

Emacs Lisp

Table of Contents

- [Emacs Lisp](#)
 - [Table of Contents](#)
 - [Lisp](#)
 - [Namespace and Security](#)
 - [The Language](#)
 - [Fundamental Data Types](#)
 - [Fundamental Data Structures](#)
 - [Variables](#)
 - [Common Functions](#)
 - [Control Flow](#)
 - [Special Forms](#)
 - [Functions](#)
 - [Emacs Byte-code](#)
 - [The Emacs Interpreter](#)
 - [Customizing Key Binds](#)
 - [Elisp within Emacs](#)
 - [Exercises](#)
 - [Customizing Emacs](#)
 - [Example Customization](#)
 - [ASIDE: Concept of Pure Functions](#)

Lisp

- **LISP**: LISt processor because the source code is comprised of lists.
- Lisp is the first **functional language**.
 - **ELisp** (Emacs Lisp) is a variant of Lisp used to write and extend the Emacs application.
 - Elisp is a language embedded as a part of larger applications (extension language)

An example of **app-specific language (ASL)**. To some extent, it's an example of the *little language philosophy*. The creators of Emacs took an existing language and mutated it to suit their particular problem.

Namespace and Security

- Emacs is very traditional - a flat, top-level namespace that can be prone to clashing
- You have a lot of liberty in making problematic changes. This is allowed for example:

```
emacs-version
; "28.2"
(setq emacs-version "100.0")
; "100.0"
(setq emacs-version "100.0")

; You can also mess around safely with scoped variables:
(let ((emacs-version "19.2"))
  (message "A %s" emacs-version)
)
; "A 19.2"
```

- Trying to evaluate a variable that's never been assigned, you get a *runtime error*, very reliably

- There is *well-defined behavior*, as opposed to *undefined behavior* like in C/C++, the former which is obviously *safer*
- The Emacs debugger also provides a traceback, but it reads bottom to top
 - To exit from the debugger, enter C-]

Scripting languages thus tend to be used for exploratory programs, student codes, one-off programs, etc.

```
(message "You can %s like printf!" "interpolate")
; "You can interpolate like printf!"
```

The Language

Fundamental Data Types

- 34 **integers**
- 34.4 **floats**
- 'eggert' **symbol** has a name, in this case eggert and has no other properties
 - a symbol can only has one name, which guarantees uniqueness via the symbol table
- "eggert" **strings** - do not care about line boundaries
 - strings with equal content are not unique

Fundamental Data Structures

Emacs uses the same notation for programs and data. In other words, we use **data notation** to write our programs. The fundamental data structure in Lisp is the **list**, which is built from **cons**, which is just a pair of values. A list is singly-linked list of cons.

- (cons a b) builds a pair and gives you the result (allocates storage on the heap)
 - [A|B] is what it looks like (implementation)

Lists

- (A B C D E) five-item list, which are built out of pairs
 - [A|]->[B|]->[C|]->[D|]->[E|/] how the list implementation looks like
- (A . B) is a list without the null character at the end of it, (does not chain)
- (A B C . D) = [A|]->[B|]->[C|D]
- () list of no elements (empty list)
- nil also the empty list
- ("A ()") list of 1 element: the string "A"
- (A ()) list of 2 elements: A and empty list
- ((A B C)) list of 1 element: a list with three elements

Specifying Data Types

- We use an apostrophe to treat something as data '(f e g) tells lisp to treat that as a list
- (f e g) would be a function call
- this is useful because lists can reprogram themselves

```
(let ((a '(f e g)) # let a be list(feg)
      (b 29))      # let b be 29
(cons a b))        # execute function cons on a and b
```

- (A (B C) D) = [A |]->[[B |]->[C |\]]]->[D|\] (nested list)

- `(let ((foo (cons 29 '(37 -6)))) = [29 |]->[37 |]->[6 |]`
 - `(cons (cdr foo)(car foo)) = [[37 | -> [-6 |]] | 29]`

Cons Nested List Example

```
(cons Hello (Brave New) World)
```

Variables

- `(defvar x 5)` `(defvar y 5)` define and set a variable
- `(setq x 5)` `(setq y 5)` Set the value of an existing variable

```
(let ((a 1) (b 2)) ; local binding within the body
  (+ a b)
)
```

