Assignment 2: Scheme Finger Exercises — CS351—Fall 2008

Due 09:00 09-Oct-2008. Email text file of solutions to: `barak+cs351-hw1@cs.nuim.ie`.

1. Scan the R^{5}RS manual and find a few functions that are generalized in interesting ways. Explain why you think they were generalized in that way. (Examples are `<=` or `-`, which are generalized to accept a number of arguments other than 2.)

> **Solution:** Many functions that one would expect to be binary and associative instead take as many arguments as one might wish. Examples: `append`, `+`, `*`, `string-append`. In the one-argument case these all return that argument, so `(*` x`)` $\Rightarrow x$, etc. And in the zero-argument case they return the identity element, so `(append)` $\Rightarrow$ `()`, `(+)` $\Rightarrow 0$, `(*)` $\Rightarrow 1$, `(string-append)` $\Rightarrow$ `""`.
>
> Others functions which are usually binary are generalized idiosyncratically, so in the one-argument case `-` gives `(-` x`)` $\Rightarrow -x$, while in the two-or-more-arguments case it gives `(-` x y`)` $\Rightarrow x - y$ or in general `(-` x y_1 y_2 $\ldots$ y_n`)` $\Rightarrow x - y_1 - y_2 \cdots - y_n$. Similarly `(/` x`)` $\Rightarrow 1/x$ while `(/` x y_1 $\ldots$ y_n`)` $\Rightarrow x/(y_1 \cdots y_n)$.
>
> An exception is the `list*` function. For some reason, `cons` was not simply generalized; instead the apparently redundant `list*` was introduced, which when given n arguments "conses" the first $n-1$ successively onto the n-th, so `(list*` x_1 x_2 $\ldots$ x_{n-1} x_n`)` $=$ `(cons` x_1 `(cons` x_2 $\cdots$ `(cons` x_{n-1} x_n`)`$\cdots$`))`. This is a strict generalization of `cons`. Note that `(list*` x_1 x_2 $\ldots$ x_{n-1} x_n`)` $=$ `(append (list` x_1 $\ldots$ x_{n-1}`)` x_n`)`.
>
> Some comparison predicates, like `=` and `<`, are generalized from two arguments to two-or-more arguments, for instance `(<` x_1 x_2 $\ldots$ x_n`)` $\Rightarrow x_1 < x_2 \vee x_2 < x_3 \vee \cdots \vee x_{n-1} < x_n$. Others, like `equal?` and `eq?`, are not. None are generalized to zero or one argument (returning true in those cases), even though this would seem natural.

2. Define `list-sum-squares` which takes a list of numbers and returns the sum of their squares.

   Example: `(list-sum-squares (list 1 4 1))` $\Rightarrow$ `18`

> **Solution:**
>
> ```
> (define list-sum-squares
>   (lambda (nums)
>     (if (null? nums)
>         0
>         (+ (expt (car nums) 2)
>            (list-sum-squares (cdr nums))))))
> ```
>
> or
>
> ```
> (define list-sum-squares
>   (lambda (nums)
>     (if (null? nums)
>         (+)
>         (+ (expt (car nums) 2)
>            (list-sum-squares (cdr nums))))))
> ```

3. Define `list-product-sqrts` which takes a list of non-negative numbers and returns the product of their square roots.

   Example: `(list-product-sqrts (list 4 9))` ⇒ `6`

Solution:

```
(define list-product-sqrts
  (lambda (nums)
    (if (null? nums)
        1
        (* (sqrt (car nums))
           (list-product-sqrts (cdr nums))))))
```

or

```
(define list-product-sqrts
  (lambda (nums)
    (if (null? nums)
        (*)
        (* (sqrt (car nums))
           (list-product-sqrts (cdr nums))))))
```

or

```
(define list-product-sqrts
  (lambda (nums)
    (apply * (map sqrt nums))))
```

4. Define `set-union` which takes two lists representing sets and returns a list representing their union. (Ordering is unimportant.)

   Example: `(set-union (list 1 2 3 4) (list 6 4 8 2))` ⇒ `(1 2 3 4 6 8)` (or `(3 1 8 4 2 6)` or any other rearrangement of the elements.)

Solution:

```
(define set-union
  (lambda (s1 s2)
    (cond ((null? s1) s2)
          ((member (car s1) s2)
           (set-union (cdr s1) s2))
          (else (cons (car s1)
                      (set-union (cdr s1) s2))))))
```

5. Define `set-intersection` which takes two lists representing sets and returns a list representing their intersection.

   Example: `(set-intersection (list 3 1 2 4) (list 4 2 8 6))` ⇒ `(2 4)` (or `(4 2)`)

Solution:

```
(define set-intersection
  (lambda (s1 s2)
    (cond ((null? s1) '())
          ((member (car s1) s2)
           (cons (car s1)
                 (set-intersection (cdr s1) s2)))
          (else (set-intersection (cdr s1) s2)))))
```

6. Define `set-disjoint?` which takes two lists representing sets and returns true iff the sets are disjoint.

Solution:

```
(define set-disjoint?
  (lambda (s1 s2)
    (null? (set-intersection s1 s2))))
```

or

```
(define set-disjoint?
  (lambda (s1 s2)
    (or (null? s1)
        (and (not (member (car s1) s2))
             (set-disjoint (cdr s1) s2)))))
```

7. Define `filter-numbers` which takes a list representing a set and returns a list representing a set containing only those members that are numbers, i.e., that pass the `number?` predicate.

   Example: `(filter-numbers '(1 one 2 two foo zero 22/7 0))` ⇒ `(1 2 22/7 0)` (or a permutation thereof.)

Solution:

```
(define filter-numbers
  (lambda (lis)
    (cond ((null? lis) lis)
          ((number? (car lis))
           (cons (car lis) (filter-numbers (cdr lis))))
          (else (filter-numbers (cdr lis))))))
```

8. Define `set-equal?` which takes two lists representing sets and returns true iff they represent the same set.

   Example: `(set-equal? '(1 2 3) '(2 1 3))` ⇒ `#t`

   Example: `(set-equal? '(1 2 () 3) '(2 1 3))` ⇒ `#f`

Solution:

```
(define set-equal?
  (lambda (s1 s2)
    (and (set-subset? s1 s2)
         (set-subset? s2 s1))))

(define set-subset?
  (lambda (s1 s2)
    (or (null? s1)
        (and (member (car s1) s2)
             (set-subset? (cdr s1) s2)))))
```

9. Define `deep-member?` which takes a symbol and an s-expression and returns true iff the symbol occurs in the given s-expression, perhaps very deeply nested.

   Example: `(deep-member 'foo '(a b (c (d e foo g)) h))` ⇒ `#t`

   Example: `(deep-member 'foo '(a b (c (d e bar g)) h))` ⇒ `#f`

   Solution:

   ```
   (define deep-member
     (lambda (a s)
       (or (equal? a s)
           (and (pair? s)
                (or (deep-member a (car s))
                    (deep-member a (cdr s)))))))
   ```

10. *Optional:* If you encountered any problems with the assignment, or have any comments on it, or other comments or suggestions, I would appreciate hearing them. As practice for working in industry, where weekly reports are not unusual, please embody these in a brief (1–3 page) typed report.

    Solution: This is my favourite class ever. The only suggestion I would make is to give longer and harder assignments, and assign more of them, so I can enjoy more practice programming, which I so love.

Hint: use recursion and make your base cases as simple as possible.

Honor Code: You may discuss these with others, but please write your answers by yourself and without reference to communal notes. In other words, your answers should be *from your own head.*