

Vectors

- Introduction
 - Indexing
 - Mutating
- Displaying Vectors
 - DrScheme's Display Variations
- Vector Procedures
 - `vector`
 - `make-vector`
 - `vector?`
 - `vector-length`
 - `vector-ref`
 - `vector-set!`
 - `vector->list` and `list->vector`
 - `vector-fill!`
- Implementing Standard Procedures

Introduction

Vectors are data structures that are very similar to lists in that they arrange data in linear fashion. Vectors differ from lists in two significant ways: Unlike lists, vectors are *indexed* and vectors are *mutable*.

Indexing

You may have noted that when we use lists to group data (e.g., the information on a course), we need to use `list-ref` to get latter elements of the list. Unfortunately, `list-ref` works by cdr'ing down the list. It takes about five steps to get to the fifth element of the list. It would be nicer if we could access any element of the group of data in the same amount of time (preferably a small amount of time).

Vectors contain a fixed number of elements and provide *random access* (also called *indexed access*) to those elements, in the sense that each element, regardless of its position in the vector, can be recovered in the same amount of time. In this respect, a vector differs from a list: The initial element of a list is immediately accessible, but subsequent elements are increasingly difficult and time-consuming to get at.

Mutating

You may have also noted that we occasionally want to change an element of a group of data (e.g., to change a student's grade in the structure we use to represent that student). When we use lists, we essentially need to build a new list to change one element.

Vectors are *mutable* data structures: It is possible to replace an element of a vector with a different value, just as one can take out the contents of a container and put in something else instead. It's still the same vector after the replacement, just as the container retains its identity no matter how often its contents are changed.

The particular values that a vector contains at some particular moment constitute its *state*. One could summarize the preceding paragraph by saying that the state of a vector can change and that state changes do not affect the underlying identity of the vector.

Displaying Vectors

When displaying a vector, Scheme displays each of its elements, enclosed in parentheses, with an extra character, #, in front of the left parenthesis. For instance, here's how Scheme displays a vector containing the symbols `alpha`, `beta`, and `gamma`, in that order:

```
#(alpha beta gamma)
```

The mesh (pound, sharp, hash) character distinguishes the vector from the list containing the same elements.

We can use the same syntax to specify a vector when writing a Scheme program or typing commands and definitions into the Scheme interactive interface, except that we have to place a single quotation mark before the mesh so that Scheme will not try to evaluate the vector as if it were some exotic kind of procedure call. (DrScheme will not make this mistake even if you forget the single quotation mark, but not all implementations of Scheme are so generous.)

I recommend that you avoid using this *vector literal* notation (that is, with the quotation mark), just as I recommend that you avoid the corresponding list literal notation for lists.

DrScheme's Display Variations

In the interaction window, when DrScheme displays a vector as the value of some top-level expression that the user supplied, it does something more ambitious, but potentially rather confusing: It inserts a numeral between the mesh character and the left parenthesis to indicate the number of elements in the vector, thus:

```
#3(alpha beta gamma)
```

However, you can instruct DrScheme to use the straight mesh-and-parentheses representation, with no numeral, by giving the following command at the beginning of your program or interactive session:

```
(print-vector-length #f)
```

That is, “Don't print the lengths of vectors”.

Vector Procedures

Standard Scheme provides the following fundamental procedures for creating vectors and selecting and replacing their elements:

vector

The constructor `vector` takes any number of arguments and assembles them into a vector, which it returns.

```
> (vector 'alpha 'beta 'gamma)
#(alpha beta gamma)

> (vector) ; the empty vector -- no elements!
#()

> (vector 'alpha "beta" '(gamma 3) '#(delta 4) (vector 'epsilon))
#(alpha "beta" (gamma 3) #(delta 4) #(epsilon))
```

As the last example shows, Scheme vectors can be *heterogeneous*, containing elements of various types, just like Scheme lists.

make-vector

The `make-vector` procedure takes two arguments, a natural number `k` and a Scheme value `obj`, and returns a `k`-element vector in which each position is occupied by `obj`.

```
> (make-vector 12 'foo)
#(foo foo foo foo foo foo foo foo foo foo foo)
> (make-vector 4 0)
#(0 0 0 0)
> (make-vector 0 4) ; the empty vector again
#()
```

The second argument is optional; if you omit it, the value that initially occupies each of the positions in the array is left unspecified. Various implementations of Scheme have different ways of filling them up, so you should omit the second argument of `make-vector` only when you intend to replace the contents of the vector right away.

vector?

