Chapter 18

Vectors

A vector is a data structure that, like a list, contains an ordered sequence of data. However, there are many significant differences between vectors and lists.

Recall that lists can be incrementally extended, one element at a time, using the cons function. As a result, as illustrated in Fig. 18.1, the individual cons cells in which the various elements of a list are stored frequently end up being scattered haphazardly about the computer’s memory. For this reason, accessing elements of a list can be relatively slow. For example, to access the one-thousandth element of a list typically requires recursively walking through the first thousand cons cells in the chain. However, as has been demonstrated repeatedly in previous chapters, much recursive processing of lists can be done while only accessing the first and rest of a list. For these sorts of applications, lists are quite handy. Furthermore, many list-based functions can be written non-destructively, which facilitates testing and debugging, since the performance of a non-destructive function depends only on the code within its body.

In sharp contrast to lists, vectors are stored within a single, contiguous block of memory—which has both advantages and disadvantages. The principal advantage is that every item of a vector can be accessed very quickly, whether the first element or the thousandth. To demonstrate why, Fig. 18.2 illustrates the layout of the elements in a typical vector. Notice that each slot has a numerical index, starting at zero. A crucial feature of the layout is that each slot takes up the same amount of space (e.g., four bytes per slot in the figure). Since each slot has the same size, the memory address of any given slot is very easy to compute. For example, if the start of the vector is located at memory address 1000, and each slot is four bytes wide, then the address of the $i$th slot is $1000 + 4 \times i$. (Thankfully, DrScheme takes care of such low-level details. A Scheme programmer need never deal directly with memory addresses.) An important feature of the “Random Access Memory” (RAM) found in modern computers is that the contents of any memory address can be fetched in the same (very small) amount of time. (The term “random” here is used to indicate that the contents of any randomly selected memory address can be fetched in the same, very small amount of time.) Thus, the time required to fetch the first element of a vector (i.e., the element with index 0) is the same as the time required to fetch the thousandth element.

One disadvantage of vectors is that the block of memory that will hold the vector’s elements must be allocated in one chunk. As a result, vectors cannot be easily extended to accommodate new elements. In particular, if a vector is created to hold up to $N$ elements, then it will never be able to hold more than $N$ elements. There is no analog to the cons function for vectors. Each vector has a fixed length.

Since not all of a vector’s elements may be known at the time the block of memory is allocated, destructive programming is typically used with vectors. For example, a Scheme function might decide to set the 24th element of a vector to be #t, then later set the 36th element to be 34.2, and still later set the 24th element to be #f (erasing the prior contents of the 24th slot). Just as accessing any desired slot of a vector is very efficient, so too is the operation of destructively modifying the contents of any desired slot. (Again, this is a feature of Random Access Memory.) Thus, the speed of accessing and modifying the contents of a vector is balanced by the challenge of using destructive programming, which can make testing and debugging your functions a little more difficult. Nonetheless, the tradeoff can be quite worthwhile; and if care is taken, the risks associated with using destructive programming can be mitigated.
18.1 Vector Expressions

Chapter 2 introduced several types of primitive data expressions, including numbers, booleans, the empty list, and symbols. For each type of expression, the syntactic rules for character sequences denoting that type of data were described. Then, Chapter 3 described how instances of those data types are evaluated. Similarly, Chapter 6 presented the syntactic rules for character sequences that denote non-empty lists, and described how the Default Rule evaluates non-empty lists. Following this trend, this section presents the #(...) syntax—called the pound syntax—for character sequences that denote vectors, as well as the semantics of evaluation for vectors. (Vectors evaluate to themselves.) It should be noted, however, that the pound syntax has limited use because the vectors it denotes are immutable (i.e., their contents cannot be changed). The pound syntax is typically only useful for testing functions that look at the contents of vectors, and perhaps do some computations based on those contents, but do not try to change the contents in any way. After presenting the pound syntax, the next section presents built-in functions for creating mutable vectors (i.e., vectors whose contents can be changed), which turn out to be more useful in practice.

A vector is a datum. Scheme provides a special syntax, the pound syntax, for denoting vectors. Just as the expression (a b c) can be used to represent (or denote) a list containing the three elements a, b and c, the expression #(a b c) can be used to represent (or denote) a vector containing the three elements a, b and c. In general, any expression of the form #(e₁ e₂ ... eₙ) can be used to represent a vector containing the n elements denoted by the expressions e₁, e₂, ..., eₙ. Thus, for example, the expression #(x #t () 32) denotes a vector containing four elements: a symbol, a boolean, the empty list, and a number.

