Chapter 18

Vectors

Problem 18.1

Define a (destructive) function that satisfies the following contract:

;;; DOUBLE-ALL!
;;; -----------------------------------------------
;;; INPUT: VECKY, a vector of numbers
;;; OUTPUT: The same vector, modified as described below
;;; SIDE EFFECT: Doubles the contents of each slot (destructively)

> (define vecky (vector 10 20 30 40 50))
> vecky
#(10 20 30 40 50)
> (double-all! vecky)
#(20 40 60 80 100)
> vecky
#(20 40 60 80 100)

Problem 18.2

Define a (destructive) function, called roll-em!, that satisfies the following contract:

;;; ROLL-EM!
;;; -----------------------------------------------
;;; INPUT: DICE, a vector of numbers
;;; OUTPUT: The same vector, but with contents destructively
;;; modified, as follows
;;; SIDE EFFECT: Replaces each slot with a random toss of a
;;; six-sided die

> (define dice (make-vector 5))
> (roll-em! dice)
#(3 2 5 3 4)
> dice
#(3 2 5 3 4)
> (roll-em! dice)
#(1 6 1 5 5)
> dice
Problem 18.3

Define a (destructive) function called roll-some! that satisfies the following contract:

;;; ROLL-SOME!
;;; --------------------------------
;;; INPUTS: DICE-VECK, a vector of dice values
;;; ROLLER, a vector of the same length as DICE-VECK,
;;; but consisting solely of 1s and 0s
;;; OUTPUT: DICE-VECK, modified as described below
;;; SIDE EFFECT: Walks through the two input vectors in parallel.
;;; For each index I, if the Ith element of ROLLER is a 1, then
;;; the Ith element of DICE-VECK is replaced by a random toss
;;; of a 6-sided die; otherwise, it is unchanged.

Here are some examples:

> my-dice
#(1 2 1 6 6 3 2 3)
> (roll-some! my-dice #(1 1 1 0 0 1 1 1))
#(5 4 2 6 6 3 4)
> (roll-some! my-dice #(1 1 1 0 0 0 1 1))
#(1 6 3 6 6 6 2 2)
> (roll-some! my-dice #(1 0 1 0 0 0 1 1))
#(4 6 5 6 6 1 6)
> my-dice
#(4 6 5 6 6 1 6)

Problem 18.4

Define a non-destructive function, called vector-reverse, that satisfies the following contract. Because it is non-destructive, it must create a new vector, instead of modifying the given vector.

;;; VECTOR-REVERSE
;;; ----------------------------------
;;; INPUT: VECK, a vector
;;; OUTPUT: A *new* vector that is just like VECK, except
;;; that its elements are in the reverse order.
;;; SIDE EFFECTS: none

Here are some examples:

> (vector-reverse #(a b c d))
#(d c b a)
> (vector-reverse #(1 2 3))
#(3 2 1)
Notice that the inputs in the above examples are immutable! So, if the function had tried to modify them, it would have caused an error!

Hints: Create a new vector of the appropriate length. Then use dotimes to walk thru the vector, setting its elements to appropriate values. Recall the print-in-reverse function from Example 18.4.3 for ideas. Don’t forget to return the new vector as output.

Problem 18.5

Define a (destructive) function that satisfies the following contract:

;; VECTOR-REVERSE!
;; ---------------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: The same vector, modified as described below
;; SIDE EFFECT: Destructively reverses the order of the
;; elements in VECKY.

> (vector-reverse! (vector 1 2 3 4 5))
#(5 4 3 2 1)
> (define vecky (vector 'a 'b 'c 'd))
> vecky
#(a b c d)
> (vector-reverse! vecky)
#(d c b a)
> vecky
#(d c b a)

Problem 18.6

Define a non-destructive function that satisfies the following contract:

;; VECTOR-MAP
;; -----------------------------------------------------------
;; INPUTS: FUNK, a function that expects one input
;; VEKK, a vector of suitable inputs for FUNK
;; OUTPUT: A *new* vector of the same length as VEQUE
;; each of whose elements is obtained by applying FUNK to
;; the corresponding element of VEKK.
;; SIDE EFFECT: *NONE*

Here is an example that assumes that the facty function has already been defined.

