Review of Functional Programming

York University CSE 3401
Vida Movahedi

Assume we have entered the following expressions in the LISP interpreter:

```
> (setq x 5)
   5
> (setq lst '(1 2 3 4))
  (1 2 3 4)
> (setq fname #'(lambda (x) (* 10 x)))
  #<FUNCTION :LAMBDA (X) (* 10 X)>
> (setq gname #'(lambda (x) (cons x 'x)))
  #<FUNCTION :LAMBDA (X) (CONS X 'X)>
```

How would LISP respond to the following?

```
> (car lst)
1

> (cdr lst)
(2 3 4)

> (cadr lst)
2
```

```
> (setq x 5)
   5
> (setq lst '(1 2 3 4))
  (1 2 3 4)
> (setq fname #'(lambda (x) (* 10 x)))
  #<FUNCTION :LAMBDA (X) (* 10 X)>
> (setq gname #'(lambda (x) (cons x 'x)))
  #<FUNCTION :LAMBDA (X) (CONS X 'X)>

> (fname  lst)
*** - EVAL: undefined function FNAME

> (fname  (car lst))
*** - EVAL: undefined function FNAME

> (apply fname  lst)
*** - EVAL/APPLY: too many arguments given to :LAMBDA

> (apply fname (car lst))
*** - EVAL/APPLY: too few arguments given to :LAMBDA
```

```
> (setq x 5)
   5
> (setq lst '(1 2 3 4))
  (1 2 3 4)
> (setq fname #'(lambda (x) (* 10 x)))
  #<FUNCTION :LAMBDA (X) (* 10 X)>
> (setq gname #'(lambda (x) (cons x 'x)))
  #<FUNCTION :LAMBDA (X) (CONS X 'X)>

> (apply fname (list (car lst)))
10

> (mapcar fname lst)
(10 20 30 40)

> (mapc fname lst)
(1 2 3 4)

> (1 . nil)
*** - EVAL: 1 is not a function name
```

```
> (setq x 5)
  5
> (setq lst '(1 2 3 4))
  (1 2 3 4)
> (setq fname #'(lambda (x) (* 10 x)))
  #<FUNCTION :LAMBDA (X) (* 10 X)>
> (setq gname #'(lambda (x) (cons x 'x)))
  #<FUNCTION :LAMBDA (X) (CONS X 'X)>

> '(1 . nil)
(1)

> (cons x 'x)
(5 . X)

> (cons '(1 2 3) 'x)
((1 2 3) . X)

> (mapcar gname lst)
((1 . X) (2 . X) (3 . X) (4 . X))

> (maplist gname lst)
(((1 2 3 4) . X) ((2 3 4) . X) ((3 4) . X) ((4) . X))
```

Use <u>cond</u> to write a function f1 as follows:

$$f1(x) = \begin{cases} -1 & x < 0 \\ 1 & 0 \le x < 10 \\ 2 & 10 \le x < 30 \\ 3 & x \ge 30 \end{cases}$$

```
(defun f1 (x)
      (cond (( < x 0) -1)
            (( < x 10) 1)
            ((< x 30) 2)
            (t 3)))
```

Use cond to write a function f2 with two arguments x and lst that does the following:

- If x is a negative number, it opens the file "data.txt", reads from it once and returns the read number (we'll assume it will be a number) as string containing the number as a float with 2 digits after the decimal point.

- If x is zero, it returns true

- If x is a positive number, it returns the first two elements of lst (we assume lst has at least two elements)

- If x is anything else, it returns nil

```
(defun f2a (x lst)
   (cond
      ((and (numberp x) (zerop x)))
      ((and (numberp x) (> x 0))
         (list (car lst) (cadr lst)))
      ((and (numberp x)(< x 0))
         (setq ins (open "data.txt" :direction :input))
         (format nil "~5,2f" (read ins))
         (close ins))
      (t nil)))

> (f2a 'a '(1 2 3))
NIL
> (f2a 0 '(1 2 3))
T
> (f2a 1 '(1 2 3))
(1 2)
> (f2a -1 '(1 2 3))
T                    → Does not work properly!
```

```
(defun f2b (x lst)
   (cond
      ((and (numberp x) (zerop x)))
      ((and (numberp x) (> x 0))
          (list (car lst) (cadr lst)))
      ((numberp x)
          (setq ins (open "data.txt" :direction :input))
          (setq y (read ins))
          (close ins)
          (format nil "~5,2f" y) ))

> (f2b -2 '(1 2 3))
"10.00"

> y
10              → global variable
```

Improve the code to not influence any global variables - <u>use aux variables</u>

```
(defun f2c (x lst &aux ins y)
   (cond
       ((and (numberp x) (zerop x)))
       ((and (numberp x) (> x 0))
            (list (car lst) (cadr lst)))
       ((numberp x)
            (setq ins (open "data.txt" :direction :input))
            (setq y (read ins))
            (close ins)
            (format nil "~5,2f" y) ))

> (setq y 3)
3
> (f2c -2 '(1 2 3))
"10.00"
> y
3
```

Improve the code to not influence any global variables - use let

```
(defun f2c (x lst)
   (let (ins y)
   (cond
      ((and (numberp x) (zerop x)))
      ((and (numberp x) (> x 0))
          (list (car lst) (cadr lst)))
      ((numberp x)
          (setq ins (open "data.txt" :direction :input))
          (setq y (read ins))
          (close ins)
          (format nil "~5,2f" y) )))
```

Improve the code to not influence any global variables - use lambda

```
(defun f2c (x lst)
   ((lambda (ins y)
   (cond
      ((and (numberp x) (zerop x)))
      ((and (numberp x) (> x 0))
          (list (car lst) (cadr lst)))
      ((numberp x)
          (setq ins (open "data.txt" :direction :input))
          (setq y (read ins))
          (close ins)
          (format nil "~5,2f" y) )) nil nil))
```