One important fact about the #(...) syntax is that the vectors it represents are immutable (i.e., their contents cannot be changed). This limits the usefulness of the #(...) syntax. However, it can be useful when testing functions that don’t need to modify their vector inputs (e.g., functions that print out the contents of a vector). Immutable vectors can also be useful as the values of global constants. For example, the vector #(clubs hearts diamonds spades) could be used to denote the suits in a deck of cards, and the vector
#(sun mon tue wed thu fri sat) could be used to denote the days of the week.

* Unlike lists, vectors evaluate to themselves!

**Example 18.1.1: Demonstrating that vectors evaluate to themselves**

The following interactions use the #(...) syntax to demonstrate that vectors evaluate to themselves.

```
> #(1 2 a b #t (+ 2 3))        ❯ The pound syntax denotes a vector
#(1 2 a b #t (+ 2 3))        ❯ That vector evaluates to itself!
> #(a b #(c d e) (x y z))
#(a b #(c d e) (x y z))
```

With the #(...) syntax, there is no need to quote the subsidiary expressions. For example, #(a b c) simply denotes the vector containing the three symbols a, b and c. When the vector gets evaluated, its elements are not evaluated! The vector simply evaluates to itself—without evaluating any of its elements. The Default Rule for evaluating non-empty lists does not apply to vectors!

Incidentally, DrScheme uses the #(...) syntax to report results that are vectors, whether they are immutable or mutable.

* Remember: Vectors denoted by the # syntax are immutable! Once created, their contents can’t be changed.

### 18.2 Constructing Vectors

Scheme provides two built-in functions, called **vector** and **make-vector**, that can be used to create new vectors. The **vector** function is similar to the built-in **list** function, mentioned briefly in Section 16.1. The **make-vector** function is typically the most practical. It enables the programmer to create a vector of any specified length.

#### 18.2.1 The Built-in vector Function

The **vector** function is a built-in function that works very much like the built-in **list** function, mentioned briefly in Section 16.1. Whereas **list** constructs a list containing the specified elements, **vector** constructs a vector containing the specified elements. Unlike the vectors denoted by the #(...) syntax, vectors created by the **vector** function are mutable (i.e., their contents can be changed—we’ll see how momentarily).

**Example 18.2.1: Using the vector function**

Here’s the contract for the **vector** function, followed by some examples of its use.

```
;;  VECTOR -- built-in function
;;  --------------------------------------------------------------
;;  INPUTS: ELT1, ELT2, ELT3, ... : any number of inputs
;;  OUTPUT: A *mutable* vector containing the specified elements

> (vector 1 (+ 2 3) ’a)
#(1 5 a)
> (list 1 (+ 2 3) ’a)
(1 5 a)
> (define vecky (vector 1 (+ 2 3) ’a))
> vecky
```
Note that since vector is a built-in function, expressions such as (vector 1 (+ 2 3) ’a) are evaluated by the Default Rule. Thus, all input expressions are evaluated before being passed as input to the vector function—which is why the a in the first two examples had to be quoted.

18.2.2 The make-vector Function

The make-vector function is a built-in function that can be used to create a vector of any specified length. It is the most common way of creating a vector because it can be used, for example, to easily create a vector with, say, 1000 slots. Like with the vector function, vectors created by make-vector are mutable (i.e., their contents can be changed). The make-vector function can be called with either one or two inputs (i.e., the second input is optional).\footnote{It is not hard to define Scheme functions that take optional inputs, as a quick internet search will reveal; however, doing so is beyond the scope of this textbook.} The first input specifies how many slots the new vector should contain. The second input, if provided, specifies the initial value for all of the slots. If the second input is not provided, make-vector fills each slot with a default value of 0.

Example 18.2.2: Using the make-vector function

Here’s the contract for the make-vector function, followed by some examples of its use.

```scheme
;; MAKE-VECTOR -- built-in function
;; -------------------------------------------------------------
;; INPUTS: NUM-SLOTS, a non-negative integer
;; INIT-VALUE, an *optional* input, can be anything
;; OUTPUT: A vector with NUM-SLOTS slots, each initially
;; filled with INIT-VALUE (or 0 if INIT-VALUE
;; not provided)

> (make-vector 5)  ⍟ called with one input
#(0 0 0 0 0)
> (make-vector 15)
#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (make-vector 5 ’a)
#(a a a a a)
> (make-vector 15 ’_)
#( _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
```

18.3 Accessing Information Stored in a Vector

To be of any use, the information stored in a vector must be accessible to the programmer. Scheme provides the vector-ref function to access the contents of any specified slot in a vector, and the vector-length function to access the (fixed) number of elements in a vector.
18.3.1 The \texttt{vector-ref} Function

