ANSI Common Lisp Practice

ismdeep

December 31, 2019

Contents

1 Chapter 01

1.1 sum

```
; (dotimes (i n s) () ...)
; i => [0, 1, ... , n]
; return value is s
; ... is operations

(defun sum (n)
   (let ((s 0))
      (dotimes (i n s)
         (incf s i))))

(format t "~D~%" (sum 10))
```

1.2 addn

```
; lambda ?
; I don't know how to use it yet. -_-

(defun addn (n)
   #'(lambda (x)
      (+ x n)))

(format t "~A~%" (addn 10))
```

2 Chapter 02

2.1 Form

```
(format t "~A~%" (+ 1 2))
(format t "~A~%" (+ 1 2 3 4 5))
(format t "~A~%" (/ (- 7 1) (- 4 2)))
```

2.2 Evaluation

```
(format t "~A~%" (quote (+ 3 5)))
(format t "~A~%" '(+ 3 4))
```

2.3 Data

```
(format t "~A~%" 'Hello)
(format t "~A~%" '(my 3 "Sons"))
(format t "~A~%" (list 'my (+ 2 1) "Sons"))
(format t "~A~%" ())
(format t "~A~%" nil)
```

2.4 List Operations

```
(format t "~A~%" (cons 1 '(2 3 4)))
(format t "~A~%" (car '(1 2 3 4)))
(format t "~A~%" (cdr '(1 2 3 4)))
(format t "~A~%" (car (cdr (cdr '(1 2 3 4)))))
(format t "~A~%" (third '(1 2 3 4)))
```

2.5 Truth

```
(format t "~A~%" (listp '(1 2 3 4)))
(format t "~A~%" (null nil))
(format t "~A~%" (not nil))
(format t "~A~%" (if (listp '(a b c))
   (+ 1 2)
   (+ 5 6)))
```

2.6 Functions

```
(defun our-third (x)
   (car (cdr (cdr x))))

(format t "~A~%" (our-third '(a b c d)))
```

2.7 Recursion

```
(defun is-member (obj lst)
   (if (null lst)
      nil
      (if (eql (car lst) obj)
         T
         (is-member obj (cdr lst)))))

(format t "~A~%" (is-member 1 '(2 3 4 1 7 8)))
```

2.8 Reading Lisp

```
(defun our-member (obj lst) (if (null lst) nil
    (if
(eql (car lst) obj) lst (our-member obj (cdr
    lst)))))
```

2.9 Input and Output

```
(format t "~A plus ~A equals ~A. ~%" 2 3 (+ 2 3))

(defun askem (string)
   (format t "~A~%" string)
   (read))

(let ((age (askem "How old are you?")))
   (format t "I'm ~A year old.~%" age))
```

2.10 Variables

```
; create local variable through let
(let ((x 1) (y 2))
   (format t "~A~%" (+ x y)))

; create local variable throught let in a
    function
(defun ask-number ()
   (format t "Please enter a number.~%")
   (let ((val (read)))
      (if (numberp val)
```

```
        val
        (ask-number))))

; call function ask-number
(format t "~A~%" (ask-number))

; create a global variable
(defparameter *global-var* 100)

; create a global constant
(defconstant LIMIT 100)


(format t "~A~%" *global-var*)

; test a symbol is a global variable
(format t "~A~%" (boundp '*global-var*))

(format t "~A~%" LIMIT)
```

2.11 Assignment

```
(declaim (sb-ext:muffle-conditions cl:warning))

(setf *glob* 98)

(format t "~A~%" *glob*)

(format t "~A~%" (let ((n 10))
   (setf n 2)
   n))

(setf x (quote (a b c)))
(setf (car x) 'x)
(format t "~A~%" x)

(setf a 1
    b 2
    c 3)

(format t "~A~%" b)
```

2.12 Functional Programming

```
(defparameter lst '(c a r a t))
(format t "~A~%" (remove 'a lst))
(format t "~A~%" lst)
(setf lst (remove 'a lst))
(format t "~A~%" lst)
```