If f3 is defined as follows, how would LISP respond to the following?

```
(defun f3 (lst n p)
      (do   ((tlst  lst  (cdr tlst))
             (rslt '(0 . nil) (cons (car tlst) rslt))
             (i (1- n) (1- i)))
          ((zerop i) (cond ((zerop p) rslt)
                           (t n)))
          (if (null tlst) (return "Error"))))
```

```
> (f3 '(1 2 3) 3 0)
(2 1 0)
> (f3 '(1 2 3) 3 1)
3
> (f3 '(1 2 3) 5 1)
"Error"
```

```
> (f3 '(1 2 3) 5 0)
"Error"
> (f3 '(1 2 3) 4 0)
(3 2 1 0)
```

What if we have a do* instead of do?

```
(defun f3b (lst n p)
      (do*  ((tlst  lst  (cdr tlst))
             (rslt '(0 . nil) (cons (car tlst) rslt))
             (i (1- n) (1- i)))
          ((zerop i) (cond ((zerop p) rslt)
                           (t n)))
          (if (null tlst) (return "Error"))))
```

```
> (f3b '(1 2 3) 3 0)
(3 2 0)
> (f3b '(1 2 3) 3 1)
3
> (f3b '(1 2 3) 5 1)
"Error"
```

```
> (f3b '(1 2 3) 5 0)
"Error"
> (f3b '(1 2 3) 4 0)
(NIL 3 2 0)
```

Alonzo Church has defined the natural numbers in lambda calculus (known as the Church numerals) as follows:

```
0 := λfx.x
1 := λfx.f x
2 := λfx.f (f x)
3 := λfx.f (f (f x))
```

Show that if PLUS is defined as

```
PLUS := λmnfx.m f (n f x)
```

then adding (PLUS) two and one is equivalent to three.

```
PLUS 2 1 =
(λmnfx.m f (n f x)) 2 1 →β
((λnfx.m f (n f x))[m := 2]) 1 =
(λnfx.2 f (n f x)) 1 =
(λnfx.(λfx.f (f x)) f (n f x)) 1 →β
(λnfx.((λx.f (f x)) [f:=f]) (n f x)) 1 =
(λnfx.(λx.f (f x)) (n f x)) 1 →β
(λnfx.(f (f x))[x:= (n f x)]) 1 =
(λnfx.f (f (n f x))) 1 →β
 (λfx.f (f (n f x))) [n := 1] =
  λfx.f (f (1 f x))  =
  λfx.f (f ((λfx.f x) f x)) →β
  λfx.f (f ( ((λx.f x)[f:=f]) x)) =
  λfx.f (f ((λx.f x) x)) →β
  λfx.f (f ((f x) [x:=x])) =
  λfx.f (f (f x)) which is equivalent to 3
```

[ref: CSE3401 Summer 2009 Assignment #2]

Write a recursive function COMPRESS and DECOMPRESS that takes a list as a parameter and replaces any consecutive occurrence of elements with the element and its count. For example:

```
> (compress `(a a a b b x 2 2))
(a 3 b 2 x 1 2 2)

> (decompress `(a 3 b 2 x 1 2 2))
 (a a a b b x 2 2)
```

```
> (compress '(a a a b b x 2 2))
(a 3 b 2 x 1 2 2)
```

```
(defun readsame (plst)
   (do ((tlst plst (cdr tlst))
        (n 0 (1+ n)))
       ((or (null tlst)
            (not (equal (car plst) (car tlst))))
        (return (list (list (car plst) n)  tlst)) )))

(defun compress (lst)
  (cond ((null lst) nil)
        (t (let ((rec (readsame lst)))
             (append (car rec) (compress (cadr rec))))) ))
```

```
> (decompress '(a 3 b 2 x 1 2 2))
 (a a a b b x 2 2)
```

```
(defun writesame (a n)
   (do    ((rslt  nil  (cons a rslt))
           (i n (1- i)))
       ((zerop i) rslt)))

(defun decompress (lst)
   (cond ((null lst) nil)
          (t (append (writesame (car lst) (cadr lst))
                     (decompress (cddr lst)))) ))
```

Write a function that creates a sequence of bits (0 or 1) of length len:

```
(defun rndX (len)
      (do ( (i len (1- i))
            (x nil (cons (random 2) x)))
         ((zerop i) x)))
```

Convert a sequence of bits to its decimal equivalent:

```
(defun reverse (lst)
      (do ( (x lst (cdr x))
            (result nil (cons (car x) result)))
          ((null x) result)))


(defun x2Dec (x)
   (do ((tx (reverse x) (cdr tx))
        (exp 0  (1+ exp))
        (dec 0  (+ dec (* (car tx) (expt 2 exp)))))
      ((null tx) dec)))
```

Write a function that finds the length of a sequence and use that to re-write the previous function w/o reversing:

```
(defun getlength (lst)
    (do ((tlst lst (cdr tlst))
         (n 0 (1+ n)))
      ((atom tlst) n)))

(defun x2Dec2 (x)
   (do ((tx  x (cdr tx))
        (exp (1- (getlength x)) (1- exp))
        (dec 0 (+ dec (* (car tx) (expt 2 exp)))))
      ((null tx) dec)))
```

Write a function that inverts a random bit in a sequence with a given probability.

```
(defun invert@loc (lst loc)
   (if (zerop loc)
       (cons (mod (1+ (car lst)) 2)
             (cdr lst))
       (cons (car lst)
             (invert@loc (cdr lst) (1- loc)))))

(defun mutation (x pm)
   (if (< (random 1.0) pm)
       (invert@loc x (random (getlength x)))
       x ))
```