Higher-order functions

York University
Department of Computer Science and Engineering

Overview

- Higher-order functions
- Apply and funcall
- Eval
- Mapping functions: mapcar, mapc, maplist,mapl

[ref.: chap 8, 9 - Wilensky]

Almighty functions!

- Higher- order functions can accept functions as inputs (and can return functions as outputs)
- If we can write functions that work on functions, we can have programs that can retrieve, create, and execute programs
- For this purpose,
  - We need to be able to accept functions as arguments
  - We need to be able to apply functions to arguments

Example

- A function that adds up the first n integers: $f(n) = \sum_{i=1}^{n} i$

```
(defun  sum_to (n)
        (do  ((i  n  (1-  i))  (sum  0  (+  sum  i)))
             ( (zerop  i)  sum)))
```

- A function that adds up the square roots of the first n integers
  – Change to (sum 0 (+ sum (sqrt i))))

- A function that adds up the squares, or cubes of the first n integers ..., rewrite again?!

Easy?

- A function to add up results of application of another function to the first n integers

$$f(g,n) = \sum_{i=1}^{n} g(i)$$

```
(defun  sum_fun (func   n)
                 (do  ((i  n  (1-  i))  (sum  0  (+  sum  (func   i))))
                      ( (zerop  i)  sum)))
```

The above code does not work. Why?

Value vs. function definition

- What does LISP do to evaluate a form such as
  (func   i) ?
  - Assumes func is a function, looks at its **function definition**
  - Applies the function definition to the actual argument (value of i)

  - When we pass the name of the function (e.g. sqrt) as the argument of sum_fun, we set the **value** of func to sqrt, not its **function definition**!

Apply

- **<u>Apply</u>** applies its first argument as a function to its second argument
- Second argument must be a list of arguments for the function
- Examples

```
> (apply  '+  '(1  2  3))
6
> (apply  'cons  '(1  (2  3)))
(1  2  3)
> (apply  'car  '((a   b   c)))
A
```

Back to our sum-fun example

- We can correct our previous code to:

```
> (defun  sum_fun (func   n)
        (do  (  (i  n  (1-  i))
                (sum  0  (+  sum  (apply  func  (list  i)))))
              ( (zerop  i)  sum)))

> (sum_fun  'sqrt  2)
2.4142137
> (defun  squared (x)  (*  x  x))
SQUARED
> (sum_fun  'squared  2)
5
```

Using Lambda functions

- Using lambda functions makes it easy to have temporary functions. For example, instead of defining squared and then using it:

  > (defun squared (x) (* x x))
  SQUARED
  > (sum_fun 'squared 2)
  5

  We can write:
  > (sum_fun (lambda (x) (* x x)) 2)
  5

Funcall

- Funcall is similar to apply, different in just passing arguments
  - Second argument is the name of a function
  - The rest are arguments to that function

```
> (apply  ‘+  ‘(1  2  3))
6
> (funcall  ‘+  1 2 3)
6
> (apply ‘cons  ‘( a  (b  c))
(A B C)
> (funcall ‘cons ‘a  ‘(b  c))
(A B C)
```

```
> (apply  ‘car  ‘((a  b  c)))
A
> (funcall  ‘car  ‘(a  b  c))
A
```

Eval

- **Eval** evaluates its only argument

```
> (setq x '(+ 1 2 3))
(+ 1 2 3)
> (eval x)
6
> (eval '(cons 'a '(b c)))
(A B C)
```

- Note that, as usual, the argument will be evaluated first and then **eval** will be applied to it.

```
> (eval (cons 'a '(b c)))
Error! Undefined function A!
```

Example

```
> (setq  v1  'v2)
> (setq  v2  'v3)
```

> v1

V2

> (eval   v1)

V3

> (eval 'v1)

V2

(eval   (cons '+ '(1 2 3)))

6

eval vs. apply

- Can we write **eval** using **apply**?

$$\text{(eval L)} \stackrel{?}{\equiv} \text{(apply (car L) (cdr L))}$$

- Works in some cases:

(setq x ‘(+ 1 2 3))	→	(+ 1 2 3)
(eval x)	→	6
(apply $\underbrace{\text{(car x)}}_{+}$ $\underbrace{\text{(cdr x)}}_{(1\ 2\ 3)}$)	→	6

eval vs. apply

- Does not always work!
  - **Apply** does not work with special operators, such as setq

(setq x ‘(setq y 25)) → (SETQ Y 25)
(<u>eval</u> x) → 25
(<u>apply</u> (car x) (cdr x)) → Error! Setq is a special operator!

