Common Lisp - The Tutorial Part 7

Closures, Loops and Strings

You spin me right 'round, baby, right 'round
Like a record, baby, right 'round, 'round, 'round
You spin me right 'round, baby, right 'round
Like a record, baby, right 'round, 'round, 'round
– Dead or Alive

See more at https://github.com/rabbibotton/clog/blob/main/LEARN.md

Introduction

Life tends to get boring when things are going in circles, but in the metaverse loops are action and functions are windows in.

Scope and Closures

Global symbols have a scope that allows any part of the lisp image to reach or modify their value. All other symbols have a lexical scope, i.e. they symbols are bound to values where they are textually defined and when that textual block of code is over, the symbols are no longer bound to those values. For example the parameter symbols are bound only within the function body to the arguments used to call the function until the function returns.

We define local symbols using let (or let* if there is a need for one definition to refer to another as part of their definition) which creates an implicit progn where they will be defined until completion of that progn. To illustrate:

```
(defun my-func (param1 param2)
  (let ((sum (+ param1 param2)))
    (print sum)))
```

[Syntax:

```
  (let ((symbol1 initial-value)
        (symbol2 initial-value)
        symbol2-noinit)
     body)                             ]
```

param1 and param2 are bound from the start of my-func until it's implicit progn is complete, i.e. the last right parenthesis is closed. sum is bound until let's implicit progn is complete.

Lisp also allows defining locally named functions in the same manner as let/let* called flet/labels (the reason for the "labels" name instead of flet* is historical, like car[1] and cdr[2]).

It is possible to return a function like other data and even though the source scope is lexically closed (the last right parenthesis closed) the environment where a function is defined lives on with it and why it is called a closure.

```
(defun my-adder (grow-by)
  (let ((sum 0))
    (lambda ()
      (incf sum grow-by))))

(defvar *two-counter* (my-adder 2))
(defvar *two-counter-two* (my-adder 2)) ; a second adder by 2
(defvar *three-counter* (my-adder 3))

(funcall *two-counter*)
=> 2

(funcall *two-counter*)
=> 4

(funcall *three-counter*)
=> 3

(funcall *two-counter-two*)
=> 2
```

The my-adder function is a "factory" for adders and with that you have the OO factory pattern and the fact that OO existed in Lisp from that 1960s in some form.

Loops

[1] "**C**ontents of the **A**ddress part of **R**egister number" http://jmc.stanford.edu/articles/lisp/lisp.pdf
[2] "**C**ontents of the **D**ecrement part of **R**egister number"

The other category of control flow operators after functions and branching is loops. Lisp has many flavors of loops and even an advanced version of loop with its own language that we will have a separate tutorial on[3].

loop..return

If return is never called loops forever.

```
(let ((n 0))
  (loop
    (princ ".")
    (if (> n 10)
        (return n)
        (incf n))))
```

dotimes

The loop repeats n times, n equals 0 .. n-1 on each loop.

```
(dotimes (n 10)
  (princ "."))
```

dolist

Each loop n is bound to the next element until all elements are done.

```
(dolist (n '(1 2 3 4 5))
  (princ n))
```

do

Do allows for multiple loops at once. Each loop is defined with:

```
 (a-symbol starting-val how-to-change-a-symbol)
```

```
(do ((x 1 (+ x 1))    ; loop 1
     (y 10 (- y 1))) ; loop 2
    ((> x 10))        ; terminate loops when true
  (princ x)
  (princ " - ")
  (princ y)
  (terpri))
```

[3] There are two extra languages in Lisp, format and loop and we will address both in next tutorial

String Processing

Strings are created using double quotes. Strings in Lisp are a single dimension array of characters.

Lisp provides string specific operators in addition to the more generic one used for sequences, arrays and equality.

Common String Operators:

```
(length x)                       ; string length
(string= x y)                    ; case sensitive equality

(string-upcase x)                ; return uppercase vs of x
(string-downcase x)              ; return lowercase vs of x
(string-capitalize x)            ; return x with each word capitalized
(string-trim x)                  ; trim white space from left and
right
(string-left-trim x)             ; trim white space from left
(string-right-trim x)            ; trim white space from right
(reverse x)                      ; reverse contents of string

(subseq x s e)                   ; return substring of x from s to e
(setf (subseq x s e) y)          ; replace in x fromm s to e with
                                 ; contents of y that fit in s to e

(concatenate 'string x y)        ; concatenate strings

(string #\x)                     ; convert character to string
(character "x")                  ; convert string to character

(parse-integer "123")            ; convert and integer string to
number
(read-from-string "123.232"      )     ; reads data from a string -
warning
(write-to-string x)              ; convert evaluated x to a string
```

Characters are written in Lisp as #\z - z being the character that it designates.

```
(char my-string x)               ; return character at x
(setf (char my-string x) y)      ; replace char at x with y

(char= x y)                      ; case sensitive equality
```

```
(code-char x)                   ; return char for value x
(char-code x)                   ; return value for char x
```

Some Character Constants:

```
#\space
#\newline
#\linefeed
#\return
#\tab
#\backspace
#\rubout
```