2.13 Iteration

```
; iteration version
(defun show-squares-iteration (start end)
   (do ((i start (+ i 1)))
      ((> i end) 'done)
      (format t "~A ~A~%" i (* i i))))

; recursion version
(defun show-squares-recursion (start end)
   (if (> start end)
      'done
      (progn
         (format t "~A ~A~%" start (* start
            start))
         (show-squares-recursion (+ start 1)
            end))))


(show-squares-iteration 1 10)
(show-squares-recursion 1 10)



; our-length iteration version
(defun our-length-iteration (lst)
   (let ((len 0))
      (dolist (obj lst)
         (setf len (+ len 1)))
      len))

; our-length recursion version
(defun our-length-recursion (lst)
   (if (null lst)
      0
      (+ (our-length-recursion (cdr lst)) 1)))

(defparameter *lst* (quote (1 2 3 4 5)))
(format t "~A~%" (our-length-iteration *lst*))
(format t "~A~%" (our-length-recursion *lst*))
```

2.14 Functions as Objects

```
(format t "~A~%" (function +))
(format t "~A~%" #'+)
(format t "~A~%" (apply #'+ '(1 2 3)))
(format t "~A~%" (apply #'+ 1 2 '(3 4 5)))
(format t "~A~%" (funcall #'+ 1 2 3 4 5))
(format t "~A~%" (lambda (x y) (+ x y)))
(format t "~A~%" ((lambda (x) (* x x)) 10))
(format t "~A~%" (funcall #'(lambda (x) (* x x))
    10))
```

2.15 Types

```
(format t "~A~%" (typep 27 'integer))
```

3 Chapter 03

3.1 Conses

```
(defparameter *x* nil)

(setf *x* (cons 'a nil))
(format t "~A~%" *x*)

(setf *x* (cons 'a '(b c)))
(format t "~A~%" *x*)

(let ((y nil))
   (setf y (list 'a 'b 'c))
```

```
   (format t "~A~%" y)
   (format t "~A~%" (car y))
   (format t "~A~%" (cdr y)))

(let ((z nil))
   (setf z (list 'a (list 'b 'c) 'd))
   (format t "~A~%" z)
   (format t "~A~%" (car (cdr z))))

(defun our-listp (x)
   (or (null x) (consp x)))

(defun our-atomp (x) (not (consp x)))

(format t "~A~%" (our-listp (list 'a 'b 'c)))
(format t "~A~%" (our-listp ()))
(format t "~A~%" (our-atomp 'a))
(format t "~A~%" (our-atomp (list 'a 'b)))
```

3.2 Equality

```
(format t "~A~%" (eql (cons 'a nil) (cons 'a
    nil)))
(format t "~A~%" (equal (cons 'a nil) (cons 'a
    nil)))
```

3.3 Why Lisp Has No Pointers

```
(let ((x nil) (y nil))
   (setf x '(a b c))
   (setf y x)
   (format t "~A~%" x)
   (format t "~A~%" y)
   (format t "~A~%" (eql x y)))
```

3.4 Building Lists

```
(let ((x nil) (y nil))
   (setf x '(a b c)
         y (copy-list x))
   (format t "~A~%" x)
   (format t "~A~%" y)
   (format t "~A~%" (eql x y))
   (format t "~A~%" (equal x y)))

(defun our-copy-list (lst)
   (if (atom lst)
      lst
      (cons (car lst) (our-copy-list (cdr
         lst)))))

(format t "~%")

(let ((x nil) (y nil))
   (setf x '(a b c)
         y (our-copy-list x))
   (format t "~A~%" x)
   (format t "~A~%" y)
   (format t "~A~%" (eql x y))
   (format t "~A~%" (equal x y)))

(format t "~A~%" (append '(a b) '(c d) 'e))
```

3.5 Example Compression

```
; run-length encoding compression
(defun compress (x)
   (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
   (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
         (if (eql next elt)
            (compr elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
               (compr next 1 (cdr lst)))))))

(defun n-elts (elt n)
   (if (> n 1)
      (list n elt)
      elt))


; run-length decoding uncompression
(defun uncompress (lst)
   (if (null lst)
      nil
      (let ((elt (car lst)) (rest (uncompress
          (cdr lst))))
         (if (consp elt)
            (append (apply #'list-of elt) rest)
            (cons elt rest)))))

(defun list-of (n elt)
   (if (zerop n)
      nil
      (cons elt (list-of (- n 1) elt))))

(format t "~A~%" (compress '(1 1 1 0 1 0 0 0 0
    1)))
(format t "~A~%" (uncompress '((3 1) 0 1 (4 0)
    1)))
```