Common Functions

- `quote` returns object, without evaluating it

```
(quote (+ 2 2))
'+ 2 2)
```

- `car` returns the first element in a list

```
(car '(rose violet daisy buttercup))
; rose
```

- `cdr` returns the rest of the list

```
(cdr '(rose violet daisy buttercup))
; (violet daisy buttercup)
```

- `cons` constructs lists

```
(cons 'I '(like lisp))
; (I like lisp)
(cons (car '(rose violet daisy buttercup)) (cdr '(rose violet daisy buttercup)))
; (rose violet daisy buttercup)
```

- `append` attaches one list to another

```
(append '(1 2 3 4) '(5 6 7 8))
; (1 2 3 4 5 6 7 8)
(cons '(1 2 3 4) '(5 6 7 8))
; ((1 2 3 4) 5 6 7 8)
```

- `(forward-char x)` moves x characters forward at cursor position

- (other-buffer)
- (current-buffer)
- (point-min) function with 0 arguments
- (point-max)
- (count-lines beginning end)

Control Flow

- (= 1 2) comparison operator (bool return)
- (> 2 2) comparison operator (bool return)
- (if (= 1 2)) "Yes" runs if true, "No" runs if false

Special Forms

- (if 0 (cons 3 5)(cons 4 7)) if 0 is nil evaluate (cons 3 5) else evaluate (cons 4 7)
 - useful as it is cheaper than a function call
- (let((arg1 arg2) (arg3 arg4)) command) goes through each pair until non-nil, and returns the corresponding item to the command
- (and arg1 arg2) evaluates from left to right, if one of the values evaluates to nil, it will return nil. Otherwise returns last item
- (or arg1 arg2) finds the first non-nil argument and returns that value

Functions

```
(defun function-name (arguments...)
  "optional-documentation..."
  (interactive argument-passing-info) ; optional
  ; body
)
```

Example:

```
(defun multiply-by-thirty-five (number)
  "Multiply NUMBER by Thirty Five."
  (* 35 number)
)
```

```
; calling the function
(multiply-by-thirty-five 2) ; 70
```

- ; you can define a function to be interactive
- ; such functions can be called via M-x
- (defun add (x y) (interactive) (+ x y))

Emacs Byte-code

- You can load external source code into the current namespace with:
 - `M-x load-file RET filename RET`
- As a solution to slow interpreting speed (due to the extensive use of pointers and dereferencing), ELisp uses **byte-codes** to compile data structures to create compact representations of a program.
- Byte-code differs from machine code:
 - **PRO:** byte-code is *portable* and works on any architecture as it is designed for some abstract machine that the Emacs application knows about.
 - **CON:** Not as performant as machine code.

Emacs Lisp has a compiler that translates functions written in Lisp into a special representation called byte-code that can be executed more efficiently. The compiler replaces Lisp function definitions with byte-code. When a byte-code function is called, its definition is evaluated by the byte-code interpreter.

- The numbers are analogous to opcodes in true machine code
 - Each number represents a certain elementary operation, like pushing data onto stack memory, adding two values, etc

For example, abstractly, this may be the compiled byte-code for some function:

```
1 push a (arg #1)
10 dup
27 *
2 push b (arg #2)
10 dup
27 *
26 +
105 sqrt
```

- Strung together they are functionally equivalent to the original Emacs function from which it was compiled, but now it can be compactly represented with a byte stream `1 10 27 2 10 27 26 105 .`
- This in turn is more performant than uncompiled Emacs code because it can be run directly by a byte-code interpreter, in contrast to high-level language that needs to be parsed (tokenized and semantically analyzed) before executing - going off of some scattered knowledge here, anyone feel free to correct me.
- The byte-code files end with the `.elc` extension. You can compile a `.el` *source* file with:

```
M-x byte-compile-file RET filename.elc RET
```

`.elc` files can be loaded in the same way as `.el` files with `load-file`.

