Section Solution: All Things Scheme

Problem 1: Building Subsets of a Certain Size

;; Function: k-subsets
;; -------------------
;; k-subsets constructs a list of all those subsets
;; of the specified set whose size just happens to equal k.
;;
;; Examples: (k-subsets '(1 2 3 4) 2) -> ((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))
;;           (k-subsets '(1 2 3 4 5 6) 1) -> ((1) (2) (3) (4) (5) (6))
;;           (k-subsets '(a b c d) 0) -> ()
;;           (k-subsets '(a b d d) 5) -> ()
(define (k-subsets set k)
  (cond ((eq? (length set) k) (list set))
        ((zero? k) '(()))
        ((or (negative? k) (> k (length set))) '())
        (else (let ((k-subsets-of-rest (k-subsets (cdr set) k))
                   (k-1-subsets-of-rest (k-subsets (cdr set) (- k 1))))
                 (append (map (lambda (subset)
                               (cons (car set) subset))
                            k-1-subsets-of-rest)
                         k-subsets-of-rest)))))

Problem 2: Up Down Permutations

a.

;; Function: is-up-down?
;; ---------------------
;; Returns true if and only if the specified list is an up-down list
;; according to the specified predicate.
;;
;; Examples: (is-up-down? () <) -> #t
;;           (is-up-down? '(1) <) -> #t
;;           (is-up-down? '(1 2 2 3) <) -> #f
;;           (is-up-down? '(1 6 2 4 3 5) <) -> #f
;;           (is-up-down? '(1 6 2 4 3 5) >) -> #t
;;           (is-up-down? '(4 8 3 5 1 7 6 2) <) -> #t
(define (is-up-down? ls comp)
  (or (null? ls)
      (null? (cdr ls))
      (and (comp (car ls) (cadr ls))
           (is-up-down? (cadr ls) (lambda (one two) (comp two one))))))))

Notice the above version uses tradition car-cdr recursion, but constructs an anonymous binary predicate out of the original by just switching the roles of the two arguments. (This is how I got > and not >= from <.) Of course, the disadvantage of this implementation is that layers of anonymous lambda build up around the original for arbitrarily large lists. The solution here is to employ some arm’s length recursion by ensuring that the first three
elements are low-high-low and then recurring on the cdr of the cdr using the original predicate.

\[
\begin{align*}
&\text{(define (is-up-down? ls comp)} \\
&\quad \text{(or (null? ls)} \\
&\quad \quad \text{(null? (cdr ls))} \\
&\quad \quad \text{(and (comp (car ls) (cadr ls))}\) \\
&\quad \qquad \text{(or (null? (cddr ls))} \\
&\quad \quad \quad \text{(and (comp (caddr ls) (cadr ls)) (is-up-down? (cddr ls) comp)))))}
\end{align*}
\]

You could even use mutual recursion and define a sister is-down-up? function. In fact, the second half of this problem does exactly that. 😊

b.

```scheme
;; Function: up-down-permute
;; --------------------------
;; up-down-permute generates all those permutations of a list that
;; just happen to be up-down permutations.
;;
;; Examples: (remove 3 '(1 2 3 4 5 4 3 2 1)) -> (1 2 4 5 4 2 1)
;; (up-down-permute '()) -> ((1 2 3 4 5 4 3 2 1))
;; (up-down-permute '(1)) -> ((1))
;; (up-down-permute '(1 2)) -> ((1 2))
;; (up-down-permute '(1 2 3)) -> ((1 3 2) (2 3 1))
;; (up-down-permute '(1 2 3 4 5)) ->
;;     ((1 3 2 5 4) (1 4 2 5 3) (1 4 3 5 2) (1 5 2 4 3) (1 5 3 4 2)
;;     (2 3 1 5 4) (2 4 1 5 3) (2 4 3 5 1) (2 5 1 4 3) (2 5 3 4 1)
;;     (3 4 1 5 2) (3 4 2 5 1) (3 5 1 4 2) (3 5 2 4 1)
;;     (4 5 1 3 2) (4 5 2 3 1))
```

```scheme
(define (construct-permute-generator comp inverted-permute ls) 
  (lambda (number) 
    (apply append 
      (map (lambda (permutation) 
        (if (comp number (car permutation)) 
          (list (cons number permutation)) 
          '())) 
      (inverted-permute (remove ls number))))))
```

```scheme
(define (up-down-permute ls) 
  (if (<= (length ls) 1) (list ls) 
    (apply append 
      (map (construct-permute-generator < down-up-permute ls) ls))))
```

```scheme
(define (down-up-permute ls) 
  (if (<= (length ls) 1) (list ls) 
    (apply append 
      (map (construct-permute-generator > up-down-permute ls) ls))))
```