> (define vek #(1 2 3 4 5)) ← #(1 2 3 4 5) is an immutable vector
> (vector-map facty vek)
#(1 2 6 24 120) ← New vector created by vector-map
> vek
#(1 2 3 4 5) ← vek hasn’t changed

Notice that vek has been defined to be an immutable vector (i.e., its contents can’t be changed) and, thus, even if vector-map wanted to change its contents, it could not. (Attempting to do so would cause an error.) After vector-map is finished, vek remains the same (i.e., vector-map is non-destructive).
Hints: Use `make-vector` to create a new vector of the same length as `vek`. Then use `dotimes` to walk thru the new vector, setting each element to the result obtained by applying `funk` to the corresponding element of `vek`.

**Problem 18.7**

Define a destructive function that satisfies the following contract:

```scheme
;; VECTOR-MAP!
;; -----------------------------------------------------------
;; INPUTS: FUNK, a function that expects one input
;; VEQUE, a vector of suitable inputs for FUNK
;; OUTPUT: VEQUE, destructively modified...
;; SIDE EFFECT: Destructively modifies VEQUE by replacing
each of its elements by the result of applying FUNK to
the corresponding element of VEQUE.
```

**Here is an example:**

```scheme
> (define vec (vector 1 2 3 4 5))  ← VEC is a mutable vector
> (vector-map! facty vec)
#(1 2 6 24 120)  ← VECTOR-MAP! does its thing
> vec
#(1 2 6 24 120)  ← VEC has been changed!
```

Recall that the built-in `vector` function creates a mutable vector (i.e., one whose contents can be changed). `vector-map!` walks through the input vector, destructively modifying its contents. Afterward, `vec` is shown to be modified. Thus, `vector-map!` is destructive!

Hints: No need to create a new vector. Just use `dotimes` to walk through the given vector, destructively modifying its contents as you go.

**Problem 18.8**

Define a destructive function that satisfies the following contract: Define a function that satisfies the following contract:

```scheme
;; VECK-REPLACE!
;; -----------------------------------
;; INPUTS: OLD, anything
;; NEW, anything
;; VECKY, a vector
;; OUTPUT: VECKY, destructively modified as follows.
;; SIDE EFFECT: Replaces each occurrence of OLD in VECKY
;; by NEW, where equality is as judged by EQ?.
```

**Here is an example:**

```scheme
> (define vequi (vector 1 2 1 3 1 4))  ← VEQUI is mutable
> vequi
#(1 2 1 3 1 4)
> (veck-replace! 1 111 vequi)
```
#(111 2 111 3 111 4)
> vequi
#(111 2 111 3 111 4) ← VEQUI has changed!

## Problem 18.9

**Define a non-destructive function, called veck-index-of, that satisfies the following contract:**

```scheme
;; VECK-INDEX-OF
;; --------------------------------------------------------------
;; INPUTS: ITEM, anything
;; VECKY, a vector
;; OUTPUT: The index of the first slot of VECKY that contains
;; an first occurrence of ITEM; or #f if ITEM does
;; not appear in VECKY.
```

**Here are some examples of its use:**

```scheme
> (veck-index-of 'x #(a b c x y z x x))
3
> (veck-index-of 'z #(a b c x y z x x))
5
> (veck-index-of 'w #(a b c x y z x x))
#f
```

**Hint:** Define a helper function, called veck-index-of-helper, that includes an extra input `I`, that serves as an index into the given vector. Make recursive function calls, incrementing `I`, until you find an occurrence of `ITEM`, or `I` gets too big. Here’s the contract for the helper function:

```scheme
;; INDEX-OF-HELPER
;; -------------------------------
;; INPUTS: ITEM, anything
;; VECKY, a vector
;; I, a numerical index
;; OUTPUT: The index of the slot of VECKY that contains the
;; first occurrence of ITEM at or after index I;
;; or #f if ITEM does not appear in VECKY.
```

## Problem 18.10: Converting a vector into a list

**The goal of this problem is to define a function, called veck-to-list, that takes a vector as its only input, and returns as its output a list containing the same elements, in the same order, as illustrated below:**

```scheme
> (veck-to-list #(a b c d))
(a b c d)
> (veck-to-list (make-vector 5))
(0 0 0 0 0)
> (veck-to-list (vector 1 #t ()))
(1 #t ())
```

**Here is the contract for veck-to-list:**
Given the tools that we have seen so far, for this problem, it is probably easiest to define a recursive helper function, called `veck-to-list-helper`, that includes an extra input \( I \) that identifies the current element of the vector. Here is the contract:

\[
\text{\textbf{VECK-TO-LIST-HELPER}}
\]

\[
\text{\textit{INPUTS:}} \quad \text{VECK, a vector}
\]
\[
\text{\textit{I, an index}}
\]
\[
\text{\textit{OUTPUT:}} \quad \text{A list containing the elements of VECK from}
\]
\[
\text{the index I onward}
\]

Here are some examples:

\[
> (\text{veck-to-list-helper} \quad #(a b c d e) \quad 2)
\]
\[
(c d e)
\]
\[
> (\text{veck-to-list-helper} \quad #(a b c d e) \quad 3)
\]
\[
(d e)
\]

Hints:

\* When the index \( I \) reaches the length of the vector, then it is no longer a valid index for that vector, indicating that you have reached the base case of the recursion.

