Lisp notebook

M. Yas. Davoodeh

October 19, 2020

Contents

1 Basics

Throughout the document:

- Tor stands for operator and rand stands for operand.

Listing 1: General syntax of Lisps

```
(tor rand rand rand ...)
```

`'` or `quote` marks code as a data and keeps it from evaluation.

Listing 2: `quote` example

```
'(x y z)
(quote (x y z))
```

To evaluate any lisp give it to `eval` function

Listing 3: `eval` example

```
(setq x '(+ 5 3)) ; the value is literal list (+ 5 3) not 8
(eval x) ; it becomes 8 in here
```

```
8
```

Listing 4: Comments in Lisps

```
;;;; Describe program with four ;;;;
;; Basic comment (or ;;;)
;; Indented comment
;;; (message "Hello") ; After code comment
```

Listing 5: Function syntax

```
(defun name (input1 input2)
  command1
  command2)
```

In Scheme you just define it in one of the ways below. The first one changes to the second.

Listing 6: Scheme equivalent for function definition

```
(define (name input1 input2)
  command1
  command2)
(define name (lambda (input1 input2)
                     command1
                     command2))
```

1.1 Variables

Setting and assigning values in Scheme family is solely done with `define`.

1.1.1 Declaration

There is no need to declare variables until compiled.

Listing 7: Defining a global variable

```
(defvar name 'value)
```

Listing 8: Use `let` to create a local scope for a variable.

```
(let ((var val) var2 (var3 val3) var4 ...)
  command
  command2)
```

1.1.2 Assign value

Listing 9: Assigning a value 5 to symbol `x`

```
(set 'x 5)
```

`setq` stands for setting a Quoted symbol. :#+CAPTION: Manipulating variable values

```
(setq x 5)
(setq c 'd)
(set c 10)
d
```

```
10
```

Listing 10: `set!` replaces `setq` in Scheme

```
(define value 0)
(set! value 1)
value
```

```
1
```

1.2 Compilation

Compilation needs variable declaration

Listing 11: Compiling a function to increase speed

```
(compile 'functionname)
```

1.3 Control constructs

1.3.1 if

Listing 12: if syntax

```
(if (criteria)
  ;; To make a clause (for example for true condition of if) use progn or let (or begin in Scheme)
  true-command
  else-command1
  else-command2
  else-command...)
```

1.3.2 Loops

1. do

Listing 13: do syntax

```
(do (step) (condition) code)
(do ((var init-val (step)) (var2 init-val2 (step2))) ((condition))
    code)
```

1.4 Manipulating lists and pairs

A pair is a structure defined as below. Pairs can be constructed with cons command. a, car or the first element is the left most symbol. d, cdr (□*k*□*d*□*r*) is the other element.

Listing 14: Ways to define a pair

```
'(a . d)
(cons 'a 'd)
```

```
(A . D)
```

A list is a list of pairs and can be defined like below

Listing 15: Ways to define a list

```
(list 'a 'b 'c 'd)
'(a . (b . (c . (d))))
'(a b c d)
```

A B C D

Listing 16: Definition of variables used in snippets below.

```
(setq n 3)
(setq mylist '(a b c d))
```

A B C D

Getting the first item of a list, a or `car` is done like below.

Listing 17: `car` example

```
(first mylist)
(car mylist)
```

```
A
```

Getting the rest of objects, `d` or the `cdr`.

Listing 18: `cdr` example

```
(rest mylist)
(cdr mylist)
```

B C D

`car` and `cdr` can be chained in such logic: Remove `c` and `r`; From right to left, do the operations left from the string. For example, a `d` in such logic is a `cdr` and an `a` stands for a `car`. Therefore a `cadr` returns the second item. `ad` means first do a `d` (`cdr`) and then do an `a` (`car`)

Listing 19: Chaining example using `cadr`

```
(cadr mylist)
```

```
B
```

Calling the nth number of input (starting from 0).

Listing 20: Calling the nth item of a list

```
(nth n mylist)
```

```
D
```

Listing 21: Calling the n th item of a list in Scheme

```
(list-ref '(a b c d) 3)
```

```
d
```

Listing 22: Getting length of a list

```
(length mylist)
```

```
4
```

Listing 23: Giving a list as rands of tors

```
(apply '+ '(1 2 3 4)) ; = sum()
```

```
10
```

1.5 IO

1.5.1 Printing

Listing 24: `print` prints lisp objects.

```
(print "Hello World!")
(print "Hello World!")
```

```
"Hello World!"
"Hello World!"
```

`print` output can be read back by `read` function.

Listing 25: `princ` prints no newline or delimiter but `prin1` just removes newline and prints delimiter.

```
(prin1 "   Hello World!")
(princ "   Hello World!")
(princ "Hello World!")
(terpri) ; newline
(princ "Hello World!")
```

```
"   Hello World!"   Hello World!Hello World!
Hello World!
```

Listing 26: `display` works just like `princ` in Scheme

```
(display "Hello World!")
(display "Hello World!")
(newline)
(display "Hello World!")
```

```
Hello World!Hello World!
Hello World!
```

Listing 27: `message`, an Elisp specific function to print in `messages buffer`

```
(message "Hello World!")
```

Listing 28: `insert` prints string in the current buffer.

```
(insert "x")
```

1.6 Functions

Functions can be nested. Last line is the return value of a function.

1.6.1 Nested functions

Listing 29: Inner functions can access symbols of outer functions.

```
(defun assert-equal (a b)
  (defun print-error () ; arguments are not directly passed
    (princ a)
    (princ " is not equal to ")
    (princ b)
    (terpri))
  (unless (= (eval a) (eval b))
      (print-error)))

(assert-equal '3 '(+ 1 2))
(assert-equal '3 '(* 1 2))
```

```
3 is not equal to (* 1 2)
```

Listing 30: Also in Scheme inner functions can access outer symbols.

```
(define (circle-details r)
  (define pi 3.1415)
  (define (area)
    (* pi r r))
  (define (circum)
    (* 2 pi r))
  (list (area) (circum)))

(circle-details 3)
```

1.6.2 Returning functions

```
(define (make-add-one)
  (define (inc x)
    (display (+ 1 x)))
  inc)
(define myfn (make-add-one))
(myfn 2)
```

```
3
```

2 Examples

2.1 Sum

Listing 31: Sum of the number 1 to n.

```
(defun sum (n)
  (if (= n 1)
    1
    (+ n (sum (- n 1)))))
(sum 4)
```

```
10
```

2.2 Map

Listing 32: A replica of `map`

```
(defun mp (fn lst)
  (unless (null lst)
    (cons (funcall fn (car lst)) (mp fn (cdr lst)))))

(defun cube (x)
  (* x (* x x)))

(mp 'cube '(2 -3 4))
```

```
(8 -27 64)
```

Listing 33: A replica of `map` in Scheme

```scheme
(define (mp fn lst)
  (if (nil? lst)
      #nil
    (cons (fn (car lst)) (mp fn (cdr lst)))))

(mp abs '(2 -3 4))
```

```
(2 3 4)
```