Generally, you'd want to keep both the `.el` (so you can make future changes) and the `.elc` (so you get a performance gain) files on hand. But we're also talking about scripting in this class, and with the sizes of our scripts, this compiled/uncompiled speed difference is not noticeable (so bothering to make a `.elc` file in the first place is.. questionable. But it's good to know its purpose and that it's how Lisp works under the hood). - **Nik Brandt (Piazza)**

The Emacs Interpreter

- Is a *single-threaded interpreter*
- There is only one instruction pointer (e.g. `%rip` on x86-64) at all times
- There's also a byte-code interpreter, which has some byte-code instruction pointer
- Emacs is itself a C program, so under the hood it's something like:

```
// points to current instruction in byte-code sequence
unsigned char *bcip;
```

Customizing Key Binds

```
(global-set-key "@" "abcxyz")
; Now typing "@" is automatically replaced with "abcxyz"

; This is like setting keyboard macros:
(global-set-key "@", 'what-cursor-position)
; Now typing "@" automatically shows cursor position in minibuffer
```

- This could be a security hazard, because you can also include control characters. Emacs does provide an easy way to write control characters as strings:

```
"\C-k" ; C-j
"^\K"
```

- Emacs however checks if you attempt to recurse like:

```
(global-set-key "!" "!")
; After 0 kbd macro iterations: Lisp nesting exceeds 'max-lisp\
; -eval-depth'
```

Elisp within Emacs

- You can use `M-: RET` to enter an Eval minibuffer.
- You can also eval a line in a main buffer by moving the cursor to the end of the Lisp list and then entering `C-x C-e`.
- You can directly enter an initial buffer without a file called the `*scratch*` buffer. You do this by pressing `q` right after starting up Emacs.

Lisp interaction mode:

- Move cursor to end of Lisp expression line.
- `C-j` to evaluate this line.
- Output will be shown and written to the buffer.
- Use instead of `C-x C-e` if you want the history to be saved in the buffer itself.

Exercises

```
quote (1 2 3)           ; (1 2 3)
'(1 2 3)                ; (1 2 3)
(list (+ 1 2) '(+ 1 2)) ; (3 (+ 1 2))
(cons (+ 1 2) '(3 4))   ; (3 3 4)
(+ 10 (car '(1 2 3)))   ; 11
(append '(1 2) '(3 4))  ; (1 2 3 4)
(reverse (append '(1 2) '(3 4))) ; (4 3 2 1)
(cdddar (1 2 3 4 5 6 7)) ; ERROR (unless cdddar defined)
```

Customizing Emacs

Some Emacs jargon:

Term	Meaning

Point Term	Current position of the cursor Meaning
Mark	Another position in the buffer
Region	Text between the mark and point

The function `point` returns the current position of the cursor as a number.

`save-excursion` is often used to keep point in the location expected by the user:

```
(save-excursion
  (actions-that-modify-point))
```

TIP: You can use `(global-set-key KEY COMMAND)` to define a custom key bind.

Example Customization

Explain how to arrange for Emacs to treat C-t as a command that causes Emacs to issue a message like this in the echo area:

1. Define a function.
2. Use `current-time-string` to get current time, and use `concat` to concatenate time with other text
3. Make the function `interactive`
4. Use `message` for output in the echo area

```
(defun print-time ()
  (interactive)
  (message (concat "It is now" (current-time-string) "."))))
```

5. Create the C-t key binding:

```
M-x global-set-key C-t print-time RET
```

ASIDE: Concept of Pure Functions

Also known as **deterministic**, functions that have two properties:

1. Given a specific input `x`, the function *always* returns the same output `y`.
2. It doesn't modify any data beyond initializing local variables required to compute its output.

```
// Pure
int f(int p) {
  int q = 5 * p * p;
  return q;
}

// Impure
int z;
int f(int p) {
  return p * z++;
}
```

Notice that as long as you're using or modifying a global mutable variable, your function risks not being a pure function.

Functional languages...

- Are only composed of functions
- Can't change variable state (no $p = p + 1$).
- No loops, only recursion (because the counter i changes its state).
- Order of execution is not important because all functions are pure, so they won't have any *side effects* by definition.