Each element of a vector is identified by its numerical index. The built-in \texttt{vector-ref} function provides an easy way to access any element of a vector by its index.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Example 18.3.1: Using \texttt{vector-ref}} \\
\hline
\end{tabular}
\end{center}

Here is the contract for the \texttt{vector-ref} function, followed by some examples of its use.

\begin{verbatim}
;;; VECTOR-REF -- built-in function
;;; ----------------------------------------------------------------------
;;; INPUTS: VECKY, a vector
;;;          INDY, an index
;;; OUTPUT: The item of VECKY stored at slot INDY

> (define vecky #(a b c d e))
> (vector-ref vecky 0)
a
> (vector-ref vecky 2)
c
\end{verbatim}

* Although the \texttt{vector-ref} and \texttt{list-ref} functions may appear similar syntactically, they operate quite differently. The \texttt{vector-ref} function accesses the specified element of a vector nearly instantaneously, while the \texttt{list-ref} function walks through each element of the specified list until it finds the one with the desired index. Thus, \texttt{vector-ref} is much more efficient than \texttt{list-ref}.

18.3.2 The \texttt{vector-length} Function

As already mentioned, each vector has a fixed length. For this reason, the length of the vector can be stored with the vector itself, when it is created. Thus, the built-in \texttt{vector-length} function does not need to walk through the entire vector to figure out how long it is; instead it can simply look up the length information that is stored with the vector. (The length of a vector is an example of a field in a data structure, which will be discussed in the next chapter.) Thus, using \texttt{vector-length} is very fast with any vector, no matter how long. This is quite different from the built-in \texttt{length} function for lists which must walk all the way through a list in order to figure out how many elements it has.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Example 18.3.2: Using \texttt{vector-length}} \\
\hline
\end{tabular}
\end{center}

Here is the contract for \texttt{vector-length}, followed by some examples of its use.

\begin{verbatim}
;;; VECTOR-LENGTH -- built-in function
;;; ----------------------------------------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: The number of elements/slots in that vector

> (define vecky #(a b c d e))
> (vector-length vecky)
5
> (vector-length (make-vector 25))
25
\end{verbatim}
In-Class Problem 18.3.1: Fetching a random element from a vector

Define a function, called fetch-random-element, that satisfies the following contract:

;;; FETCH-RANDOM-ELEMENT
;;; -----------------------------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: One of the elements of VECKY, chosen at random

Here are some examples of the desired behavior:

> (fetch-random-element #(a b c d e f))
c
> (fetch-random-element #(a b c d e f))
a
> (fetch-random-element #(a b c d e f))
d

Hint: Use the built-in vector-length, random and vector-ref functions.

18.4 Recursively or Iteratively Processing Vectors

Since a vector is an ordered sequence of elements, we can define recursive functions to walk through vectors in much the same way that we defined recursive functions to walk through lists. One example of this will be given below. However, writing recursive functions to walk through vectors would quickly grow tiresome because it would involve repeating the same kind of pattern over and over again. To avoid this kind of repetition, we will instead use the dotimes special form to enable us to iteratively walk through any vector. However, before introducing this use of dotimes, we first demonstrate how to manually write a recursive function to walk through some or all of a vector.

Example 18.4.1: (Optional) Manually walking through a vector

The following function prints out the contents of a vector from a given starting index. On each recursive function call, the value of the index is incremented, until it goes past the end of the vector. Recall that if a vector has length n, then the legal indices range from 0 to n − 1.

;;; PRINT-VECTOR-FROM
;;; ---------------------------------------------------------------
;;; INPUTS: VECKY, a vector
;;; I, a non-negative integer, no greater than
;;; the length of VECKY
;;; OUTPUT: None
;;; SIDE EFFECT: Prints out the elements of VECKY
;;; from index I onward.

(define print-vector-from
  (lambda (vecky i)
    (cond
      ;; Base Case: I is too big
      ;; Note: The last legal index of VECKY is LENGTH-1
      ((>= i (vector-length vecky))
        (void))
      ;; Recursive Case: I is in range
      (void) (print-vector-from vecky (+ i 1))))
;; Recursive Case: I is a legal index
(else
 ;; Print one element
 (printf "Element \char\%A of VECKY is: \char\%A\%" i (vector-ref vecky i))
 ;; Let recursion print the rest of the elements
 (print-vector-from vecky (+ i 1))))) 

> (print-vector-from #(a b c d) 0)
Element 0 of VECKY is: a
Element 1 of VECKY is: b
Element 2 of VECKY is: c
Element 3 of VECKY is: d

