Common Lisp - The Tutorial Part 6

Globals and Lists

Wax on, right hand. Wax off, left hand. Wax on, wax off.
Breathe… in through nose, out the mouth.
Wax on, wax off.
Don't forget to breathe. Very important.
Wax on, wax off. Wax on, wax off.
– Mr. Miyagi

[See more at https://github.com/rabbibotton/clog/blob/main/LEARN.md](https://github.com/rabbibotton/clog/blob/main/LEARN.md)

Introduction

Lisp is named Lisp for "**Lis**t **P**rocessing" and we have seen the main input to the compiler are lists in a form, well, called the **form**, (operator argument1 argument2…). Like in Karate, forms, well practiced sets of moves, gives you power in a fight to know how to respond with well practiced attacks and defenses, the use of the **form** in Lisp gives you power to manipulate code as data and data as code[1], homoiconicity, a power other languages can only dream of, and you shall taste[2] of it in this tutorial.

The **EVIL** Globals

Before we go further, I am compelled to introduce you to three top-level-forms that sometimes are the source of EVIL and sometimes they LIVE for a greater purpose… "Dynamic Scope", a cool and bizarre twist of history that changes EVIL => LIVE, born of history and fairy magic, and for another tutorial, is one such good purpose. We shall reveal others now.

```
(defvar *symbol-for-global-variable* "optional initial value"
 "I LOVE DOC strings")
```

The convention for global variables and parameters is to place earmuffs around the variable name, ***my-var*.** This is to let everyone know "I have evil potential". Not everyone agrees with this convention but I suggested "Until you fully know both sides of an argument, don't form an opinion." (Wise Words from a Wise Guy - me).

[1] The spice is the worm, the worm is the spice.
[2] It tastes like chicken. (All code does)

What is the "evil potential" of a global variable? It is, that it is global. You can read and change the value of the variable from any function or even the REPL of the Lisp image at any time and even concurrently[3] and that may affect any use of that variable silently and with no notice. Globals have power as well. In CLOG's second demo (clog:run-demo 2) a chat app, a global hash[4], allows all the users to interact with each other in just a few lines of code.

Since Lisp is image based, not file -> compile -> run based, there is something unique about these global variables. If I tell the compiler to compile the defvar with a new "optional initial value" it won't change the value that already exists for the global variable, that option is only set the first time it is evaluated into our Lisp image. In others M-C-x has no real function on a defvar after the first time (the DOC string will change if modified[5] :).

```
(defparameter *symbol-for-global-parameter* "value"
 "I LOVE DOC strings")
```

A parameter is a global variable that every time you recompile the definition it is bound again to "value". In many ways this is the purpose of a constant in most languages, a value is not usually changed by the application. The nature of Lisp is that you may want to make changes live on a setting and so you can.

```
(defconstant symbol-for-global-constant "initial value"
 "I LOVE DOC strings")
```

A constant never changes and the compiler will raise an error if you try. In most cases beyond giving a name to a numerical number this is unlikely the best choice and should use defparameter.

List Power

Lists of data are entered with the `list` operator:

```
(list 1 2 3 4)
=> (1 2 3 4)
```

[=> means evaluates to (i.e. what would be in memory)]

The form returns a list object containing 4 elements that are numbers, Lists can contain any object:

---

[3] A tutorial on currency is for sure coming.
[4] A hash is like a list where each element has a name, we will get there.
[5] In the future we will do a tutorial on introspection tools as the doc strings are accessible to your code too.

```
(list 1 #\a "2" 'symbol-name #'print)
   =>(1 #\a "2" SYMBOL-NAME #<FUNCTION PRINT>)
```

[The REPL can't print what a function looks like so it uses a #<> notation to represent it.]

This form returns a list with five elements, each a different type, a number, a character, a string, a symbol, and a function.

There are a HUGE number of operators for operating on Lists, some even have duplicate names for the same functionality. Yes, now you know it too, the ugly truth of Common Lisp… with age and wisdom comes skin wrinkles. Common Lisp is a mix breed of many many previous Lisps (some with names as ridiculous as **nil**) and it picked up along the way slang words (only some are four letters). No one is forcing you to use that slang, but it is not hard to figure out what someone says, you just use: http://l1sp.org the Lisp "the search engine" lookup or can ask the REPL with `(describe 'symbol-to-describe)`. Another good resource is https://jtra.cz/stuff/lisp/sclr/index.html that lists commonly used parts of Common Lisp.

We will use the REPL for now to demonstrate a few to get started. First let's set up a list in a global variable to work with:

```
(defvar *mylist* (list 1 2 3 4 5))
```

The first element:

```
(car *mylist*)
=> 1
```

or

```
(first *mylist*)
=> 1
```

Get the rest of the list after the first:

```
(cdr *mylist*)
=> (2 3 4 5)

(rest *mylist*)
=> (2 3 4 5)
```

Get the second, third, fourth, and fifth:

```
(second *mylist*)
=> 2
```

```
(third *mylist*)
=> 3
(fourth *mylist*)
=> 4
(fifth *mylist*)
=> 5
```

Get the n'th element:

```
(nth 3 *mylist*)
=> 4 ; Notice 4 - nth starts at 0
```

Append two lists:

```
(append *mylist* (list 1 2 3))
=> (1 2 3 4 5 1 2 3)
```

Reverse list:

```
(reverse *mylist*)
=> (5 4 3 2 1)
```

You will notice that *mylist* was not modified by any of the above operators, i.e. there are no side effects.

There are a few operators with side effects worth knowing now:

Push an element to start of list:

```
(push 99 *mylist*)
=> (99 1 2 3 4 5)

*mylist*
=> (99 1 2 3 4 5)
```

And as a side effect *mylist* was changed.

Pop an element off of list:

```
(pop *mylist*)
=> 99

*mylist*
=> (1 2 3 4 5)
```

And as a side effect *mylist* was changed.

A Shortcut

Using the list operator to create a list is the "long" way:

(list 1 2 'symbol-1 'symbol-2 "a string")

For convenience you can use quote to say what comes next, don't evaluate, it is data.

(quote (1 2 symbol-1 symbol-2 "a string"))

And further can simplify that with the single quote **'**

'(1 2 symbol-1 symbol-2 "a string")

Which evaluates to a list like the operator list but also removes the need to use a single quote to treat the symbol name as a symbol.

setf General Assignment and Lists

Lisp has an operator called setf that is more than a "this equals that". You can for example set individual elements:

```
(setf (second *mylist*) 99)
=> 99

*mylist*
=> (1 99 3 4 5)
```

Of course we can just replace the current value with:

```
(setf *mylist* '(1 2 3))
=> (1 2 3)
```