Principles of Programming, Fall 2009
Practice 2
Recursive Fuction, First Class Function, Pair and List

Woosuk Lee, Suwon Jang, Sungkeun Cho
Programming Research Lab.@SNU

September 14, 2009

Recursive function Write down the substitution process of the procdedure. Does it work what you expect?

```
(define sum (lambda (x) (if (= x 1) 1 (+ x (sum (- x 1))))))
(sum 1)
(sum 5)
(sum 0)
```

Function as Argument Try

```
(define apply5 (lambda (f) (f 5)))
(apply5 square)
```

Pair and list Try

```
(cons 1 2)
(car (cons 1 2))
(cdr (cons 1 2))
(car (cons (cons 1 2) 3))
(cdr (cons (cons 1 2) 3))
(car (cons 1 (cons 2 3)))
(cdr (cons 1 (cons 2 3)))
()
null
(cons 'foo ())
```

```
(cons 'foo (cons 'bar ()))
(list)
(list 'foo)
(list 'foo 'bar)
'(foo bar)
(list 1 2 3)
'(1 2 3)
(null? ())
(null? '(1 2 3))
```

Exercise

1. Define the factorial fuction called `fact` that takes a number as its argument and computes factorials. For examples,

   ```
   (fact 3)
   6

   (fact 0)
   1
   ```

2. Define a procedure called combination that takes two numbers as its argument, namely n, m, and computes ${}_nC_m$. For examples,

   ```
   (combination 4 2)
   6

   (combination 9 4)
   126
   ```

   Write two definitions of *combination* - one that uses above *fact* fuction and one that does not use multiplication($\times$) or division($/$).[1]

3. Define a procedure called *sigma* that takes two numbers and a fuction as its arguments, namely a, b, f respectively, and computes the following.

   $$\sum_{n=a}^{b} f(n) = f(a) + f(a+1) + \cdots + f(b)$$

   For example,

   ```
   (define (f n) (* n n))
   (sigma 1 3 f)
   14
   ```

[1] *Hint.* Consider *Pascal's triangle.*

4. Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure *compose* that implements composition. For example,

```
(define (square x) (* x x))
(define (inc x) (+ x 1))
((compose square inc) 6)
49
```

5. Using *cond*, *car*, *cdr* primitives, define a procedure called *nth* that takes a integer and list as its argument returning nth element of list. For example

```
(nth 0 (list 1 2 3))
;1

(nth 10 (list 1 2 3))
;error : out of bound!
```

6. The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
(square-list (list 1 2 3 4))
;(1 4 9 16)
```

7. We can make the procedure `square-list` easily by using the `map` procedure.

```
(define (square-list items))
  (map square items))
```

   Define a procedure `my-map` that acts like `map`.

```
(my-map square (list 1 2 3 4))
;(1 4 9 16)

(my-map abs (list -1 -2 -3 -4))
;(1 2 3 4)
```

8. Define a procedure `fold` that takes a list, a function and an arbitary value as its arguments, and computes following.

$$
\begin{aligned}
(\texttt{fold}\ f\ c\ '()) &= c \\
(\texttt{fold}\ f\ c\ '(a_1\ \cdots\ a_n)) &= (f (f (f\ c\ a_1)\ a_2)\ \cdots a_n)
\end{aligned}
$$

   For examples,

```
(fold + 0 (1 2 3 4 5))
;15
```

9. (*Optional*) Assume that you have three pegs and a set of disks, all of different diameters, with holes in them (so that they can slide onto the pegs). Start with all the disks on a single peg, in order of size (with the smallest on top). The object of the puzzle[2] is to move the pile of disks to a specified peg, by moving one disk at a time. A legal move consists of taking the top disk from any peg and putting it on either of the other two pegs; but a disk may never be placed on top of a disk that is smaller than itself.

   We will write a procedure *move-tower* that takes four arguments - the number of disks in the pile, the peg the disks are on, the peg the disks should be moved to, and the extra peg - and prints the sequence of moves. For example, consider moving three disks from peg 1 to peg 3 by evaluating *(move-tower 3 1 3 2)*. This should print:

   ```
   move top disk from 1 to 3
   move top disk from 1 to 2
   move top disk from 3 to 2
   move top disk from 1 to 3
   move top disk from 2 to 1
   move top disk from 2 to 3
   move top disk from 1 to 3
   ```

   You can use following procedure that takes two arguments - the peg the disks are on, the peg the disk should be moved to - and prints one step of moves.

   ```
   (define (print-move from to)
     (newline)
     (display "move top disk from ") (display from)
     (display " to ") (display to))
   ```

[2]This puzzle is well known by 'Hanoi tower problem'