Principles of Programming, Fall 2011

Practive 2*

Recursive Fuction, Data Abstraction

Programming Research Lab,@SNU

Seungjung Lee, Youngseok Lee

September 26, 2011

1. Pair

   Pair can be implemented by `cons`. Function `car` returns first element of pair , and `cdr` returns second element of one.

   ```
   (define p (cons 1 2))
   (car p)
   (cdr p)
   ```

2. List

   Actually, list is structure that is represented by weaved pair like chain. Pair which is made by `cons` can implement list.

   (a) List made by `cons`, List made by `list`

   ```
   (cons 1 ())
   (list 1)
   (cons 1 (cons 2 (cons 3 (cons 4 ()))))
   (list 1 2 3 4)
   (equal? (list 1 2) (cons 1 (cons 2 ())))
   ```

   (b) Is it list?

   ```
   (list? ())
   (list? (list 'a 'b 'c))
   (list? (cons 1 ()))
   (list? 123)
   (list? (cons 1 2))
   ```

*translated by Youngseok Lee.

3. Function about list

```
(null? ())
(length '(1 2 3 4))
(append '(1 2) '(3 4 5))
(reverse '(1 2 3))
(list-ref '(1 2 3 4 5) 2)
(list-ref '(1 2 3 4 5) 5)
(list-tail '(1 2 3 4 5) 2)
(map (lambda (x) (* x x)) '(1 2 3 4))
(map car '( (1 2) (3 4) (5 6)))
(filter odd? '(1 2 3 4 5))
(for-each (lambda (x) (display x)) '(1 2 3 4))
```

Practice

1. Define a procedure called `my-filter` that is identical to `filter`. (Do not use `filter` procedure above.)

2. Define a procedure called `my-reverse` that is identical to `reverse`. (Do not use `reverse` and `append` procedure above.)

3. 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[1] 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
```

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

```
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))
```

4. Make structure which can handle set. It is one of the easiest method that list of distinctive elements represents set. Now define a function `make-set` which takes list of elements and produces set. Using `equal?` procedure, compare between elements. For examples,

```
(make-set '(1 1 2 3 1 2 3))
(1 2 3)
(make-set '())
()
(make-set (list 'these 'are 'symbols))
(these are symbols)
(make-set (list (list 1 2) (list 2 3 4) (list 1 2)))
((1 2) (2 3 4))
```

If you made set, you should need a procedure called `is-member?` which takes element and set and returns a boolean value. For examples,

```
(is-member? 1 (make-set '(1 2 3)))
#t
(is-member? '(1 2) (make-set (list (list 1 2) (list 3 4 5))))
#t
```

Now define basic operations. Define two fuctions `union-set` and `intersection-set`. These functions take two sets and output union and intersection. For examples,

```
(union-set (make-set '(1 2 3)) (make-set '(2 3 4)))
(1 2 3 4)
(intersection-set (make-set (list 'a 'b 'c)) (make-set '(1 2 3)))
()
```

And then, define a little more complicated procedure called `cartesian-product`. This procedure takes two sets and outputs product set. Product set means set of pairs which contains elements of each input set. For examples,

```
(cartesian-product (make-set '()) (make-set '(1 2)))
()
(cartesian-product (make-set '(1 2)) (make-set ('black 'white)))
((1 . black) (1 . white) (2 . black) (2 . white))
```