Slide 10: Hello World

(defun hello ()
    (print "hello world")
)

This prints twice because it is printed once in the function and also returned from the function call. The last thing to be returned from a Lisp program is printed to the screen at the end of execution.

Can compensate by not printing and simply returning value:

(defun hello ()
    "hello world"
)

Slide 14: List Manipulation

Variable manipulation first:

(setf x 1)

(setf cake `pants)

(setf $!@$ '&&&&&)

Different ways to define a list:

(setf mylist `(1 2 3))

(setq mylist `(1 2 3))
(set 'mylist `(1 2 3))
(set 'mylist `(1 2 3))
(set 'mylist (quote (1 2 3)))
(set (quote mylist) (quote (1 2 3)))

Car & cdr:
(setf mylist `(5 pants orange))
(car mylist)
(cdr mylist)

How would you get pants?
Can cdr forever:
(cdr (cdr mylist))
(cdr (cdr (cdr mylist)))

But can’t (car (car mylist)) – for that need:
(setf mylist `(((5) pant) orange))
Now can (car (car (car mylist)))

Cons:
(setf laundry `(pants shirt))

Order matters:
(cons ‘hat laundry) – list with hat at front
(cons laundry ‘hat) – cell holding laundry list in left half and hat in right half

This means that
(cons 1 (cons 2 (cons 3 nil)))

and

`(1 2 3)

are equivalent

---

**Slide 16: List Manipulation Continued**

Illustrate difference between quote and list

(list 1 2 (* 5 5))

Vs

(quote (1 2 (* 5 5))) and

`(1 2 (* 5 5))

(listp 5)

(listp `(1 2 3))

Push and pop behave as anticipated:

(setf mylist `(5 pants orange))

(push `green mylist)

mylist

(pop mylist)

Mylist

Append concatenates lists:

(append `(a b c) `(1 2 3))
Contrast with cons:
(cons `(a b c) `(1 2 3))

Remove and member also behave as expected, except:
returns tail starting with element if found:
lists not modified in place
(setf mylist `(shoes pants shirt))
(member `pants mylist)
(member `pants (remove `pants mylist))

No surprises with length
(length mylist)

Eval: the big one – evaluates a string as though it were code
It’s what’s being run whenever we hit return
-think of quote as the anti-eval
(eval `(* 3 4))
(setf x (* 3 4)
(setf x `(* 3 4))
(eval x)

**Slide 17: Arithmetic**
(setf x 5)
(incf x)
(decf x)

**Slide 21: Property List**
(setf (get 'x 'y) 4)
(get 'x 'y)

**Slide 24: let**
(let ((a 5))
(+ a 1))

**Slide 24: Conditionals**
(if t 10 20)
(if nil 10 20)
(if nil 10)

**Slide 27: Functions**
(defun bringtowel (laundrylist)
  (append laundrylist '(towel)))

(setf func1 (lambda(x) (+ x 3)))
(setf func2 (lambda(x) (* x 10)))
(defun call (x y) (funcall x y))
(call func1 3)
(call func2 10)

Apply vs funcall:
(funcall #'+ 3 4)
(apply #'+ 3 4 '(3 4))

Slide 28: Mapping functions
(mapcar func1 '(1 2 3))

Slide 31: Equal

;Code source: http://stackoverflow.com/questions/4427321/setting-up-a-equal-function-in-common-lisp-using-only-eq
(defun list-equality (list1 list2)
  (if (and (not (null list1))
           (not (null list2)))
      (let ((a (car list1)) (b (car list2)))
        (cond ((and (listp a) (listp b))
               (and (list-equality a b)
                    (list-equality (cdr list1) (cdr list2))))
              (t
               (and (eq a b)
                    (list-equality (cdr list1) (cdr list2))))
      (= (length list1) (length list2))))