> (print-vector-from #(a b c d) 2)
Element 2 of VECKY is: c
Element 3 of VECKY is: d

The wrapper function, print-vector-wr, can then be defined as follows:

;;; PRINT-VECTOR-WR
;;; ---------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: None
;;; SIDE EFFECT: Prints out the elements of VECKY

(define print-vector-wr
 (lambda (vecky)
   (print-vector-from vecky 0)))

> (print-vector-wr #(a b c))
a
b
c

In general, any function that needs to recursively process the elements of a vector can do so by defining a helper function that includes an extra input \(i\) whose value is initially zero and increments by one on each recursive function call until it exceeds the last legal index for the given vector. However, the dotimes special form simplifies the task of walking through a vector.

**Example 18.4.2: Printing the contents of a vector using dotimes**

The print-vector function, below, illustrates the use of the dotimes special form to walk through a vector, printing out its contents. Note the use of the vector-length function to specify the number of iterations to perform.

;;; PRINT-VECTOR
;;; ---------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: None
;;; SIDE EFFECT: Displays the contents of VECKY in the
```
;; Interactions Window, one element per row

(define print-vector
  (lambda (vecky)
    ;; I takes on the values: 0, 1, 2, ..., LENGTH-1
    (dotimes (i (vector-length vecky))
      ;; Print out the Ith element of VECKY
      (printf "A: A\n" i (vector-ref vecky i))))

> (define vecky #(a b c d e))
> (print-vector vecky)
a
b
c
d
e
```

---

**Example 18.4.3: Printing the contents of a vector in reverse order**

The following function prints out the contents of a given vector in reverse order. Notice the use of the local variable rev-indy, whose value is the index of the next element to be printed. For example, you should convince yourself that for a vector of length four, the counter variable indy will range from 0 to 3, while the local variable rev-indy will range from 3 to 0.

```
;; PRINT-IN-REVERSE
;; --------------------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: None
;; SIDE EFFECT: Prints out the contents of VECKY
;; in reverse order

(define print-in-reverse
  (lambda (vecky)
    ;; LEN = number of elements in VECKY
    (let ((len (vector-length vecky)))
      (dotimes (indy len)
        ;; INDY takes on values from 0 to LEN-1
        ;; REV-INDY takes on values from LEN-1 to 0
        (let ((rev-indy (- len indy 1)))
          (printf "vecky[\nA\n] = A\n" rev-indy
                  (vector-ref vecky rev-indy))))))

> (print-in-reverse #(a b c d))
vecky[3] = d
vecky[2] = c
vecky[1] = b
vecky[0] = a
```
In-Class Problem 18.4.1: Using `dotimes` to print out every other element of a vector

Define a function, called `print-every-other-one-veck`, that takes a vector as its only input. It should not return any output value. Instead, it should print out every other element of the given vector. Here is the contract, followed by some examples:

```scheme
;; PRINT-EVERY-OTHER-ONE-VECK
;; ---------------------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: None
;; SIDE EFFECT: Prints out every other element of VECK

> (print-every-other-one-veck #(a b c d e))
a
c
e
> (print-every-other-one-veck #(a a b b c c d d))
a
b
c
d
```

Hint: Use the `even?` function. If the current index is even, then print out the corresponding element.

In-Class Problem 18.4.2: Testing whether two vectors are “equal”

Define a function, called `vector-equal?`, that satisfies the following contract:

```scheme
;; VECTOR-EQUAL?
;; ---------------------------------------------
;; INPUTS: VECK-ONE and VECK-TWO, any vectors
;; OUTPUT: #t if VECK-ONE and VECK-TWO have the same elements, in the same order; #f otherwise.

> (vector-equal? #(a b c) #(a b c))
#t
> (vector-equal? (make-vector 3) (vector 0 0 0))
#t
> (vector-equal? #(a b c) #(a b c d))
#f
```

Notice that the two input vectors cannot be equal if they have different lengths. Therefore, `vector-equal?` can immediately return `#f` if the two vectors have different lengths. On the other hand, if they do have the same length, then it can call a helper function, `vector-equal?-helper`, to manually walk through the vectors in parallel, comparing their corresponding elements. Note that using `dotimes` is not an option for this problem because the helper function should be able to stop early if it ever discovers corresponding elements that are not the same. Here’s the contract for the helper function:

```scheme
;; VECTOR-EQUAL?-HELPER
;; ---------------------------------------------
```
; Inputs: VECK-ONE, VECK-TWO, two vectors of the same length
; I, an index
; Output: #t if the corresponding elements of VECK-ONE and
; VECK-TWO are the same from index I onward.
; #f otherwise.

