LISP NOTES #3

Loading LISP Programs

```
 [1]> (load "c:/lisp/reverse.lsp")
;; Loading file C:\lisp\reverse.lsp ...
;; Loaded file C:\lisp\reverse.lsp
T
 [2]> (symbol-function 'dorevs)
#<FUNCTION DOREVS (ORIG) (DECLARE (SYSTEM::IN-DEFUN DOREVS))
  (BLOCK DOREVS (REVS ORIG NIL))>
[3]> (dorevs '(a b c d))
(D C B A)
```

Recording the Session

```
(dribble “c:/lisp/session.lsp”)
;; Dribble of #<IO TERMINAL-STREAM> started on NIL.

#<OUTPUT BUFFERED FILE-STREAM CHARACTER #P"C:\\lisp\\sessionx.lsp">
[23]> (+ 2 3)
5
[24]> (dribble)
2009-06-24 17:35:55
;; Dribble of #<IO TERMINAL-STREAM> finished on NIL.
(dribble)
```

Features of Functional Programming

In standard programming the code to be executed had to be known ahead of time. In functional programming the code can be stored as data and evaluated as needed. Functional programs build up code (i.e. forms in LISP) as it runs then evaluate it (i.e forms in LISP) dynamically. In essence, the programs can write and evaluate its own code. Also, the fode (.e. functions) can be passed as arguments and functions accept code (i.e. functions) as arguments.

Mapping Functions

Def. Mapping functions are functions that can apply a function repeatedly to any number of sets of arguments. Please note that they take functions as arguments.
Note: see also the text book page 139 point (5) and (6).

Passing Functions as Arguments

Let’s define some helper functions first

(defun func2 (x) (* x x))
(defun func3 (x) (* x x x))
Now let's try this (which is incorrect)
(defun try1 (fn x) (fn x))
⇒ try1
(try1 'func2 5)
⇒ Undefined function FN

Why? When you evaluate the form (fn x), the code associated with the symbol FN is retieved and not with the value stored in the symbol FN.
Solution: use the APPLY or FUNCALL functions. They retrieve the code associated with the symbol stored in the value of a given symbol.

(defun try2 (fn x) (apply fn x))
⇒ try2
(defun try3 (fn x) (funcall fn x))
⇒ try3
Now try it:
(try2 'func2 '(5))
⇒ 25
(try3 'func2 5)
⇒ 25

Note: APPLY expects the second argument as a list this is why we call TRY2 passing '(5).
How do we make it so that we do not have to pass a list? Put it into a list inside the function.

(defun try4 (fn x) (apply fn (list x)))
⇒ try4
(try4 'func2 5)
⇒ 25

The following do the same and produce (A B C):

(apply 'cons '(a (b c)))
(funcall 'cons 'a '(b c))
(cons 'a '(b c))

Note: APPLY and FUNCALL work only with ordinary functions (i.e. not with DEFUN, SETQ etc.)

Examples

(+ '(1 2 3 4 5))
⇒ Error

```
(apply '+ '(1 2 3 4 5))
        ⇒ 15
```

Important Example

```
(defun sum-loop (func x)
  (if (zerop x)
    0
    (+
      (apply func (list x))
      (sum-loop func (1- x))
    )
  )
)
        ⇒ SUM-LOOP
(sum-loop 'func2 5)
        ⇒ 55
(sum-loop 'func3 5)
        ⇒ 225
(sum-loop 'sqrt 5)        ; this is a predefined function, and it can be passed as well !!!
        ⇒ 8.382333
```

What does sum-loop do?
It computes func(x)+func(x-1)+func(x-2)+…+func(2)+func(1)+0
So (sum-loop 'func2 5) does 5*5+4*4+3*3+2*2+1*1+0=55

Dynamic Scoping and FUNARG (Functional Argument) Problem

Question: If the function passed as an argument has free symbols, then are these symbols referring to the global (top-level) symbol or are they local (uninitialized)?
Answer: If the function passed as an argument has free symbols, then are these symbols have a dynamis scope i.e. they refer to the latest value of that symbol up the calling chain. Therefore they may refer to some local variable earlier that has the same symbol and if no such earlier local variable exists then they refer to the global (top-level) variable.
Definition: Dynamically scoped variables are interpreted in the context that exists when the function is applied.
Note: the book defines that Commpn LISP has dynamic scoping for the FUNARG problem but the version Common LISP that I used uses global scoping!

My Example

```
(setq x 5)
(defun test2 (y) (apply 'func2 (list x)))
(defun test1 (x) (test2 4))
```

Now

(test1 10)

⇒ 25

Note: it is possible to define CLOSURE for functions in LISP i.e. functions where free symbols will always be interpreted in the context in which that function was created.

