun bufinfo ()
    (interactive)
    (message "The buffer is %s and has %d bytes"
        (buffer-name)
        (buffer-size)))
```

The call `(interactive)` flags the function as one that can be invoked with `ESC-x`.

The `message` function creates a string, interpolating additional arguments, and shows the string in the minibuffer.

Simple editor functions, continued

The `buffer-string` function produces the current buffer contents as a string:

```
*** Welcome to IELM ***  Type (describe-mode)
for help.
ELISP> (buffer-string)
"*** Welcome to IELM ***  Type (describe-mode)
for help.\nELISP> (buffer-string)\n"
ELISP>
```

The `split-string` function splits a string. The simple mode of operation splits on whitespace:

```
ELISP> (split-string "  just  a   test ")
("just" "a" "test")
```

A function to calculate the number of words in the current buffer:

```
(defun bufwords()
    (interactive)
    (let
        ((words (split-string (buffer-string))))
        (message "%d words in buffer"
            (length words))))
```

A function can be *bound* to a key with `global-set-key`:

```
(global-set-key "\eBW" 'bufwords)
```

Typing `ESC B W` runs the `bufwords` function.

Buffer positions

The term *point* refers to the current position in a buffer. The function `point` returns the current value of point—a position in the buffer.

`(point)` ranges in value from `1` to `(buffer-size)+1`.

Point is thought of as being between characters but Emacs shows the cursor on the character following point. Example:

abcdef

If the cursor is on the "`c`", `(point)` returns 3.

The function `point-max` returns the maximum value of point in a buffer:

```
ELISP> (equal (point-max) (1+ (buffer-size)))
t
```

The function `goto-char` provides one way to change the point:

```
(goto-char 1)

(goto-char (point-max))

(goto-char 50)
```

It is not an error to call `goto-char` with an out of bounds value.

Buffer positions, continued

The functions `char-after` and `char-before` return the ASCII code for the character immediately before, or after a position.

Here is a function that steps through each character in a buffer in turn and displays, in the minibuffer, the character's position, printable representation, and ASCII code:

```
(defun step()
    (interactive)
    (goto-char 1)
    (while (not (= (point) (point-max)))
        (setq char (char-after (point)))
        (message "Char %d: '%c' (%d)"
            (point) char char)
        (sit-for 0 300) ; sleep for 0s, 300ms
        (goto-char (1+ (point)))))
```

If run on a buffer containing its own source, here are the lines that are successively displayed in the minibuffer:

```
Char 1: '(' (40)
Char 2: 'd' (100)
Char 3: 'e' (101)
...
```

Two other character-oriented motion commands:

```
(forward-char count)
(backward-char count)
```

They move the point forwards or backwards by the count specified, which can be negative.

Line-based motion

`(forward-line` *`N`*`)` moves point to the beginning of the Nth line from the current line. If N is zero, point is moved to the beginning of the current line. N may be negative; -1 is the beginning of the previous line.

If N lines do not lie ahead, point is set to `(point-max)`.

`(end-of-line)` moves point to the end of the current line.

The following function creates a list of the lines in the current buffer, minus newlines:

```
(defun buflines ()
    (goto-char 1)
    (let (first line (lines nil))
        (while (not (eobp))
            (setq first (point))
            (forward-line 1)
            (setq line
                (buffer-substring
                    first (1- (point))))
            (setq lines
                (append lines (list line)))
            )
        lines
        )
    )
```

Note that `(eobp)` returns `t` if point is at the end of the buffer.

Problem: There's a minor bug. Find and fix it.

Regions

A *marker* is a Lisp object that specifies a position in a buffer. If the text in a buffer is changed, all markers in the buffer are updated so that they stick with the text they originally marked.

Each buffer has a distinguished marker called "the mark" that is used by various user-level commands.

`set-mark-command`, bound to `^@` (control-shift-@) sets the mark.

The text between the point and the mark is known as "the region".

When invoked by the user, commands such as `kill-region` and `indent-region` operate on the text between the point and the mark.

A function to display the number of characters in the region:

```
(defun rsize ()
    (message "%d characters in region"
             (abs (- (mark) (point)))))
```

Problem: Write a function to return a list of the lines in the current region.

Changing buffer contents

Text can be inserted into a buffer at the point with the `insert` function:

```
(insert expr1 expr2 ... exprN)
```

Each expression is a string or ASCII character code.

Here is a simple function that inserts the integers from 1 through N into the current buffer, each on a separate line:

```
(defun insert-n (N)
    (let ((i 1))
        (while (<= i N)
            (insert (int-to-string i) "\n")
            (setq i (1+ i)))))
```

Changing buffer contents, continued

An ASCII code can be specified by an integer, such as 97 for `a`, but the notation *`?C`* can be used to specify the code for character *`C`*.

A function that inserts the letters from `a` to `z`:

```
(defun insert-az()
    (let ((char ?a))
        (while (<= char ?z)
            (insert char)
             (setq char (1+ char)))))
```

The `?C` notation is known as the *read syntax* for characters. Among the possibilities:

```
?\n     ?\t     ?\^d     ?\\
```

Changing buffer contents, continued

Imagine a function that "spells out" text in a region:

```
Before:
    just testing this
         M      P     (Mark and Point)

After:
    just T-E-S-T-I-N-G this
```

Implementation:

```
(defun spell-out ()
    (cond ((< (mark) (point))
       (exchange-point-and-mark)))

    (upcase-region (point) (mark))

    (while (< (point) (mark))
        (forward-char 1)
        (insert "-"))

    (delete-char -1)
    )
```

Note that `upcase-region` capitalizes all characters in a region specified by two positions.

Changing buffer contents, continued

A function to comment out the block of lines in the region:

```
(defun comment ()
    (cond ((< (mark) (point))
          (exchange-point-and-mark)))
    (let
        ;
        ; Extract the extension of the file
        ; associated with this buffer
        ;
        ((file-ext (car (reverse
            (split-string (buffer-file-name)
                          (regexp-quote "."))))))
        cmt-string)
        ;
        ; Set cmt-string based on source file type
        ;
        (setq cmt-string (cond
            ((string= file-ext "el") ";")
            ((string= file-ext "pl") "%")
            ((string= file-ext "icn") "#")
            ((string= file-ext "java") "//")))

        ;
        ; Iterate over lines in region, inserting
        ; cmt-string and a blank at the start
        ; of each line.
        ;
        (forward-line 0)
        (while (< (point) (mark))
            (insert cmt-string " ")
            (forward-line 1))
        )
    )
```

Note:

`split-string`'s second argument is a *regular expression*. The `call (regexp-quote ".")` produces a regular expression that matches a single period.

Example: `plot`

Recall our Prolog point plotter:

```
9
8
7      *
6
5
4
3   *      *
2
1
0
 0123456789
```

Invocation:

```
ELISP> (plot '((2 3) (5 7) (8 3)))
```

How can this problem be approached?

`plot`, continued

Step 1: Create a labeled 10x10 grid filled with blanks:

```
(load "helpers.el")

(defun plot (pts)
    (interactive)

    (get-empty-buffer "*plot*")

    (switch-to-buffer "*plot*")

    (let ((row ?9))

        (while (>= row ?0)
            (insert row)
            (insert-char ?  10)
            (insert "\n")
            (setq row (1- row)))
        )

    (insert " 0123456789")

    ; ... plot points ...

    )
```

`get-empty-buffer` comes from `helpers.el`.

`(insert-char` *`char count`*`)` inserts *`count`* copies of *`char`*. The call above inserts ten blanks.

`plot`, continued

Step 2: Plot the points:

```
(defun plot (pts)

    ; ... draw grid ...

    (while pts

        (let ((x (nth 0 (car pts)))
              (y (nth 1 (car pts))))

            (goto-line (- 10 y))

            (forward-char (1+ x))

            (delete-char 1)

            (insert ?*))

        (setq pts (cdr pts)))
    )
```

`(goto-line` *N*`)` sets point to the beginning of the N-th line of the buffer (1-based).

`(delete-char` *N*`)` deletes N characters following point. If negative, deletes N characters before point.

More on `interactive`

In addition to flagging a function as suitable for invocation as a command the `interactive` function can used to query the user for arguments.

Here is a modified version of `insert-n` that prompts the user for the number of lines to insert:

```
(defun insert-n (N)
    (interactive "nHow many? ")
    (let ((i 1))
        (while (<= i N)
            (insert (int-to-string i) "\n")
            (setq i (1+ i)))))
```

The first argument character, "`n`", indicates that the user should be prompted for a number read from the minibuffer.

The rest of the argument, "`How many? `", is displayed in the minibuffer.

```
ESC-x insert-n
How many? 20
```

The value typed by the user becomes the value of the argument `N`.

The above interaction is equivalent to this:

```
(insert-n 20)
```

More on `interactive`, continued

Any number of arguments may be specified with `interactive`. This version of `insert-n` prompts for both a count and a string to insert:

```
(defun insert-n (N str)
    (interactive "nHow many? \nsString? ")
    (let ((i 1))
        (while (<= i N)
            (insert str "\n")
            (setq i (1+ i)))))
```

The newline (`\n`) separates the specifiers for the two arguments.

`insert-n` may still be called as a regular Lisp function:

```
(insert-n 3 "abc")
```

More on `interactive`, continued

The specifier "`r`" (region) indicates to supply the values of point and mark, smallest first, as two numeric arguments:

```
(defun rsize (begin end)
    (interactive "r")
    (message "%d characters in region"
        (- end begin)))
```

Note that with "`r`" there is no prompt.

There are many other type specifiers for `interactive`.

Common errors

One of the two most common syntax errors in Lisp is too few parentheses:

```
(defun bufwords()
    (let
        ((words (split-string (buffer-string)))
        (message "%d words in buffer"
            (length words))))
```

Error message when loading:

```
End of file during parsing
```

The other of the two most common syntax errors in Lisp is too many parentheses:

```
(defun bufwords()
    (let
        ((words (split-string (buffer-string))))
        (message "%d words in buffer"
            (length words)))))
```

Error message when loading:

```
Invalid read syntax: ")"
```

Track the errors down by using Emacs' parentheses matching.

Common errors, continued

Parentheses may be matched overall, but misplaced:

```
(defun f (x y z)
    (let ((a 10) (b 20)
        (setq x (+ y z a)))))
```

Run-time error message:

```
'let' bindings can have only one value form:
setq, x, (+ y z a)
```

Another example:

```
(defun bufwords()
    (interactive)
    (let
        ((words (split-string (buffer-string))))
        (message "%d words in buffer")
            (length words)))
```

Run-time error message:

```
Not enough arguments for format string
```

Another:

```
(defun sumlist (L)
    (let ((sum 0))
        (while L
            (setq sum (+ sum (car L))
            (setq L (cdr L))))
        sum))
```

Error:

```
Wrong type argument: symbolp, (setq L (cdr L))
```

Common errors, continued

If a run-time error is encountered and the variable `stack-trace-on-error` is set to `t`, a stack trace is displayed in the buffer `*Backtrace*`.

The trace for the previous error:

```
message("%d words in buffer")
(let ((words ...))
   (message "%d words in buffer")
   (length words))
bufwords()
eval((bufwords))
eval-expression((bufwords) nil)
call-interactively(eval-expression)
```

Two handy `defuns`:

```
(defun tn ()
    (interactive)
    (setq stack-trace-on-error t))

(defun tf ()
    (interactive)
    (setq stack-trace-on-error nil))
```

Type `ESC-x tn` to turn tracing on; `ESC-x tf` to turn it off.

Or, set with `ESC-x set-variable` or `setq`:

```
(setq stack-trace-on-error t)
```

Debugging

A simple debugging technique is to insert calls to `message` followed by calls to `read-char`:

```
(defun f (L n s)
    (message "(f %S %d %s)" L n s) (read-char)
    (let (head)
        (setq head (car L))
        (message "head: %S" head) (read-char)
        (setq L (cdr L))
        (message "new L: %S" L) (read-char)
        )
    )
```

This causes execution to pause with the message output displayed. Pressing any key will cause `(read-char)` to return and execution to continue.

Potentially voluminous output can be directed to a buffer with `print`:

```
(defun debug-out (s)
    (print s (get-buffer "out")))

(defun f (L n s)
    (debug-out (format "(f %S %d %s)" L n s))
    (let (head)
        (setq head (car L))
        (debug-out (format "head: %S" head))
        (setq L (cdr L))
        (debug-out (format "new L: %S" L))
        )
    )
```

Emacs debuggers

Emacs has two built-in debuggers: `debug` and Edebug. The simpler of the two is `debug`.

Some basics about `debug`:

> Setting `debug-on-error` to true causes the debugger to be entered when an error is encountered.
>
> A breakpoint can be set on a function the command `debug-on-entry`. The breakpoint can be removed with `cancel-debug-on-entry`.
>
> A call to `(debug)` causes the debugger to be entered.
>
> Commands in debug mode:
>
> | | |
> |---|---|
> | `d` | Step into function call |
> | `c` | Step over function call |
> | `j` | Run current function to completion |
> | `e` | Evaluate expression |
> | `h` | Show commands |

Both `debug` and Edebug are described in the *GNU Emacs Lisp Reference Manual* in the section *Debugging Lisp Programs*.

Code as data

Recall an early `defun` example:

```
(defun f (x y)
    (let ((xsq (* x x)) (y2 (+ y y)) sum)
        (setq sum (+ xsq y2))
        (format "xsq = %d, y2 = %d, sum = %d"
            xsq y2 sum)
        )
    )
```

Once `f` has been loaded, `symbol-function` can be used to get the definition of `f`:

```
ELISP> (setq fdef (symbol-function 'f))
(lambda
  (x y)
  (let
      ((xsq
        (* x x))
       (y2
        (+ y y))
       sum)
    (setq sum
          (+ xsq y2))
    (format "xsq = %d, y2 = %d, sum = %d" xsq y2
sum)))
```

Code as data, continued

Another function:

```
(defun add (x y)
    (+ x y))
```

Definition:

```
ELISP> (symbol-function 'add)
(lambda
  (x y)
  (+ x y))
```

Once fetched, a function definition can be manipulated like any other list.

The function `rplaca` can be used to replace the car of a list:

```
ELISP> (setq L '(1 2 3))
(1 2 3)

ELISP> (rplaca L 'x)
x

ELISP> L
(x 2 3)
```

Code as data, continued

We can change the operation of `add`:

```
ELISP> (setq add_def (symbol-function 'add))
(lambda
  (x y)
  (+ x y))

ELISP> (setq L (nth 2 add_def))
(+ x y)

ELISP> (rplaca L '-)
-

ELISP> (symbol-function 'add)
(lambda
  (x y)
  (- x y))

ELISP> (add 3 4)
-1
```

Lambda expressions

By definition, a function in Lisp is simply a list whose first element is `lambda`.

```
ELISP> (defun one () 1)
one

ELISP> (symbol-function 'one)
(lambda nil 1)
```

We can create a nameless function to do the same thing and then run it:

```
ELISP> ((lambda () 1))
1
```

A function to double a number:

```
ELISP> ((lambda (x) (* x 2)) 7)
14
```

The Lisp function `mapcar` is similar to `map` in ML:

```
ELISP> (mapcar 'length '("abc" "d" "ef"))
(3 1 2)
```

One way to sort:

```
ELISP> (sort '(5 9 5 2 4 2) '<)
(2 2 4 5 5 9)
```

Another:

```
ELISP> (sort '(5 9 5 2 4 2)
             '(lambda (x y) (< x y)))
(2 2 4 5 5 9)
```