\* Remember, you are returning a list in the base case, and a list in the recursive case!

\* For this problem, it is probably easier not to use an accumulator.

Incidentally, after you have implemented this function, you can compare it to the built-in function, `vector->list`, that does the same thing!

---

### Problem 18.11: Converting a list into a vector

The goal of this problem is to define a function, called `list-to-veck`, that takes a list as its only input, and returns as its output a vector containing the same elements, in the same order, as illustrated below:

\[
> (\text{list-to-veck} \quad '(a b c d))
\]
\[
#(a b c d)
\]
\[
> (\text{list-to-veck} \quad (\text{list} \ 1 \ #t \ ()))
\]
\[
#(1 \ #t \ ())
\]

Here is the contract for `list-to-veck`:

\[
\text{\textbf{LIST-TO-VECK}}
\]

\[
\text{\textit{INPUT:}} \quad \text{LISTY, a list}
\]
\[
\text{\textit{OUTPUT:}} \quad \text{A vector containing the same elements as LISTY,}
\]
\[
\text{and in the same order}
\]
\[
\text{\textit{SIDE EFFECTS:}} \quad \text{none}
\]
Hints:

You can call the built-in `length` function once to find out how long the list is—but don’t call it more than once! Use it to create a vector of the same length. Then, you might think about using `docharge` to walk thru the vector, but that would be inefficient if you are planning to use the `fetch-nth-element` or `list-ref` functions (cf. in-class problem 16.2.3) to access the successive elements of the list. Instead, define a recursive helper function that satisfies the following contract. It will use list-based recursion to walk thru the list. On each recursive function call, the first element of the list will be the one you want to give to the `vector-set!` function. This approach will be much more efficient.

```scheme
;; LIST-TO-VECK-HELPER
;; -----------------------------------------------------------
;; INPUTS: LISTY, a list
;; VECK, a vector
;; I, an index (non-negative integer)
;; OUTPUT: The vector, VECK, modified as described below
;; SIDE EFFECTS: When called with I=0, copies the contents
;; of LISTY into VECK. When called with I>0, copies the
;; contents of LISTY into the slots of VECK, starting at
;; index I.
```

Here are some examples:

```scheme
> (list-to-veck-helper '(a b c d e) (make-vector 5) 0)
#(a b c d e)
> (list-to-veck-helper '(c d e) (make-vector 5) 2)
#(0 0 c d e)
```

Incidentally, after you have finished defining these functions, you may want to compare them to the built-in `list->vector` function that does the same thing!

---

**Problem 18.12**

Define a non-destructive function, called `every-other-one-vector`, that satisfies the following contract.

```scheme
;; EVERY-OTHER-ONE-VECTOR
;; ---------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: A vector containing every other element of VECKY.
;; Note: The output vector should contain roughly half the
;; elements of VECKY.
```

Here are some examples of its behavior:

```scheme
> (every-other-one #(0 1 2 3 4 5 6))
#(0 2 4 6)
> (every-other-one #(1 2 3 4 5 6))
#(1 3 5)
```

Hints:
Create a new vector whose length is roughly half that of the input vector. Then use \texttt{doptimes} to walk through that new vector, copying relevant elements from the input vector to the new vector.

You may find the built-in \texttt{even?}, \texttt{odd?}, or \texttt{quotient} functions helpful. (\texttt{even?} and \texttt{odd?} were introduced in Problem 14.2; \texttt{quotient} was introduced in Section 5.3.)

You should not use lists for this function; use vectors!