  - **Eval** works with constants and variables too

(setq x 2) → 2
(eval x) → 2
(apply (car x) (cdr x)) → Error! 2 is not a list!

Example

- Defining our own if function using cond

```
> (setq  n  10)
10
> (our-if   (<  n  5)   '(+   n   2)   '(-   n  3))
7

(defun  our-if   (test  trueform  falseform)
          (cond        (test     (eval   trueform))
                       (t        (eval   falseform))))
```

  - Note that (< n 5) is evaluated to **t** or **nil** first, and then passed on to our-if
  - For above example, what does **cond** return in our-if?

    Answer. The second cond clause will be evaluated, returning 7 and therefore cond will return 7.

Example (cont.)

- We can also write the code this way (why?)

```
(defun our-if2 (test trueform falseform)
       (eval     (cond (test   trueform)
                       (t      falseform))))
```

- If we evaluate the following, what does **cond** return in our-if2?

  (setq n 10)
  (our-if2 (< n 5) ‘(+ n 2) ‘(- n 3))

  Answer. It returns (- n 3) to be evaluated by eval.

Context problems with eval

- Context in which forms are evaluated

```
> (defun our-if3 (test trueform falseform)
        (setq n 100)
        (cond   (test   (eval  trueform))
                (t      (eval  falseform))))

> (setq n 10)
> (our-if3 (< n 5) '(+ n 2) '(- n 3))
97
```

10 100

Be careful in which context the forms are evaluated!

Exercise: What if we use **let** instead of **setq** in definition of our-if3?

Mapping functions

- Mapping functions apply a function to multiple inputs.
  - **Apply** applies a function to one input (that may be a list).

- Example:

> (mapcar ‘1+ ‘(10 20 30 40))
(11 21 31 41)

> (mapcar ‘atom ‘(x (a b) c nil 10))
(T NIL T T T)

> (mapcar ‘+ ‘(10 20 30) ‘(1 2 3))
(11 22 33)

mapcar, mapc

- **Mapcar**
  - Evaluates all its arguments
  - Starts with a nil *result* (an empty list)
  - Until the arguments are empty, loops
    - Applies its first argument to the **car**s of each latter argument
    - **cons**es *result* with the result of above application
    - **cdr**s down the argument lists
  - Returns *result*

- **Mapc**
  - Just like mapcar, except it does not construct *result*
  - Less computation since no **cons**ing
  - Returns its second argument

Example

- Assume we want to set coordinates x and y of four points p1 to p4.

  P1 (0, 0)   P2(1,2)   P3(4,-1)   P4(2,3)

  – Assume we are using properties x and y for symbols p1 to p4 to store the coordinates

```
(setf  (get  p1   'x)  0)
(setf  (get  p1   'y)  0)
(setf  (get  p2   'x)   1) ...
```

  – It is more convenient to define a function such as:

```
(defun setC (point  xval   yval)
        (setf   (get   point   'x)  xval)
        (setf   (get   point   'y)  yval))
```

Example (cont.)

- Now we can use **mapcar**:

```
> (mapcar  'setC  '(p1 p2 p3 p4)
                  '(0  1  4  2)
                  '(0  2  -1  0))
```

- What does mapcar return?

```
(0 2 -1 0)
```

Mapcar returns a list of all values returned by setC

(which is the value returned by the last form in setC)

- We don't need the return value, so it's better to use **mapc**:

```
> (mapc  'setC  '(p1 p2 p3 p4) '(0 1 4 2) '(0 2 -1 0))
(P1  P2  P3  P4)
```

Mapc returns its second argument.

maplist, mapl

- Similar to mapcar and mapc
- Apply function to successive **cdr**s instead of **car**s

- Example:

```
> (maplist 'append '(a) '(x))
((A X))

> (maplist 'append '(a b) '(x y))
((A B X Y) (B Y))

> (maplist 'append '(a b c) '(x y z))
((A B C X Y Z) (B C Y Z) (C Z))
```

Exercise: Substitute append with **cons** or **list**, and see what maplist returns.

Lambda notation again!

- Lambda abstractions can also be used with mapping functions:

```
> (mapcar  (lambda (x) (* x 2))  ‘(10  20  30))
(20   40   60)

> (mapcar  (lambda (x) (cons  ‘a  x))  ‘((x y z) (1 2 3) (nil (b) c)))
((A  X  Y  Z) (A  1  2  3)  (A  NIL  (B)   C))

> (mapcar (lambda (x y) (+ (* 10  x)  y))  '(1  5  7) '(4  6  8))
(14   56   78)
```