3.6 Access

```
(defun our-nthcdr (n lst)
   (if (zerop n)
      lst
      (our-nthcdr (- n 1) (cdr lst))))

(defparameter *glob* '(1 2 3 4))

(format t "~A~%" *glob*)
(format t "~A~%" (nth 0 *glob*))
(format t "~A~%" (nthcdr 2 *glob*))
(format t "~A~%" (our-nthcdr 2 *glob*))
(format t "~A~%" (cadr *glob*))
```

3.7 Mapping Functions

```
(format t "~A~%" (mapcar #'(lambda (x) (* x x))
    '(1 2 3 4 5 6 7 8)))

(format t "~A~%" (mapcar #'list '(a b c) '(1 2 3
    4)))

(format t "~A~%" (maplist #'(lambda (x) x) '(a b
    c)))
```

3.8 Trees

```
(defun our-copy-tree (tr)
   (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))

(let ((tree '(a (b c) d)) (tree-2 nil))
   (setf tree-2 (our-copy-tree tree))
   ; (setf tree-2 tree)
   (format t "~A~%" tree)
   (format t "~A~%" tree-2)
   (format t "~A~%" (eql tree tree-2)))


(format t "~A~%" (substitute 'y 'x '(and
    (integerp x) (zerop (mod x 2)))))

(format t "~A~%" (subst 'y 'x '(and (integerp x)
    (zerop (mod x 2)))))

(defun our-subst (new old tree)
   (if (eql tree old)
      new
      (if (atom tree)
         tree
         (cons (our-subst new old (car tree))
               (our-subst new old (cdr tree))))))

(format t "~A~%" (our-subst 'y 'x '(and
    (integerp x) (zerop (mod x 2)))))
```

3.9 Understanding Recursion

```
(defun len (lst)
   (if (null lst)
      0
      (+ (len (cdr lst)) 1)))

(format t "~A~%" (len '(a a b b d)))
```

3.10 Sets

```
(defun is-member (obj lst)
   (if (consp (member obj lst))
      T
      Nil))

(format t "~A~%" (is-member 'a '(a b c)))

(format t "~A~%" (member '(a) '((a) (z)) :test
    #'eql))
(format t "~A~%" (member '(a) '((a) (z)) :test
    #'equal))
(format t "~A~%" (member 'a '((a) (z)) :key
    #'car))
(format t "~A~%" (member 'a '((a) (z)) :key
    #'car :test #'equal))
(format t "~A~%" (member 'a '((a) (z)) :key
    #'car :test #'eql))
(format t "~A~%" (member-if #'oddp '(2 3 4)))


(format t "~A~%" (adjoin 'a '(a b c)))

(format t "~A~%" (union '(a b c) '(c d f)))
(format t "~A~%" (intersection '(a b c) '(b c
    d)))
(format t "~A~%" (set-difference '(a b c) '(b c
    d)))
```

3.11 Sequences

```
(format t "~A~%" (length '(a b c)))
(format t "~A~%" (subseq '(a b c d) 1 2))
(format t "~A~%" (subseq '(a b c d) 1))
(format t "~A~%" (reverse '(a b c)))

(defun mirror? (s)
   (let ((len (length s)))
      (and (evenp len)
         (let ((mid (/ len 2)))
            (equal (subseq s 0 mid)
                   (reverse (subseq s mid)))))))

(format t "~A~%" (mirror? '(a b b a)))

(format t "~A~%" (sort '(0 2 1 3 8) #'>))

(format t "~A~%" (every #'oddp '(1 3 5)))
(format t "~A~%" (some #'evenp '(1 2 3 5)))
(format t "~A~%" (every #'> '(1 2 3) '(0 1 2)))
```

3.12 Stacks

```
(let ((stack '()))
   (format t "~A~%" stack)
   (push 'a stack)
   (push 'b stack)
   (format t "~A~%" stack)
   (format t "~A~%" (pop stack))
   (format t "~A~%" stack))

; Define a reverse function with stack
(defun our-reverse (lst)
   (let ((acc nil))
      (dolist (item lst)
         (push item acc))
      acc))

(format t "~A~%" (our-reverse '(a b c d)))
```