The type predicate `vector?` takes any Scheme value as argument and determines whether it is a vector.

```

> (vector? (vector 'alpha 'beta 'gamma))
#t
> (vector? '#(alpha beta gamma)) ; discouraged notation
#t
> (vector? (list 'alpha 'beta 'gamma)) ; a list, not a vector
#f
> (vector? "alpha beta gamma") ; a string, not a vector
#f
> (vector? '#(#f)) ; a one-element vector (the element is #f)
#t

```

vector-length

The `vector-length` procedure takes one argument, which must be a vector, and returns the number of elements in the vector.

```

> (vector-length (vector 3 1 4 1 5 9))
6
> (vector-length (vector 'alpha 'beta 'gamma))
3
> (vector-length (vector))
0

```

vector-ref

The selector `vector-ref` takes two arguments -- a vector `vec` and a natural number `k` (which must be less than the length of `vec`). It returns the element of `vec` that is preceded by exactly `k` other elements. (In other words, if `k` is 0, you get the element that begins the vector; if `k` is 1, you get the element after that; and so on.)

```

> (vector-ref (vector 3 1 4 1 5 9) 4)
5
> (vector-ref (vector 'alpha 'beta 'gamma) 0)
alpha
> (vector-ref (vector 'alpha 'beta 'gamma) 3)
vector-ref: index 3 out of range [0, 2] for vector: #(alpha beta gamma)

```

vector-set!

All of the previous procedures look a lot like list procedures. Now let's see one that's much different. We can actually use procedures to change vectors.

The mutator `vector-set!` takes three arguments -- a vector `vec`, a natural number `k` (which must be less than the length of `vec`), and a Scheme value `obj` -- and replaces the element of `vec` that is currently in the position indicated by `k` with `obj`. This changes the state of the vector irreversibly; there is no way to find out what used to be in that position after it has been replaced. It is a Scheme convention to place an exclamation point meaning "Proceed with caution!" at the end of the name of any procedure that makes such an irreversible change in the state of an object.

The value returned by `vector-set!` is unspecified; one calls `vector-set!` only for its side effect on the state of its first argument.

```
> (define sample-vector (vector alpha beta gamma delta epsilon))
> sample-vector
#(alpha beta gamma delta epsilon)
> (vector-set! sample-vector 2 'zeta)
> sample-vector ; same vector, now with changed contents
#(alpha beta zeta delta epsilon)
> (vector-set! sample-vector 0 "foo")
> sample-vector ; changed contents again
#("foo" beta zeta delta epsilon)
> (vector-set! sample-vector 2 -38.72)
> sample-vector ; and again
#("foo" beta -38.72 delta epsilon)
```

Vectors introduced into a Scheme program by means of the mesh-and-parentheses notation are “immutable” -- applying `vector-set!` to such a vector is an error, and the contents of such vectors are therefore constant. (Some implementations of Scheme, including DrScheme, don't enforce this rule.)

vector->list and list->vector

The `vector->list` takes any vector as argument and returns a list containing the same elements in the same order; the `list->vector` procedure performs the converse operation.

```
> (vector->list '(31 27 16))
(31 27 16)
> (vector->list (vector))
()
> (list->vector '(#\a #\b #\c))
#(#\a #\b #\c)
> (list->vector (list 31 27 16))
#(31 27 16)
```

vector-fill!

The `vector-fill!` procedure takes two arguments, the first of which must be a vector. It changes the state of that vector, replacing each of the elements it formerly contained with the second argument.

```
> (define sample-vector (vector 'rho 'sigma 'tau 'upsilon))
> sample-vector ; original vector
#(rho sigma tau upsilon)
> (vector-fill! sample-vector 'kappa)
> sample-vector ; same vector, now with changed contents
#(kappa kappa kappa kappa)
```

The `vector-fill!` procedure is invoked only for its side effect and returns an unspecified value.

Implementing Standard Procedures

Some older implementations of Scheme may lack the `list->vector`, `vector->list`, and `vector-fill!` procedures, but it is straightforward to define them in terms of the others. We'll look at a number of different implementations of `list->vector`, each of which has the same introductory documentation.

```
;;; Procedure:
;;; list->vector
;;; Parameters:
;;; lst, a list.
;;; Purpose:
;;; Convert the list to a vector.
;;; Produces:
;;; vec, a vector
;;; Preconditions:
;;; lst is a list
;;; Postconditions:
;;; vec is a vector.
;;; The length of vec equals the length of lst.
;;; The ith element of vec equals the ith element of lst for
;;; all "reasonable" i.
```

In most implementations, we'll need to recurse over the list, adding elements to a corresponding vector. We'll also need to build that vector, first. Since we only want to build one vector, we should use the husk-and-kernel technique (which we've previously used for error checking). The husk creates the vector and tells the kernel and then calls the kernel.

