eLisp (Emacs Lisp)

Elisp is the scripting language that powers Emacs, which is used to write extensions and customize Emacs. It is also a dialect (方言) of the Lisp programming language.

`;` - comment notation in eLisp
`nil` - represents the boolean value `false`, the empty list `()`, and the end of a list. Can only be compared via `eq` and `equal`, but not `=`.

- Scratch
  Open 'Scratch' to code with eLisp
  `C-x b` RET `*scratch*`
  Evaluate(run) eLisp code
  `C-j` or `C-x C-e`

- .el file (eLisp file)
  Load the file by `M-x load-file` RET `file_name` in Emacs
  Run the file by `M-: function_name` in Emacs
  Run the file by `M-x function_name` in Emacs (while interactive enables)

Basics Grammar

```
(message "Hello, world!")    ; print("Hello, world!")
(setq i 0)                   ; i = 0 (assign value)
(+ 2 3)                      ; 2 + 3 (addition)
(= 2 3)                      ; 2 == 3 (whether equal to)
```

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.

```
# This is a cons(pair)
[A|B]
(A . B)

# This is a list
[A| ]->[B| ]->[C| ]->[D| ]->[E|/]
(A B C D E)

# You can use . to write at the cons level
[A| ]->[B| ]->[C|D]
(A B C . D)

# The empty list; also used as the null terminator
```

```
()

# Thus this is equivalent to (A B C D E)
(A B C D E . ())

# Nested lists
[A| ]->[ | ]->[D|/]
        v
       [B| ]->[C|/]
(A (B C) D)
```

In Emacs, the *empty list* is like the *null pointer* - its byte representation is all 0s as well.

Emacs Byte-code

When Elisp code is compiled, the code is translated into a set of byte-code instructions (aks **Emacs Byte-code**) that are understood by the Emacs Lisp virtual machine.
This byte-code is not as fast as native **machine code**, but it's significantly faster to execute than interpreting the Elisp source code directly.

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*(可移植的) and works on any architecture as it is designed for some abstract machine that the Emacs application knows about.
- **CON:** not as performant or fast as machine code.

From the GNU documentation:

> 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(operating codes) 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)**

Objects

Number

eLisp does not have a fixed limit on the size of integers, which means it can be worked with very large numbers **without overflow**.

String

Escape Sequences:

- `\n` - New Line
  `(setq myString "First line\nSecond line")`
- `\t` - Tab
  `(setq myString "Column1\tColumn2")`
- `\"` - Double Quote
  `(setq myString "She said, \"Hello\" to me.")`
- `\\` - Backslash
  `(setq myString "This is a backslash: \\")`

Symbol

Pair (Cons)

Remember each **cons** or **pair** requires two arguments.
You can use **cons** to create a **list** with the `nil` as an ending. (check back the part of "Fundamental Data Structures" above)

```
(cons 12 “abc”)     ; output: (12 . “abc”)
```

```
(cons -3 (cons 12 nil))     ; output: (-3 12)
; with a nil, a list is created but NOT a cons.

(cons -3 (cons 12))         ; error
; 'cons' function requires two arguments, but the second call to cons is
only provided with one
```

Eval (Evalution)

Evaluation follows a specific set of rules that dictate **how expressions(code) are interpreted and executed** by the Lisp environment
Elisp allows programmer to **prevent evaluation** by using two prefixes: **Single Quote(`'`)** or **Backtick(`` ` ``)**.

```
(setq l `(a b c))       ; result: set l = (a b c)
(setq l (a b c))        ; error
; (setq variable value)
; Without the quote, eLisp treats 'a' as a function with 'b' and 'c' as
arguments
; Error will appear if 'a' is not a function or if 'a' cannot accept 'b'
and 'c' as valid arguments
```

```
(setq m `(x y z w))     ; result: set m = (x y z w)
(setq p (append l m))   ; result: set p = l + m = (a b c x y z w)
```

```
(cons 3 l)              ; result: (3 a b c)
; since `(a b c) is not been evaluated, then it is stored as a list in l
; the result is (3 a b c), a list, but not (3 . (a b c)), a cons, since l
is a list
(cons 3 `l)             ; result: (3 . l)
; since `l is not been evaluated, then `l is only a letter l
```

```
(setq x `(cons 3 l))    ; result: x = (cons 3 l)
(setq y (cons 3 l))     ; result: y = (3 a b c)
```

```
(eval (+ 1 2))          ; return: 3
(eval `(+ 1 2))         ; return: 3
```

```
(1 2 3)                 ; result: error
; '1' is treated as the function name with '2' and '3' as its arguments
`(1 2 3)                ; return (a b c)
```

Control Flow

if-statement

```
; if-statement template
(if condition
    then-part
  else-part)
; then-part: The code to execute if condition is true.
; else-part: The code to execute if condition is false.
```

```
(setq number 5)

(if (> number 3)
    (message "Number is greater than 3")
  (message "Number is not greater than 3"))
```

cond

A conditional construct similars to 'switch' in C++/Java

```
; cond template
(cond
  (condition1 body1...)
  (condition2 body2...)
  ...
  (t default-body...))
```

```
(setq number 7)

(cond
 ((< number 5) (message "Number is less than 5"))
 ((= number 5) (message "Number is equal to 5"))
 ((< number 10) (message "Number is less than 10 but greater than 5"))
 (t (message "Number is 10 or greater")))
```

For-loop

In eLisp, the typical for-loop as seen in many other languages is implemented using `dolist` or `dotimes`

- `dolist` iterates over each element in a list

  ```
  ; dolist template
  (dolist (var list)
    body...)
  ; var: the loop variable.
  ; list: the list to iterate over.
  ```

  ```
  (dolist (element `(1 2 3 4 5))
    (message "Element: %d" element))
  ```

