Exercise 1.14 [$\star\star$] Given the assumption $0 \le$ `n` $<$ *length*(`von`), prove that `partial-vector-sum` is correct.

There are many other situations in which it may be helpful or necessary to introduce auxiliary variables or procedures to solve a problem. Always feel free to do so.

1.2.4 Exercises

Getting the knack of writing recursive programs involves practice. Thus we conclude this section with a number of exercises.

Exercise 1.15 [$\star$] Define, test, and debug the following procedures. Assume that `s` is any symbol, `n` is a nonnegative integer, `lst` is a list, `v` is a vector, `los` is a list of symbols, `vos` is a vector of symbols, `slist` is an s-list, and `x` is any object; and similarly `s1` is a symbol, `los2` is a list of symbols, `x1` is an object, *etc*. Also assume that `pred` is a predicate, that is, a procedure that takes any Scheme object and returns either `#t` or `#f`. Make no other assumptions about the data unless further restrictions are given as part of a particular problem. For these exercises, there is no need to check that the input matches the description; for each procedure, assume that its input values are members of the specified sets.

To test these procedures, at the very minimum try all of the given examples. Also use other examples to test these procedures, since the given examples are not adequate to reveal all possible errors.

1. `(duple n x)` returns a list containing `n` copies of `x`.

   ```
   > (duple 2 3)
   (3 3)
   > (duple 4 '(ho ho))
   ((ho ho) (ho ho) (ho ho) (ho ho))
   > (duple 0 '(blah))
   ()
   ```

2. `(invert lst)`, where `lst` is a list of 2-lists (lists of length two), returns a list with each 2-list reversed.

   ```
   > (invert '((a 1) (a 2) (b 1) (b 2)))
   ((1 a) (2 a) (1 b) (2 b))
   ```

3. `(filter-in pred lst)` returns the list of those elements in `lst` that satisfy the predicate `pred`.

   ```
   > (filter-in number? '(a 2 (1 3) b 7))
   (2 7)
   > (filter-in symbol? '(a (b c) 17 foo))
   (a foo)
   ```

4. `(every? pred lst)` returns `#f` if any element of `lst` fails to satisfy `pred`, and returns `#t` otherwise.

```
> (every? number? '(a b c 3 e))
#f
> (every? number? '(1 2 3 5 4))
#t
```

5. `(exists? pred lst)` returns `#t` if any element of `lst` satisfies `pred`, and returns `#f` otherwise.

```
> (exists? number? '(a b c 3 e))
#t
> (exists? number? '(a b c d e))
#f
```

6. `(vector-index pred v)` returns the zero-based index of the first element of `v` that satisfies the predicate `pred`, or `#f` if no element of `v` satisfies `pred`.

```
> (vector-index (lambda (x) (eqv? x 'c)) '#(a b c d))
2
> (vector-ref '#(a b c)
    (vector-index (lambda (x) (eqv? x 'b)) '#(a b c)))
b
```

7. `(list-set lst n x)` returns a list like `lst`, except that the `n`-th element, using zero-based indexing, is `x`.

```
> (list-set '(a b c d) 2 '(1 2))
(a b (1 2) d)
> (list-ref (list-set '(a b c d) 3 '(1 5 10)) 3)
(1 5 10)
```

8. `(product los1 los2)` returns a list of 2-lists that represents the Cartesian product of `los1` and `los2`. The 2-lists may appear in any order.

```
> (product '(a b c) '(x y))
((a x) (a y) (b x) (b y) (c x) (c y))
```

9. `(down lst)` wraps parentheses around each top-level element of `lst`.

```
> (down '(1 2 3))
((1) (2) (3))
> (down '((a) (fine) (idea)))
(((a)) ((fine)) ((idea)))
> (down '(a (more (complicated)) object))
((a) ((more (complicated))) (object))
```

