CS 2500 Exam 2 HONORS Solutions – Fall 2013

Problem 1 Design the function `concat`, which consumes a list of lists and appends them all to produce a single list. Give `concat` its most general signature and define it using a loop function. You may not use `append` or `apply`. 8 POINTS

```
;; [List-of [List-of X]] -> [List-of X]
;; Concatenate all the lists in lolox into a single list
(define (concat lolox)
  (foldr (lambda (lox ans)
           (foldr cons ans lox))
         empty lolox))


(check-expect (concat (list)) '())
(check-expect (concat (list '())) '())
(check-expect (concat (list '(1 2) '())) '(1 2))
(check-expect (concat (list '() '(3 4))) '(3 4))
(check-expect (concat (list (list 1 2) (list 3) (list 4 5)))
              (list 1 2 3 4 5))
```

Problem 2 A *sequence* represents a series of values. Sequences may be finite or infinite. In this problem, we'll work with infinite sequences. 18 POINTS

Here are three examples of infinite sequences:

index	0	1	2	3	…
positive integers	`1`	`2`	`3`	`4`	…
even natural numbers	`0`	`2`	`4`	`6`	…
lists of `'a`	`'()`	`'(a)`	`'(a a)`	`'(a a a)`	…

Here is a data definition for representing infinite sequences:

```
;; A [Sequence X] is a [Natural -> X]
;; interpretation: when the function is applied to an
;; index (a Natural), it gives back the element at
;; that index.
```

Here is an example of a `[Sequence Natural]`, the even natural numbers:

```
(define even-nats (lambda (i) (* 2 i)))
```

Here is a convenient function for producing a list with the first `n` elements of an infinite sequence:

```
;; seq->listn : [Sequence X] Natural -> [List X]
;; Build a list with the first n elements of the
;; sequence s
(define (seq->listn s n)
  (map s (build-list n (lambda (x) x))))
```

For example,

```
> (seq->listn even-nats 10)
(list 0 2 4 6 8 10 12 14 16 18)
```

You may use `even-nats` and `seq->listn` for tests, but they should not be used otherwise.

(a) (8 pts) Design the following functions:

- `seq-head`, which consumes a sequence `s` and returns its 0th element.
- `seq-rest`, which consumes a sequence `s` and returns a sequence with all but the 0th element of `s`.

```
;; seq-head : [Sequence X] -> X
;; Get the 0th element in the sequence
(define (seq-head s)
  (s 0))

(check-expect (seq-head even-nats) 0)


;; seq-rest : [Sequence X] -> [Sequence X]
;; Produce a sequence with all but the 0th element of s
(define (seq-rest s)
  (lambda (i) (s (add1 i))))

(check-expect (seq->listn (seq-rest even-nats) 5)
              (list 2 4 6 8 10))
```

(b) (10 pts) A *series* for a sequence `s` gives the sums of the elements in `s`. More precisely, adding the 0th through ith elements of an infinite sequence `s` forms the ith element of another infinite sequence, called a series.

For example, the series for the sequence of positive integers `1`, `2`, `3`, `4`, `...` is: `1`, `3`, `6`, `10`, `....`

Design the function `seq->series`, which consumes a `[Sequence X]`, and a function for adding `X`s (with signature `[X X -> X]`), and produces a series for the given sequence.

```
;; seq->series : [Sequence X] [X X -> X] -> [Sequence X]
;; Given a sequence s of Xs, and a function addx that can
;; add two Xs, produce the series for sequence s.
(define (seq->series s addx)
  (lambda (i)
    (local ((define (sumx i)
              (cond [(zero? i) (s i)]
                    [else (addx (s i)
                                (sumx (sub1 i)))])))
      (sumx i))))

(check-expect (seq->listn (seq->series even-nats +) 5)
              (list 0 2 6 12 20))
```

```
Alternative solution:
```

```
(define (seq->series s addx)
  (lambda (i)
    (cond [(zero? i) (s i)]
          [else (addx (s i)
                      ((seq->series s addx) (sub1 i)))])))
```

Problem 3 Consider the following data definition for *finite* sequences: 14 POINTS

```
;; A [Maybe X] is one of:
;; - 'undef
;; - X

;; A [FiniteSeq X] is a [Sequence [Maybe X]]
;; Constraint: there exists some index i>0 such that
;;   - no elements at indices [0,i) equal 'undef
;;   - all elements at indices >= i equal 'undef
```

Informally, the above data definition allows us to represent a finite sequence `1`, `2`, `3` as the infinite sequence `1`, `2`, `3`, `'undef`, `'undef`, `'undef`, `...`

(a) (2 pts) Define `even-nats-4to8`, an instance of `[FiniteSeq Natural]` that represents the sequence of even natural numbers in the range [4,8]—that is, the finite sequence `4`, `6`, `8`.

```
Either of the following is okay.

(define even-nats-4to8
  (lambda (i) (cond
                [(= i 0) 4]
                [(= i 1) 6]
                [(= i 2) 8]
                [else 'undef])))

(define even-nats-4to8
  (lambda (i) (if (< i 3)
              (+ (* 2 i) 4)
              'undef)))
```

(b) (12 pts) Design the function `fs-length`, which consumes a finite sequence and two natural numbers `lo` and `hi` and produces the length of the finite sequence. Assume that `lo < hi` and that there exists an index `i` in the range `[lo,hi)` such that the element at index `i+1` is `'undef` but the element at index `i` is not.

For example, for the finite sequence `even-nats-4to8` that you defined in part (a):

```
> (fs-length even-nats-4to8 0 100)
3
```

To get credit for this problem, you will need to use an efficient generative recursion design.

```
;; fs-length : [FiniteSeq X] Natural Natural -> Natural
;; Compute the length of the finite sequence fs, assuming
;; that the length is a number in the range (lo,hi].
;; Assume: lo < hi
;; Assume: there exist an index i in [lo,hi) such that
;; element at index i+1 is 'undef while the element at
;; index i is not.

;; Generative recursion
;; HOW: Determine midpoint between lo and hi. If element
;; at midpoint is 'undef then length between lo and mid;
;; otherwise length between mid and hi.
;;
;; TERMINATES for all possible finite sequences because
;; the recursive calls are guaranteed to receive smaller
;; sequences than the given s.

(define (fs-length s lo hi)
  (cond [(= (add1 lo) hi) hi]
        [else
         (local ((define mid (quotient (+ lo hi) 2))
                 (define s@mid (s mid)))
           (cond [(and (symbol? s@mid) (symbol=? s@mid 'undef))
                  (fs-length s lo mid)]
                 [else
                  (fs-length s mid hi)]))]))

(check-expect (fs-length even-nats-4to8 2 3) 3)
(check-expect (fs-length even-nats-4to8 0 1000000) 3)
```