EVAL Function

The APPLY and FUNCALL functions work only with ordinary function, but how about any arbitrary s-expression (even those that contain special functions as DEFUN or SETQ), can they still be evaluated? Yes, the function EVAL does that. EVAL evaluates the form given to it.
Note: (eval 'arg) evaluates the form arg
Note: (eval arg) evaluates the argument arg first, then evalueates this result next

(setq x '(cons 'a '(b c)))

⇒ (cons 'a '(b c))

(eval 'x)

⇒ (cons 'a '(b c))

(eval x)

⇒ (a b c)

Example (code On-The-Fly created and evaluated)

(eval '(cons 't '(2 3)))

⇒ (t 2 3)

More Mapping Functions

They take a function as an argument and apply it to a set of arguments.
MAPCAR, MAPC – apply a function to the set made up of corresponding elements of each of the given lists.

(mapcar '1+ '(100 200 300))

⇒ (101 201 301)

(mapcar '+ '(100 200 300) '(1 2 3))

⇒ (101 202 303)

(mapcar 'atom '(a b c (x y) nil (a b) x y))

⇒ (t t t nil t nil t t)

Note: the above example produces a list of ATOM properties of elements of the given list.
Note: MAPCAR returns a list of results of function applications, thus it has some overhead of constructing such list. MAPC returns just the first argument, thus it is more efficient.

(defun paint (obj col) (setf (get obj ‘color) col))
(mapcar ‘paint ‘(house barn lenin) ‘(white blue red))
⇒ (white blue red)
(mapc ‘paint ‘(house barn lenin) ‘(white blue red))
⇒ (house barn lenin)

MAPLIST, MAPL – apply a function to the set made up of corresponding sub-lists of each of the given lists.

(maplist ‘length ‘(1 2 3))
⇒ (3 2 1)
(maplist #‘(lambda (x) x) ‘(1 2 3))
⇒ ((1 2 3) (2 3) (3))
(maplist #'(lambda (x y) (cons (car x) y)) '(1 2 3) '(a b c))
⇒ ((1 A B C) (2 B C) (3 C))

Note: MAPL is hardly ever used since it is not very useful.
Tip: You can use MAPLIST to compute a sliding-window computations on a given list (ex. Average, min, max etc.)

Lambda Functions and the Essence of Functions

Note: see also the text book chapter 9.8 point (1) and (2).
Essentially a function meaning is the list of its formal parameters and what it does. The name that we give to functions is just a label used for convenience.
We can have the function with no name using the LAMBDA function and by constructing a lambda expressions:

(lambda list-of-parameters list-of-blocks)

The following actually are the same functions, the first has just an extra label REDSYMB:

(defun redsymb (a) (and (symbolp a) (equal (get a ‘color) ‘red)))
(lambda (a) (and (symbolp a) (equal (get a ‘color) ‘red)))

And can be used equivalently

(mapcar ‘redsymb ‘(a house lenin b))
⇒ (nil nil t nil)
(mapcar #‘(lambda (a) (and (symbolp a) (equal (get a ‘color) ‘red))) ‘(a house lenin b))
⇒ (nil nil t nil)

Note: #’ is a shortcut for (function …) and is needed by my version of Common LISP to ensure that the user is aware that he/she is passing a function as an argument. Other implementations of Common LISP may work just fine if we submit

(mapcar ‘(lambda (a) (and (symbolp a) (equal (get a ‘color) ‘red))) ‘(a house lenin b))

Lambda expressionbs allow to define functions without the need to permanently storing it (i.e. associating it with a name). Lambda notation allows for lambda abstraction, which is to abstract what the function does regardless of the name given to it.

Example

```
(sum-loop #’(lambda (x) (* x x x x x x)) 5)
        96825
```

Note: DEFUN actually builds a lambda expression and associates it with a given symbol. You can see what lambda expression is associated with a symbol by using the SYMBOL-FUNCTION function

```
(symbol-function ‘sum-loop)
#<FUNCTION SUM-LOOP (FUNC X) (DECLARE (SYSTEM::IN-DEFUN SUM-LOOP))
  (BLOCK SUM-LOOP
   (IF (ZEROP X) 0 (+ (APPLY FUNC (LIST X)) (SUM-LOOP FUNC (1- X)))))>
```

Note: SETF function can be changed to change the association of the symbol and the lambda expression

```
(func2 5)
        ⇒ 25
(setf (symbol-function ‘func2) #’(lambda (x) (+ x 3)))
        #<FUNCTION :LAMBDA (X) (+ X 3)>
(func2 5)
        ⇒ 8
```

Some Useful Functions

(read) – reads an s-expression from the keyboard
(print …) – prints some output to std. out
(time form) – measures the time to evaluate the given form
(describe ‘obj) – describes a symbol (the value and lambda expression)
(machine-type) – returns the type of machine that Common LISP is running on