Vectors

Summary: In this laboratory, you will explore various aspects of the Vector data type that Scheme provides as an alternative to lists.

Contents:

- **Exercises**
  - Exercise 0: Preparation
  - Exercise 1: Create a Simple Vector
  - Exercise 2: Modifying Lists and Vectors
  - Exercise 3:Specifying Vector Length
  - Exercise 4: Summing Vectors
  - Exercise 5: Filling Vectors
- **If You Have Extra Time**
  - Extra 1: Rotating Vectors
  - Extra 2: Reversing Vectors
  - Extra 3: Rotating Vectors, Revisited
- **Notes**
  - Notes on Exercise 5: Filling Vectors

Exercises

Exercise 0: Preparation

Tell DrScheme not to print the lengths of vectors by entering `(print-vector-length #f)`.

Exercise 1: Create a Simple Vector

a. In DrScheme’s interaction window, type in a vector literal that denotes a vector containing just the two elements 3.14159 and 2.71828. How does DrScheme display the value of this vector?

b. Create a vector that contains the same two values by using the `vector` procedure.

c. Create a vector that contains the same two values by using the `make-vector` and `vector-set!` procedures.

Exercise 2: Modifying Lists and Vectors

Consider the following code.
> (define aardvark (list 1 2 3 4))
> (define baboon aardvark)
> (define aardvark (cons 5 (cdr aardvark)))
> (define chinchilla (vector 1 2 3 4))
> (define dingo chinchilla)
> (vector-set! chinchilla 0 5)

a. What do you expect the output of the following commands to be?

> (list-ref aardvark 0)

_____

> (list-ref baboon 0)

_____ b. Verify your answer experimentally.

c. What do your results suggest about Scheme?

d. What do you expect the output of the following commands to be?

> (vector-ref chinchilla 0)

_____  

> (vector-ref dingo 0)

_____ e. Verify your answer experimentally.

f. What do your results suggest about Scheme?

Exercise 3: Specifying Vector Length

a. Tell DrScheme to print the length of vectors by entering (print-vector-length #t).

b. Enter each of the following vector expressions in DrScheme; consider the result (perhaps by examining individual elements with vector-ref); and indicate what vector has been created.

- #4(0)
- #4(1)
- #4(1 2)
- #2(1 2 3 4)
- (make-vector 4 0)

c. Tell DrScheme not to print the lengths of vectors and reenter each expression. Do your results differ? What do the differences suggest?

Exercise 4: Summing Vectors

Write a procedure, (vector-sum numbers), which takes one argument, a vector of numbers, and returns the sum of the elements of that vector.
You can use `vector->list` from the reading as a pattern for `vector-sum` -- only a few judicious changes are needed. However, you should not use `vector->list` as a helper.

**Exercise 5: Filling Vectors**

In the reading on vectors, we saw that it was possible to implement `list->vector` and `vector->list` by using more primitive operations (particularly `vector-set!` and `vector-length`).

Write your own version of `vector-fill!`. Remember that `vector-fill!` takes two parameters, a vector and a value, and puts that value in every position of the vector.

Note: You may find that you want to do two things for a particular position: fill the value at that position and recurse. Remember that when you want to sequence actions, you should use a `begin` clause: `(begin exp1 exp2)`.

**If You Have Extra Time**

**Extra 1: Rotating Vectors**

Write a procedure, `(rotate! vec)` that rotates the elements in `vec`. That is, `rotate!` puts the initial element of `vec` at the end, the element at position 1 in position 0, the element at position 2 in position 1, and so on and so forth.

**Extra 2: Reversing Vectors**

a. Write a procedure, `(reverse-vector vec)`, that creates a new vector whose elements appear in the reverse order of the elements in `vec`.

b. Write a procedure, `(reverse-vector! vec)`, that reverses `vec` “in place”. That is, instead of producing a new vector, it rearranges the elements within `vec`.

**Extra 3: Rotating Vectors, Revisited**

Write a procedure, `(rotate! vec amt)`, that rotates the values in `vec` by `amt` positions. That is, the first `amt` values in `vec` move to the end, the value in position `amt` moves to position 0, the value in position `amt+1` moves to position 1, and so on and so forth.

**Notes**

**Notes on Exercise 5: Filling Vectors**

Just as in the case of `list->vector`, you will probably want to define a helper procedure that fills only part of the vector. Your termination condition will certainly be different and should probably involve the length of the vector.