---

**Problem 18.13: Computing a histogram**

Define a function, called \texttt{compute-histogram}, that satisfies the following contract:

```
;; COMPUTE-HISTOGRAM
;; -------------------------------
;; INPUT: VECK-O-DICE, a vector of dice values (each 1 thru 6)
;; OUTPUT: A vector of length 7, where the \textit{i}th slot contains
;; the number of occurrences of \textit{i} in VECK-O-DICE.
;; (The 0th slot of the output vector is ignored.)
```

Here's an example:

```
> (define my-dice #(1 2 1 2 6 6 6 5 2))
> (compute-histogram my-dice)
#(0 2 3 0 0 1 3)
```

In this case, \textit{my-dice} contains:

```
two   1s
three 2s
no    3s
no    4s
one   5
three 6s
```

These counts are reflected in the histogram computed by this function. Note that the histogram is a vector with seven slots, numbered 0 thru 6. The zeroeth slot is ignored. We only care about slots 1 thru 6. For each \textit{i} > 0, the slot of the output vector at index \textit{i} contains the number of occurrences of \textit{i} in \textit{my-dice} (or \textit{veck-o-dice}). Here's another example:

```
> (define my-dice #(3 3 3 3 1 1 1))
> (compute-histogram my-dice)
#(0 4 0 5 0 0 0)
```

Hint: Create a vector of length seven called \texttt{histy}, then use \texttt{doptimes} to walk through the \texttt{veck-o-dice} (not \texttt{histy}). For each index \textit{i}, look at the \textit{i}th slot of \texttt{veck-o-dice} and use its value to figure out which slot of \texttt{histy} to increment. In effect, \texttt{histy} is a vector with seven accumulators, one of which we are ignoring.

---

**Problem 18.14**

You may wish to review Problem 16.6 before starting this problem.

Define a function, called \texttt{veck-has-satisfier?}, that satisfies the following contract:
;; VECK-HAS-SATISFIER?
;; -----------------------------------------------
;; INPUTS: FUNK, a predicate that expects one input
;; VECK, a vector of suitable inputs for FUNK
;; OUTPUT: #t, if VECK contains an element that satisfies
;; FUNK (i.e., that makes FUNK return #t)
;; #f, otherwise.

Here are some examples:

> (veck-has-satisfier? number? #(a #t 3 x))
#t
> (veck-has-satisfier? symbol? #(1 2 3 4))
#f

Note that it would be inefficient to implement this function using \texttt{dotimes} because \texttt{dotimes} invariably
walks through the entire vector. This function should stop as soon as it finds an element of \texttt{veck} that
satisfies \texttt{funk}. Therefore, you should define a recursive helper function that takes an extra input, \texttt{i}, that
serves as an index into the vector \texttt{veck}.

---

**Problem 18.15**

Define a function, called \texttt{has-three-of-a-kind?}, that satisfies the following contract:

;;; HAS-THREE-OF-A-KIND?
;;; -----------------------------------------------
;;; INPUT: VECK-O-DICE
;;; OUTPUT: #t if the vector of dice contains *at least*
;;; three of one kind; #f otherwise.

Here are some examples:

> (has-three-of-a-kind? #(1 2 1 2 1)) ← \text{has three ones}
#t
> (has-three-of-a-kind? #(4 2 4 4 4)) ← \text{has four fours}
#t
> (has-three-of-a-kind? #(6 6 6 6 6)) ← \text{has five sixes}
#t
> (has-three-of-a-kind? #(6 5 6 5 2)) ← \text{does not have three of a kind}
#f

Notice that having four or five of a kind also counts as having three of a kind.

Hint: One way to solve this problem: Compute a histogram vector, then check whether it has an element
that is 3 or bigger. Can you think of a way to use \texttt{veck-has-satisfier?} from Problem 18.14
to check whether the histogram vector contains an element that is 3 or bigger?

---

**Problem 18.16**

Define a function, called \texttt{has-large-straight?}, that satisfies the following contract:

;;; HAS-LARGE-STRAIGHT?
;; INPUT: VECK-O-DICE, a vector of five dice values
;; OUTPUT: #t if the vector of dice contains all of the
;; numbers in \{1,2,3,4,5\} or \{2,3,4,5,6\}, in any order;
;; #f otherwise.

Here are some examples:

> (has-large-straight? #(2 4 5 3 1))
#t
> (has-large-straight? #(6 5 4 2 3))
#t
> (has-large-straight? #(6 4 1 2 3))
#f

Hints: You may assume that the input vector has exactly five slots. Compute a histogram and go from there! What must the histogram look like for a large straight? (Use the built-in equal? predicate (cf. In-class problem 18.4.2) to make your life easier!) Alternatively, convert the vector of dice into a list, then use the built-in sort function (cf. Example 16.5.6) to sort its contents into non-decreasing order. What must it look like at that point?