

# Principles of Programming, Fall 2009

## Practice 3

### More on Lists and Recursive Functions

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

September 21, 2009

**Primitive List operations** In fact, list is regarded as primitive type. Scheme standard offers some primitive operations for list management.

```
empty
(list 1 2 3)
(list? '(1 2 3))
(length '(1 2 3))
(append '(1 2) '(3 4))
(reverse '(1 2 3))
(list-tail '(1 2 3 4) 2)
(list-ref '(1 2 3 4) 2)
(map (lambda (x) (+ x 1)) '(1 2 3 4))
(for-each (lambda (x) (display x)) '(1 2 3 4))
```

#### Exercise

1. We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is  $(1\ 2\ 3)$ , then the set of all subsets is  $((\ )\ (3)\ (2)\ (2\ 3)\ (1)\ (1\ 3)\ (1\ 2)\ (1\ 2\ 3))$ . Complete the following definition of a procedure that generates the set of subsets of a set.

```
(define (subsets s)
  (if (null? s)
      (list null)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

2. Evaluating a polynomial in  $x$  at a given value of  $x$  can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\cdots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0$$

In other words, we start with  $a_n$ , multiply by  $x$ , add  $a_{n-1}$ , multiply by  $x$ , and so on, until we reach  $a_0$ . Fill in the following template to produce a procedure that evaluates a polynomial using *Horner's rule*. Assume that the coefficients of the polynomial are arranged in a sequence, from  $a_0$  through  $a_n$ .

```
(define (horner-eval x coefficient-sequence)
  (foldr (lambda (this-coeff higher-terms) <??>)
    0
    coefficient-sequence))
```

For example, to compute  $1 + 3x + 5x^3 + x^5$  at  $x = 2$  you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
;79
```

3. Define a procedure called *my-reverse* that takes a list as its argument and returns a list of the same elements in reverse order.
4. Modify your *my-reverse* procedure to produce a *deep-reverse* procedure that takes a list as argument and returns as its value the list with its elements reverse and with all sublists deep-reverse as well. For example,

```
(define x (list (list 1 2) (list 3 (list 4 5))))

(reverse x)
;((3 (4 5)) (1 2))

(deep-reverse x)
;(((5 4) 3) (2 1))
```

5. Define a procedure called *my-filter* that takes a list and a predicate function as its arguments and produces a new list containing exactly those elements of the original list for which the given predicate returns the boolean value true. (Do not use *filter* procedure.) For example,

```
(my-filter (list 1 3 2 4) (lambda (x) (> x 2)))
;(3 4)

(my-filter (list 1 3 2 4) (lambda (x) (= x 1)))
;(1)
```

6. Define a procedure *my-append* combines two lists. (Do not use *append* procedure.) For example,

```
(my-append (list 1 2 3) (list 4 5 6))  
;(1 2 3 4 5 6)
```

7. The procedures `+`, `*`, and `list` take arbitrary numbers of arguments. One way to define such procedures is to use `define` with *dotted-tail notation*. In a procedure definition, a parameter list that has a dot before the last parameter name indicates that, when the procedure is called, the initial parameters (if any) will have as values the initial arguments, as usual, but the final parameter's value will be a `list` of any remaining arguments. For instance, given the definition

```
(define (f x y . z) <body>)
```

the procedure `f` can be called with two or more arguments. If we evaluate

```
(f 1 2 3 4 5 6)
```

then in the body of `f`, `x` will be 1, `y` will be 2, and `z` will be the list `(3 4 5 6)`. Given the definition

```
(define (g . w) <body>)
```

the procedure `g` can be called with zero or more arguments. If we evaluate

```
(g 1 2 3 4 5 6)
```

then in the body of `g`, `w` will be the list `(1 2 3 4 5 6)`.

Use this notation to write a procedure *same-parity* that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

```
(same-parity 1 2 3 4 5 6 7)  
;(1 3 5 7)
```

```
(same-parity 2 3 4 5 6 7)  
;(2 4 6)
```

And define more generic procedure *append-many* that combines arbitrary number of lists by using the *my-append* procedure that you defined above.

```
(append-many '(1 2 3) '(4 5 6) '(7 8 9))  
;(1 2 3 4 5 6 7 8 9)
```