However, we may need other parameters for the kernel, so it's time to think a little bit about the kernel. Since we want to copy all the elements of the list to the vector, we probably need to repeat some basic step, and the only way we know how to do repetition is recursion. To keep track of what we're copying, we'll probably need a counter that we pass to the helper. I'll call that counter `pos`, since it keeps track of a position in the list or vector.

What should we copy from at each step? We could copy the element at position `pos` of the list into position `pos` of the vector. However, that's somewhat inefficient because it requires us to step through to the *i*th element of the list.

Since we no longer need a value from the list once we've copied it, we can `cdr` through the list as we step through the vector. Now, our goal is to copy the initial element of the list into the appropriate position of the vector.

When do we stop? When we run out of elements to copy.

So, here's how to set things up.

```
(define list->vector
  (lambda (lst)
    (list->vector-kernel! lst ; Copy the whole list
                          0 ; Starting at position 0
                          (make-vector (length lst) null)
                          ; Into a new vector of the appropriate length
                          )))
```

The kernel is a little bit more complicated. We need to keep track of where we are in the vector. We may also want to carefully specify what this kernel is supposed to do. (Such specification is not always necessary for helpers, but I think it clarifies things in this case.)

```
;;; Procedure:
;;; list->vector-kernel!
;;; Parameters:
;;; lst, a list to copy
;;; pos, a position in the vector
;;; vec, a vector
;;; Purpose
;;; Copy values from lst into positions pos, pos+1, ...
;;; Produces:
;;; vec, the same vector but with different contents.
;;; Preconditions:
;;; The length of vec = pos + the length of lst. [Not verified]
;;; pos >= 0. [Not verified]
;;; Postconditions:
;;; Element pos of vec now contains element 0 of lst.
;;; Element pos+1 of vec now contains element 1 of lst.
;;; Element pos+2 of vec now contains element 2 of lst.
;;; ...
;;; The last element of vec now contains the last element of lst.
(define list->vector-kernel!
  (lambda (lst pos vec)
    ; We don't bother to verify the preconditions because this
    ; procedure should only be called by something that has already
    ; verified the preconditions (explicitly or implicitly).

    ; If there's nothing left to copy, stop.
    (if (null? lst) vec
        ; Otherwise, copy the initial element of the list and
        ; then copy the remaining elements
        (begin
          (vector-set! vec pos (car lst))
          (list->vector-kernel! (cdr lst) (+ pos 1) vec))))))
```

Is this the only way to write the procedure? No. More advanced students might know about the `apply` procedure which makes life significantly easier.

```
(define list->vector
  (lambda (ls)
    (apply vector ls)))
```

In other words: Call the `vector` procedure, giving it the elements of `ls` as its arguments.

Now let's consider how to go in the opposite direction. Once again, we'll probably need to recurse to step through positions and need to keep track of the position with an extra variable. Should we create a list first and then populate it? No. We don't typically mutate lists. So, we'll update the list "on the fly", adding elements and extending the list at each step. Let's start with the helper. The helper will build a list from a subrange of the vector.

```

;;; Procedure:
;;; vector->list-kernel
;;; Parameters:
;;;   vec, a vector
;;;   start, an integer
;;;   finish, an integer
;;; Purpose:
;;;   Builds a list that contains elements start ... finish-1 of vec
;;; Produces:
;;;   lst, a new list.
;;; Preconditions:
;;;   start >= 0
;;;   finish >= start
;;;   finish <= length of vec
;;; Postconditions:
;;;   The element at position i of lst is the element at position
;;;     i+start of vec for all reasonable i.
;;;   Does not change vec.
(define vector->list-kernel
  (lambda (vec start finish)
    ; We stop at the finish.
    (if (= start finish)
        ; There are no more elements to put into a list, so use
        ; the empty list.
        null
        ; Otherwise, add the element at position start to a list of
        ; the remaining elements.
        (cons (vector-ref vec start)
              (vector->list-kernel vec (+ 1 start) finish))))))

```

Now, we just have to set things up for this kernel. We start at position 0. We end just before the length of the vector. So, here goes ...

```

;;; Procedure:
;;; vector->list
;;; Parameters:
;;;   vec, a vector
;;; Purpose:
;;;   Builds a list that contains the elements of vec in the same order.
;;; Produces:
;;;   lst, a new list.
;;; Preconditions:
;;;   none
;;; Postconditions:
;;;   The element at position i of lst is the element at position
;;;     i of vec for all reasonable i.

```

```
;;; Does not change vec.
(define vector->list
  (lambda (vec)
    (vector->list-kernel vec 0 (vector-length vec))))
```