Notice that the helper function is only ever called on vectors having the same length.

After you’ve implemented this function, you may wish to know that the built-in equal? function can be used to test the equality of vectors, as illustrated below.

> (equal? #(1 2 3) #(1 2 3))
#t
> (equal? #(a b c) (vector 'a 'b 'c))
#t
> (equal? #(0 0 0) (make-vector 4))
#t

18.5 Destructively Modifying a Vector

The vector-set! function is provided to enable a programmer to destructively modify the contents of a specified slot in a vector. It is frequently called a mutator function, because it enables a programmer to mutate the contents of a vector.

* The name of the function ends with an exclamation point to remind us of its destructive side effect.

Example 18.5.1: The vector-set! function

Here is the contract for the vector-set! function, followed by an example of its use.

; VECTOR-SET! -- Built-in Function
; -----------------------------
; Inputs: VECKY, a vector
; INDY, a numerical index
; NEW-VAL, anything
; Output: *void*
; Side Effect: Destructively changes the contents of VECKY
; at slot INDY to become NEW-VAL

> (define vecko (vector 0 10 20 30))
> vecko
#(0 10 20 30)
> (vector-set! vecko 2 'x)
> vecko
#(0 10 x 30)
In-Class Problem 18.5.1: Initializing a vector

Define a function, called init-veck, that satisfies the following contract:

;; INIT-VECK
;; ---------------------------------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: Don’t care
;; SIDE EFFECT: Sets the value of slot 0 to 0, the value
;; of slot 1 to 1, and so on.

Here is an example of its use:

> (define vecky (make-vector 5))
> vecky
#(0 0 0 0 0)
> (init-veck vecky)
> vecky
#(0 1 2 3 4)

Hint: Use dotimes and vector-set!.

In-Class Problem 18.5.2: Swapping elements of a vector

Define a destructive function, called vector-swap!, that destructively modifies a vector by swapping two of its elements as specified by the following contract:

;; VECTOR-SWAP!
;; ----------------------------------------------
;; INPUTS: VECKY, a vector
;; I, J, two numerical indices
;; OUTPUT: don’t care
;; SIDE EFFECT: Destructively swaps the contents of VECKY
;; at slots I and J.

Here are some examples of its use:

> (define vecky (vector 'a 'b 'c 'd 'e 'f))
> vecky
#(a b c d e f)
> (vector-swap! vecky 1 4)
> vecky
#(a e c d b f)
> (vector-swap! vecky 0 4)
> vecky
#(b e c d a f)
18.6 Summary

This chapter introduced vectors that, like lists, are ordered sequences of data. However, vectors and lists have many important differences. For example, whereas the elements of a list reside in cons cells that may be scattered haphazardly throughout a computer’s memory, the contents of a vector reside within a single, contiguous block of memory. As a result, accessing the individual elements of a vector is typically much faster than accessing the individual elements of a list. Whereas lists are especially amenable to recursive processing using non-destructive programming, vectors are especially amenable to iterative processing using destructive programming. Although care must be taken when using destructive programming with vectors, the efficiency of vectors often makes the tradeoff worthwhile.

Scheme provides a special syntax, the pound syntax, for denoting immutable vectors. For example, the expression #(1 a x) denotes a vector containing three elements. The pound syntax is of limited use, for example, when testing functions that look at vectors but do not try to change their contents, or for denoting global constants (e.g., a vector containing the names of the days of the week).

To enable programmers to do meaningful computations with vectors, Scheme provides a small handful of built-in functions. The functions, vector and make-vector, are constructor functions that can be used to create new vectors, either by explicitly specifying all of the individual elements or by specifying only the total number of slots. The functions, vector-ref and vector-length, are accessor functions that can be used to access the information contained in a vector: vector-ref fetches a specified element from a vector, while vector-length returns the number of elements contained in a vector. The function, vector-set!, is a mutator function: it can be used to destructively modify the contents of an individual slot in a vector. (There is also a type-checker predicate for vectors, called vector?, but it is rarely needed.)

Because the individual elements of a vector are accessed by their numerical indices, the dotimes special form frequently comes in handy when needing to perform computations on the contents of vectors.

Built-In Functions Introduced in this Chapter

- vector: Construct a vector containing the specified items
- make-vector: Construct a vector of a specified length
- vector-ref: Fetches a specified element of a given vector
- vector-length: Fetches the length of a given vector
- vector-set!: Sets the value of a specified slot of a given vector
- vector->list: Convert vector into a list (cf. Problem 18.10)
- list->vector: Convert list into a vector (cf. Problem 18.11)