10. `(vector-append-list v lst)` returns a new vector with the elements of `lst` attached to the end of `v`. Do this without using `vector->list`, `list->vector`, and `append`.

```
> (vector-append-list '#(1 2 3) '(4 5))
#(1 2 3 4 5)
```

Exercise 1.16 [$\star\star$]

1. `(up lst)` removes a pair of parentheses from each top-level element of `lst`. If a top-level element is not a list, it is included in the result, as is. The value of `(up (down lst))` is equivalent to `lst`, but `(down (up lst))` is not necessarily `lst`.

```
> (up '((1 2) (3 4)))
(1 2 3 4)
> (up '((x (y)) z))
(x (y) z)
```

2. `(swapper s1 s2 slist)` returns a list the same as `slist`, but with all occurrences of `s1` replaced by `s2` and all occurrences of `s2` replaced by `s1`.

```
> (swapper 'a 'd '(a b c d))
(d b c a)
> (swapper 'a 'd '(a d () c d))
(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

3. `(count-occurrences s slist)` returns the number of occurrences of `s` in `slist`.

```
> (count-occurrences 'x '((f x) y (((x z) x))))
3
> (count-occurrences 'x '((f x) y (((x z) () x))))
3
> (count-occurrences 'w '((f x) y (((x z) x))))
0
```

4. `(flatten slist)` returns a list of the symbols contained in `slist` in the order in which they occur when `slist` is printed. Intuitively, `flatten` removes all the inner parentheses from its argument.

```
> (flatten '(a b c))
(a b c)
> (flatten '((a) () (b ()) () (c)))
(a b c)
> (flatten '((a b) c (((d)) e)))
(a b c d e)
> (flatten '(a b (() (c))))
(a b c)
```

5. `(merge lon1 lon2)`, where `lon1` and `lon2` are lists of numbers that are sorted in ascending order, returns a sorted list of all the numbers in `lon1` and `lon2`.

```
> (merge '(1 4) '(1 2 8))
(1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 90))
(3 35 62 81 83 85 90 90 91)
```

Exercise 1.17 [⋆⋆⋆]

1. `(path n bst)`, where `n` is a number and `bst` is a binary search tree that contains the number `n`, returns a list of `lefts` and `rights` showing how to find the node containing `n`. If `n` is found at the root, it returns the empty list.

```
> (path 17 '(14 (7 () (12 () ()))
                (26 (20 (17 () ())
                        ())
                    (31 () ()))))
(right left left)
```

2. `(sort lon)` returns a list of the elements of `lon` in increasing order.

```
> (sort '(8 2 5 2 3))
(2 2 3 5 8)
```

3. `(sort predicate lon)` returns a list of elements sorted by the predicate.

```
> (sort < '(8 2 5 2 3))
(2 2 3 5 8)
> (sort > '(8 2 5 2 3))
(8 5 3 2 2)
```

Exercise 1.18 [⋆⋆⋆] This exercise has three parts. Work them in order.

1. Define the procedure `compose` such that `(compose p1 p2)`, where `p1` and `p2` are procedures of one argument, returns the composition of these procedures, specified by this equation:

```
((compose p1 p2) x) = (p1 (p2 x))
> ((compose car cdr) '(a b c d))
b
```

2. `(car&cdr s slist errvalue)` returns an expression that, when evaluated, produces the code for a procedure that takes a list with the same structure as `slist` and returns the value in the same position as the leftmost occurrence of `s` in `slist`. If `s` does not occur in `slist`, then `errvalue` is returned. Do this so that it generates procedure compositions.

```
> (car&cdr 'a '(a b c) 'fail)
car
> (car&cdr 'c '(a b c) 'fail)
(compose car (compose cdr cdr))
> (car&cdr 'dog '(cat lion (fish dog ()) pig) 'fail)
(compose car (compose cdr (compose car (compose cdr cdr))))
> (car&cdr 'a '(b c) 'fail)
fail
```

3. Define `car&cdr2`, which behaves like `car&cdr`, but does not use `compose` in its output.