COMP 210, FALL 2000
Lecture 33: Vectors

Reminders:

1. Final homework is due next class.
2. Exam 3 handed out Friday. You are responsible for the contents of lab on the exams.

Review
We looked at a couple of examples. We looked at **swap** and talked about how the existence of **set!** reveals things about the implementation of DrScheme that we could not discern without it. [For example, the fact that it creates copies of each argument to a function–-a technique that we refer to as "call-by-value" in COMP 311 and 412.

We looked at an example of set-structure!, and saw where using set-structure! lets us modify data in place rather than rebuilding a large list (such as the white pages).

Vectors
The International Tennis Federation (ITF) provides rankings of the top 100 tennis players. For each player, the organization stores his or her name, home country, and number of matches won. Since people frequently ask for statistics on players according to their rank, the organization wants a program through which they can find the information for a player with a given ranking. The head of information technology for the ITF was talking with the head of JetSet Airlines. She got a strong recommendation to hire the COMP 210 program to develop this software.

Let's develop a data definition and lookup program for this problem.

```
;; A player is a structure
;;   (make-player name home wins)
;; where name and home are symbols and wins is a number
(define-struct player (name home wins))

;; A ranking is a (list of player) containing 100 elements
;;    with the players in ascending rank order
;; find-by-rank : ranking  number[<=100] → player
;; Purpose: returns the player with the given rank, starting from rank 1
(define (find-by-rank a-ranking player-num)
  (local [(define (helper alop at-num)
      (cond [(= at-num player-num) (first alop)]
              [else (helper (rest alop) (add1 at-num))]))]
    (helper a-ranking 1)))
```

You could also have written

```
(define (find-by-rank a-ranking player-num)
  (cond [(= player-num 1) (first a-ranking)]
      [else (find-by-rank (rest a-ranking) (sub1 player-num))]))
```

This program is similar to one built-in to Scheme called list-ref. List-ref consumes a list L and a number N and returns the Nth element in L, counting from 0. We could therefore have written find-by-rank using list-ref as follows:

```
(define (find-by-rank a-ranking player-num)
  (list-ref a-ranking (sub1 player-num)))
```

How long does it take to find a player by her rank? It depends on the rank. Finding the top ranked player requires one call to helper; finding the 100th ranked player requires 100 calls to helper. On average, find-by-rank looks at 50 (50.5?) players to return an answer (assuming that the input requests are uniformly distributed over the numbers from 1 to 100). We should be able to do better than this.

One big hint should come from the fact that we have a finite set of players. This problem cries out for a structure rather than a list, since we know in advance thenumber of data items that will be managed. Let's try that

```
;; a ranking is a structure
;;   (make-ranking  p1 p2 p3 … p100)
;; where the p_i are players
(define-struct ranking p1 p2 p3 … p100)
```

Now, how do we write find-by-rank?

```
;; find-by-rank : ranking  number[<=100] → player
;; Purpose: returns the player with the given rank, starting from rank 1
 (define (find-by-rank a-ranking player-num)
   (cond      [(= player-num 1) p1]
              [(= player-num 2) p2]
              [(= player-num 3) p3]
              …
              [(= player-num 100) p100]
  ))
```

This has some of the right ideas—we don't walk a list of rankings to find the *I*th player, but it also has some of the same problems. How many cond-clauses get evaluated, on average? If we assume normally distributed inputs, the answer should be 50 (or 50.5) All we did was push the complexity into the data definition and convert the recursion into running through the cond-clauses.

The real problem is that we cannot compute the name of an element of the structure. We know the value *i*, and we know that we want p*i* , but we don't have a way to do arithmetic on the names of structure elements.

To solve this problem efficiently, we need a data structure that has the finite nature of structures, but allows us to reference the individual elements using a computed name. As with a structure, the cost of accessing any of the individual elements should be the same, independent of which element we access.

Scheme provides a data structure with these properties. Scheme calls it a ***vector***.

Vectors have a fixed number of components. Components are numbered according to their ordinal position in the vector, and they are accessed by that number. (Because Scheme was written by computer scientists, the first element of a vector is numbered zero, rather than one.) If I want to create a vector that contains my three favorite courses at Rice, I can write

```
(define KeithFavorites (vector `COMP412 `CAAM460 `ENGL314))
```

Given that vector, I can perform several operations on it:

1. Find out how many components it contains:

```
(vector-length KeithFavorites)
> 3
```

2. Retrieve its nth component (counting from 0)

```
(vector-ref KeithFavorites 1)
>`CAAM460
```

Just like list-ref, except it doesn't have to walk the

3. