Emacs Lisp

Introduction

A little history

Introduction

GNU Emacs is a full-featured text editor that contains a complete Lisp system. Emacs Lisp is used for a variety of things:

- Complete applications such as mail and news readers, IM clients, calendars, games, and browsers of various sorts.
- Improved interfaces for applications such as `make`, `diff`, FTP, shells, and debuggers.
- Language-specific editing support.
- Management of interaction with version control systems such as CVS, Perforce, SourceSafe, and StarTeam.
- Implementation of Emacs itself—a substantial amount of Emacs is written in Emacs Lisp.

And more...

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; some were implemented in FLPL (FORTRAN-based List Processing Language).

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

There are many dialects of Lisp. Perhaps the most commonly used dialect is Common Lisp, which includes CLOS, the Common Lisp Object System.

See http://www-formal.stanford.edu/jmc/history/lisp/lisp.html for some interesting details on the early history of Lisp.

[1] Don't quote me!

A little history, continued

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.

Then, James Gosling at Carnegie-Mellon, 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, in the early 1980s.

GNU Emacs is available on most platforms.

Lisp basics

Running Emacs Lisp

Lisp expressions

Comparisons and boolean values

Variables

Lists

Functions

`let` / `while` / `cond`

Higher-order functions

Code that writes code

Running Emacs Lisp

GNU Emacs is usually named `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>
```

Use `C-X C-C` (control-X then control-C) to exit Emacs.

Lisp expressions

The syntax of Lisp is among the simplest of all programming languages. Lisp has only one type of expression—the function call. Function calls have this form:

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

Examples:

```
ELISP> (+ 3 4)
7

ELISP> (length "abcd")
4

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

ELISP> (type-of "testing")
string

ELISP> (buffer-size)
217
```

Lisp programs are primarily composed of function calls.

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> (* 1 2 3 4 5)
120

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

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

Lisp expressions, continued

Complex expressions are built up by nesting:

```
ELISP> (* (+ 3 4) (- 5 3))  ; Most languages: (3+4)*(5-3)
14

ELISP> (1+ (length "abcd"))
5

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


Comparisons and boolean values

There are a number of functions that perform comparisons. They typically return `t` if successful and `nil` if not:

```
ELISP> (< 1 2)
t

ELISP> (= (* 3 4) (+ 4 4 4))
t

ELISP> (string= "9798" (concat ?a ?b))
t

ELISP> (numberp "xyz")
nil
```

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

```
ELISP> (not nil)
t
```

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

Variables

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

`setq` is used to assign a value to a variable. It returns the value assigned.

```
ELISP> (setq sum 0)
0

ELISP> (setq new-value 10)
10

ELISP> (setq sum (+ sum new-value))
10

ELISP> (setq <x> 7)
7

ELISP> (setq x\ x "abc")
"abc"

ELISP> (setq \  (concat <x> x\ x <x>))
"7abc7"
```


Lists

The fundamental data structure in Lisp is the list. Here are some examples of lists:

```
(1 2 3 4)

(x y z)

(+ 3 4)

(car ford)

(setq y (* (dot) (dot)))

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

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

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

Lists, continued

By default, `ielm` assumes that a list is a function call 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))    ; Note: only a leading apostrophe
(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)
```

`ielm` uses indentation to show the structure of lists but lists are typically shown in a more compact form on these slides.

Lists, continued

Lisp popularized the head/tail representation of lists that is now common. 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 '(x y z)))
y
```

[2] The names "car" and "cdr" are said to have originated with the initial 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

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

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

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

ELISP> (car L)
(a b c)

ELISP> (cdr L)
(1 2 3)
```

If the second argument of `cons` is not a list, a *dotted pair* is created:

```
ELISP> (cons 1  2)
(1 . 2)

ELISP> (cdr (cons 1 2))
2
```


Lists, continued

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

```
ELISP> ()
nil

ELISP> (cons 1 nil)
(1)

ELISP> (cdr '(1))
nil

ELISP> (cons 1 (cons 2 (cons 3 (cons (+ 1 1 1 1) nil))))
(1 2 3 4)
```

Some built-in list functions

Here is a sampling of the many built-in functions that operate on lists:

```
ELISP> (length '(a b c))
3

ELISP> (nth 1 '(a b c))
b

ELISP> (member 20 '(10 20 30))
(20 30)

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

ELISP> (list '(a b) 1 2 '(10 20 30))
((a b) 1 2 (10 20 30))

ELISP> (append '(a b) 1 2 '(10 20 30))
(a b 49 50 10 20 30)

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


Functions

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

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

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

A function to calculate the area of a circle:

```
(defun area (radius)
  "Calculates the area of circle with RADIUS"
  (* pi radius radius))  ; 'pi' is a built-in variable
```

Usage:

```
ELISP> (area 5)
78.53981633974483
```

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

What would it mean to interpret the above `defun` as a plain function call?

Have we seen another "function" that in fact must be a special form?

Functions, continued

The documentation for a function can be accessed with `describe-function`:

```
ELISP> (describe-function 'area)

    area is a Lisp function in `c:/y/whm/372/el/all.el'.
    (area RADIUS)

    Calculates the area of circle with RADIUS
```

The expression `'area` creates a *symbol*. There is no trailing quote—the atom ends at the next lexical element, a parenthesis in this case. Strings, like `"area"` can often be used interchangeably with symbols although not in this case.

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`.

Functions, continued

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

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

Definition:

```
(defun linelen (p1 p2)
    (setq x1 (car p1))
    (setq y1 (cdr p1))
    (setq x2 (car p2))
    (setq y2 (cdr p2))
    (setq xdiff (- x2 x1))
    (setq ydiff (- y2 y1))

    (sqrt (+ (* xdiff xdiff) (* ydiff ydiff))) ; return value
    )
```

How would you rate the readability of the above code?

A scoping issue

An unfortunate side-effect of `linelen` is that it creates or changes variables that are visible outside of `linelen`. Example:

```
ELISP> (setq x1 "This is x1")
"This is x1"

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

ELISP> x1
0           (Because of (setq x1 (car p1)) in linelen!)
```

Java uses *static scoping* for variables. For example, the scope of an instance variable is all the methods of a class. The scope of a parameter is the body of the associated method. The scope of a variable declared in a `for` loop is that loop.

Emacs Lisp uses *dynamic scoping*. When a variable is referenced it looks for the most recently created instance of the variable and uses it. If a variable being set with `setq` doesn't exist, it is created.

`let`

The special form `let` creates variable bindings that have a limited lifetime.

Here is the general form:

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

Each *varExpr* is either a variable or a list containing a variable and an initializing expression.

The specified variables are created and initialized, possibly hiding existing variable bindings. *expr1* through *exprN* is evaluated. The value of the `let` is the value of *exprN*. When the `let` is complete, the variable bindings are erased, making any previous bindings visible again.

Here is a contrived 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)))
```

A Java analog for the `let`:

```
{ double xsq = x * x, y2 = y + y, sum; ... }
```


`let`, continued

```
(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)))
```

Note that a call to `f` doesn't change bindings visible in the caller—`sum` is unaffected, `xsq` and `y2` don't exist.

```
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

ELISP> y2
*** Eval error ***  Symbol's value as variable is void: y2
```


`let`, continued

`linelen` rewritten with `let`:

```
(defun linelen2 (p1 p2)
    (let    ((x1 (car p1))
             (y1 (cdr p1))
             (x2 (car p2))
             (y2 (cdr p2))
             (xdiff (- x2 x1))
             (ydiff (- y2 y1)))
        (sqrt (+ (* xdiff xdiff) (* ydiff ydiff))))
    )
```

while

The special form `while` provides a fairly conventional while-loop.

Here is the general form:

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

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

Here is 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))

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

Problem: Shorten the `while`'s `test-expr`.

cond

The special form `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.

```
(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"))
    )
```


cond, continued

For reference:

```
(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"))
    )
```

Usage:

```
ELISP> (cond-ex1 10)
"N is even"

ELISP> (cond-ex1 1000)
"N > 100"

ELISP> (cond-ex1 7)
"None of the above"
```


cond, continued

Imagine a function `(divide L N)` that separates the values in `L` based on whether the values are smaller or larger than `N`.

```
ELISP> (divide '(5 2 4 10 3 -3) 5)
((2 4 3 -3) (10))
```

Implementation:

```
(defun divide(L N)
    (let ((smaller nil) (bigger nil) elem)
        (while L
            (setq elem (car L))
            (setq L (cdr L))
            (cond
                ((< elem N)
                    (setq smaller (cons elem smaller)))
                ((> elem N)
                    (setq bigger (cons elem bigger)))
                )
            )
        (list (reverse smaller) (reverse bigger)))
    )
```

Higher-order functions

`mapcar` applies a function to every element in a list and produces a list of the results:

```
ELISP> (mapcar 'length '("a" "test" "of" "mapcar"))
(1 4 2 6)
```

A function in Lisp can be represented with a list whose first element is `lambda`.

```
ELISP> (mapcar '(lambda (n) (* n 2)) '(10 20 30))
(20 40 60)
```

Here is one way to write `mapcar`:

```
(defun mymapcar (f L)
    (cond
        ((consp L)
            (cons
               (apply f (car L) nil)
               (mymapcar f (cdr L))))
        (t ())))
```

A note about `apply`: The value produced by `(apply '+ 3 4 nil)` is 7.

Code that writes code

It is simple to write code that writes code. Imagine a `compose` function:

```
ELISP> (compose '(f g h))
(lambda (x) (f (g (h x))))
```

Here is `compose`:

```
(defun compose (L) (list 'lambda '(x) (buildargs L)))

(defun buildargs (L) "for '(a b c) returns '(a (b (c x)))"
    (cond
        ((= (length L) 1) (append L '(x)))
        (t (list (car L) (buildargs (cdr L))))))
```

Usage:

```
ELISP> (fset 'last (compose '(car reverse)))
(lambda (x) (car (reverse x)))

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


Code that writes code, continued

The function `symbol-function` produces the code for a function:

```
ELISP> (symbol-function 'area)
(lambda
  (radius)
  "Calculates the area of circle with RADIUS"
  (* pi radius radius))

ELISP> (symbol-function 'mymapcar)
(lambda
  (f L)
  (cond
   ((consp L)
    (cons
     (apply f
            (car L)
            nil)
     (mymapcar f
               (cdr L))))
   (t nil)))
```


Code that writes code, continued

Problem: Write (`partapp` *`function value`*) which produces a partial application.

```
ELISP> (symbol-function 'add2)
(lambda
  (a b)
  (+ a b))

ELISP> (partapp 'add2 5)
(lambda
  (b)
  (let
      ((a 5)
       (+ a b))))

ELISP> (mapcar (partapp 'add2 5) '(1 2 3))
(6 7 8)
```

Interaction with the editing subsystem

Simple editor functions

Inserting text

Movement and "the point"

Tagging a word

Line manipulation

Commenting out a block of lines

`vals.el`

Simple editor functions

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

There are several editor-specific Lisp datatypes: buffer, window, 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:

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

ELISP> (buffer-size)
107

ELISP> (buffer-string)
"*** Welcome to IELM ***  Type (describe-mode) for help.\nELISP>
(buffer-name)\n\"*ielm*\"\nELISP>  (buffer-size)\n107\nELISP>
(buffer-string)\n"

ELISP> (buffer-size)
300
```

Note that the contents of the `*ielm*` buffer include the text just typed because we're in the `*ielm*` buffer. Also note that interaction increases the size of the buffer.

Simple editor functions, continued

Here is a function that reports the name and size of the current buffer.

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

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

The `message` function creates a string, interpolating arguments like `printf` in C, and displays the string in the minibuffer, at the bottom of the screen.

Simple editor functions, continued

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

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

Here is a function that calculates 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 keystroke sequence with `global-set-key`:

```
(global-set-key "\e.w" 'bufwords)
```

Typing `ESC . w` runs the `bufwords` function.

Inserting text

The `insert` function inserts text into the current buffer at the current cursor position.

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

(global-set-key "\e.i" 'insert-n)
```

The argument to `interactive` indicates that the user should be prompted with "How many?" The number entered by the user is assigned to `N`.

Inserting text, continued

Here is a function that makes an entry in a history file. It opens the file, goes to the end, adds the users login name and a timestamp, and positions the cursor for the start of the entry.

```
(defun history-entry ()
    (interactive)
    (find-file "History")
    (end-of-buffer)
    (insert-string "\n")
    (insert-string
        (user-login-name) ", " (current-time-string) "\n\n\t"))
```

Problem: Enhance it so that if somebody is logged is as "root" it complains instead of creating an entry.

Movement and the "point"

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`.

Here is a function that produces the length of the current line:

```
(defun line-length ()
    (interactive)
    (let (start)
        (beginning-of-line)
        (setq start (point))
        (end-of-line)
        (message "Line is %d characters long"
           (- (point) start))))

(global-set-key "\e." 'line-length)
```

How does it work?

`line-length` has a needless side-effect. What is it? How can it be fixed?

Tagging a word

Below is a function that puts an HTML tag around the current word.

It uses `backward-word` and `forward-word` to locate both ends of the current word.

It special-cases the situation of being at the beginning of a word and doesn't move backwards in that case.

```
(defun tag-word (tag)
  (interactive "sTag: ")
  (cond
    ((not (= (char-before (point)) ? )) (backward-word 1)))
  (insert "<" tag ">")
  (forward-word 1)
  (insert "</" tag ">"))

(global-set-key "\e.t" 'tag-word)
```

Line manipulation

Here is a function that deletes the current line in the buffer and returns the deleted text:

```
(defun delete-line ()
    (interactive)
    (let (start line)
        (beginning-of-line)
        (setq start (point))
        (end-of-line)
        (setq line (buffer-substring start (1+ (point))))
        (delete-region start (1+ (point)))
        line))

(global-set-key "\e.d" 'delete-line)
```

Two new functions are introduced:

`(buffer-substring start end)` returns the buffer text between the positions.

`(delete-region start end)` deletes the buffer text between the positions.

Line manipulation, continued

We can use `delete-line` to create a function that lets us haul lines up and down with `Alt-Up` and `Alt-Down`.

```
(defun haul-line (move)
    (interactive)
    (let ((line (delete-line)))
        (message "Line = '%s'" line)
        (forward-line move)
        (insert line)
        (forward-line -1)))

(global-set-key [M-up] '(lambda () (interactive) (haul-line -1)))
(global-set-key [M-down] '(lambda () (interactive) (haul-line 1)))
```

Instead of binding the key sequences to intermediate functions like `haul-line-up` and `haul-line-down`, the lambda notation is used to directly call `haul-line` with a suitable value.

Commenting out a block of lines

If a block of text is swept out with the mouse that block of text is known as "the region". The position where the sweep started is produced by the `mark` function. The cursor is left where the sweep stopped and that position is produced by `point`.

Other operations can cause "the mark" to be set as well. The text between `(point)` and `(mark)` is "the region".

Let's develop some code to comment out a block of source code lines. The first piece is a simple-minded function to produce a comment-to-end-of-line string based on the extension of the file being edited in the current buffer:

```
(defun get-comment-string-for-buffer()
    (let ((file-ext (file-name-extension (buffer-file-name))))
        (cond
            ((string= file-ext "pl") "%")
            ((string= file-ext "rb") "#")
            ((string= file-ext "java") "//")
            ((string= file-ext "el") ";"))))
```


Commenting out a block, continued

Here's the function that does the commenting. It first swaps the point and mark if necessary and then proceeds a line at a time from the point to the mark and inserts the comment string at the start of each line.

```
(defun comment-lines ()
    (interactive)
    (cond ((< (mark) (point))
          (exchange-point-and-mark))) ; be sure point <= mark
    (let
        ((cmt-string (get-comment-string-for-buffer)))
        (forward-line 0)
        (while (< (point) (mark))
            (insert cmt-string " ")
            (forward-line 1))
        )
    )

(global-set-key "\e.c" 'comment-lines)
```

Larger example: `vals.el`

Imagine a function `vals` that calculates the count, sum, average, minimum, and maximum of the values contained in a rectangular region of text.

As some sample data, here are file sizes, in bytes, for some Elisp source files:

```
  49 area.el
1282 buf.el
 564 cond.el
 350 errors.el
 387 helpers.el
 167 while1.el
 401 while2.el
```


`vals`, continued

`vals`'s first step is to use `extract-rectangle` to get the rectangular region of text bounded at opposite corners by the region selected by the user. The result is a list of strings, like `("  49" "1282" " 564")`.

It then loops through the strings. If a string represents a number it is included in the computation. If not, it is quietly ignored.

```
(defun vals (begin end)
    (interactive "r")
    (let (
         (lines (extract-rectangle begin end))
         (nlines 0)
         (sum 0)
         nums snums min max avg)

        (while lines
            (setq nlines (1+ nlines))
            (setq num (read (concat (car lines) " ()")))
            (if (numberp num)
                    (setq nums (append nums (list num))))
            (setq lines (cdr lines))
            )
```


`vals`, continued

At this point the region has been fully processed. `nums` is a list of the numbers that were found. A series of calculations are performed on `nums` and the user is presented with the results.

```
            (setq N (length nums))

            (cond
                ((not (= N 0))
                    (setq nums (sort (copy-sequence nums) '<))
                    (setq min (car nums))
                    (setq max (car (reverse nums)))
                    (setq sum (apply '+ nums))
                    (setq avg (/ (float sum) N))
                    (message (concat "%d lines, %d values; min = %g, "
                                     "max = %g, sum = %g, avg = %g")
                        nlines N min max sum avg))
                (t (message "%d lines, no numbers" nlines)))
            )
        )

    (global-set-key "\eV" 'vals)
```