- `dotimes` iterates a specified number of times

  ```
  ; dotimes template
  (dotimes (var count)
    body...)
  ; var is the loop variable (usually an integer)
  ; count is the number of iterations
  ```

  ```
  (dotimes (i 5) ; similars to (i = 0; i < 5; i++)
    (message "Iteration: %d" i))
  ; output: 0 1 2 3 4
  ```

While-loop

```
; while-loop template
(while condition
  body...)
```

```
(setq counter 0)

(while (< counter 5)
  (message "Counter: %d" counter)
  (setq counter (+ counter 1)))
```

Functions

1. Declaration and Definition

- `(interactive)`
  '(Interactive)' identifier is optional. It makes the function callable via M-x in Emacs. If your function is intended to be used interactively (like a command), you should include this. Call it with `M-x function_name`.

- Function Definition Template

```
(defun function-name (parameter1 parameter2 ...)
  "Optional documentation string."
  (interactive)
  ;; Function body
)
```

- Example 1 - Regular Function without `interactive`

```
(defun is-even (n)
  (= (% n 2) 0)
  ; (if (= (% n 2) 0) t nil)
)
```

- Example 2 - `interactive` Function

```
(defun is-even (n)
  "Check if N is an even number."
  (interactive "nEnter a number: ")
  ; n to Prompts the user to enter a number
  ; s to Prompts the user to enter a string

  (if (equal (% n 2) 0)
    (message "%d is even" n)
    (message "%d is odd" n)
  )
)
```

2. Calling a Function

- `%d` - A format specifier (placeholder) for an integer. It expects an integer_value and places it in the string at the location of %d.
- `%s` - A format specifier (placeholder) for a string. It expects a string_value and places it in the string at the location of %s.

```
(add-two-numbers 3 5)
; this line will return 3 + 5 = 8

(message “Hello world %d %s!” 34 “xyz”)
; Output: Hello world 34 xyz!
; In this example, %d is replaced by 34, %s is replaced by "xyz".
```

3. Common Build-in Functions

- Message (print)

  ```
  (message "This is a message: %s" "Hello World")
  ```

- String Manipulation

  ```
  (concat "Hello, " "world!") ; Concatenates strings.
  (substring "Hello, world!" 0 5) ; Extracts a substring [0,5). Output:
  "Hello"
  ```

- List manipulation

  ```
  (list (+ 1 2) `(+ 1 2)) ; Returns (3 (+ 1 2))
  (cons 1 (cons 2 nil)) ; Returns (1 2), which is a list

  (car `(a b c)) ; Returns: a
  ; Returns the first element of the list.

  (cdr `(a b c)) ; Returns (b c)
  ; One `d` means to return the rest of the list after first element.

  (cdddr `(1 2 3 4 5)) ; Returns (4 5)
  ; ‘cdddr’ is a function that gets the third cdr of a list. It's
  equivalent to calling cdr three times in succession.

  (cadddr `(1 2 3 4 5)) ; Returns 4
  ; ‘cadddr’, a represents the head char, d stands for delete

  ; !!! E R R O R !!!
  ```

```
(car `((1 2 3) (4 5 6))) & (car `(1 2 3) `(4 5 6))
; car only can process one list

(append `(a b) `(c d)) ; Concatenates two lists to form (a b c d).
```

- Math Functions

  ```
  (+ 1 2 3) ; Adds numbers.
  (* 4 5) ; Multiplies numbers.
  (sqrt 16) ; Calculates the square root.
  (expt 2 3) ; Calculates the power, 2^3
  ```

- Comparison

  `=` - Numerical Comparison
  Numerical Comparison is **ONLY** used to compare numerical numbers. It **CANNOT** be used to compare non-numeric values such as *`list`* *or* *`nil`*

  `equal` - Content Comparison

  `eq` - Identity Comparison

  p.s. When the `equal` and `eq` are used as Numerical Comparison, they check both operands and values.

  ```
  (= 3 3.0) ; t (stands for `true`)

  (setq a (cons 3 19))
  (setq b (cons 3 19))
  (equal a b) ; t

  (eq a b) ; nil (stands for `false`)

  (= `(a b c) nil) ; !!ERROR!! (wrong-type argument)
  (equal `() nil) ; t
  (eq `() nil) ; t

  (equal 3 3.0) ; nil (since 3 is integer, and 3.0 is floating-point
  number)
  (equal 3 3) ; t
  (eq 3 3.0) ; nil (since 3 is integer, and 3.0 is floating-point
  number)
  (eq 3 3) ; t
  ```

- Some others

```
(setq x 10)
; ‘setq’ is used to set the value of a variable (x) to 10.
(quote (1 2 3)) & (`(1 2 3))
; This results in the list (1 2 3) without attempting to evaluate it.
It's often abbreviated as '(1 2 3).
(reverse `(1 2 3)) # Returns (3 2 1)
; The single quote (') in (reverse '(1 2 3)) in eLisp is used to
prevent the list (1 2 3) from being evaluated.
```

> **Compare the usage of Single Quote(`'`) and Backtick(`` ` ``) in eLisp**
> While both the Backtick and the Single Quote **prevent evaluation**, the *Backtick* provides additional flexibility by allowing certain parts of the expression **to be evaluated**.
>
> Single Quote
> `'(+ 1 (+ 1 2)`
> `; Output: (+ 1 (+ 1 2))`
>
> Backtick
> The backtick(`) is used for backquoting, which allows portions of the quoted expression **to be evaluated** by prefixing them with a comma(`,`)
> `` `(+ 1 ,(+ 1 2)) ``
> `; Output: (+ 1 3)`