Scheme and Haskell Assignment Solution

1. **Reverse**
   a. **Scheme**

```
(define (append list1 list2)
        (cond
              ((null? list1) list2)
              (else (cons (car list1)
                     (append (cdr list1) list2)))
        ))
(define (reverse list_a)
        (if (null? (cdr list_a))
              list_a
              (append (reverse (cdr list_a)) (list (car list_a)))))
```

   b. **Haskell**

```
reverseList :: [a] -> [a]
reverseList [] = []
reverseList(x:xs) = (reverseList xs) ++ [x]
```

2. **Union**
   a. **Scheme**

```
(define (member atm list_a)
        (cond
              ((null? list_a) #F)
              ((eq? atm (car list_a)) #T)
              (else (member atm (cdr list_a)))
        )
)
```

```
(define (union setlist_1 setlist_2)
        (cond
                ((null? setlist_1) setlist_2)
                ((member (car setlist_1) setlist_2) (union (cdr setlist_1)
                setlist_2))
                (else (cons (car setlist_1) (union (cdr setlist_1) setlist_2)))
        )
)
```

b. **Haskell**

```
unionList (x:xs) (y:ys) = case compare x y of
LT -> x : unionList xs (y:ys)
EQ -> x : unionList xs ys
GT -> y : unionList (x:xs) ys
unionList xs [] = xs
unionList [] ys = ys
```

3. **Sort**

a. **Scheme**

```
(define (insert atm list_a)
        (cond
                ((null? list_a) (cons atm '()))
                ((< atm (car list_a)) (cons atm list_a))
                (else (cons (car list_a) (insert atm (cdr list_a))))
        )
)
(define (ascending list_a)
        (if (null? list_a)
                '()
                (insert (car list_a) (ascending (cdr list_a)))
```

```
        )
)
```

b. Haskell

```
ascending :: [Int] -> [Int]
ascending [] = []
ascending (h:t) = ascending[b | b <- t, b <= h] ++ [h] ++ ascending[b | b <- t, b > h]
```

4. Max Min

a. Scheme

```
(define (max list_a)
        (cond
                ((null? list_a) ‘())
                ((null? (cdr list_a)) (car list_a))
                ((> (car list_a) (max (cdr list_a))) (car list_a))
                (else (max (cdr list_a)))
        )
)
(define (min list_a)
        (cond
                ((null? list_a) ‘())
                ((null? (cdr list_a)) (car list_a))
                ((< (car list_a) (min (cdr list_a))) (car list_a))
                (else (min (cdr list_a)))
        )
)
(define (minmax list_a) (cons (min list_a) (cons (max list_a) '())))
```

b. Haskell

```
ascending :: [Int] -> [Int]

ascending [] = []

ascending (h:t) = ascending[b | b <- t, b <= h] ++ [h] ++ ascending[b | b <- t, b > h]

minMax :: [Int] -> [Int]

minMax [] = []

minMax list = [head (ascending list), last (ascending list)]
```

5. Permutations

a. Scheme

```
(define (remove x lst)

    (cond

        ((null? lst) '())

        ((= x (car lst)) (remove x (cdr lst)))

        (else (cons (car lst) (remove x (cdr lst))))))

(define (permute lst)

    (cond

    ((= (length lst) 1) (list lst))

    (else (apply append (map (lambda (i) (map (lambda (j) (cons i j)) (permute (remove i lst))))

lst)))))
```

b. Haskell

```
permute :: [a] -> [[a]]

permute [] = [[]]

permute (x:xs) = foldr (++) [] (map (interleave [] x) (permute xs))

where

 interleave :: [a] -> a -> [a] -> [[a]]

 interleave xs x [] = [xs ++ [x]]
```

```
interleave xs x (y:ys) =
(xs ++ (x:y:ys)) :
(interleave (xs ++ [y]) x ys)
```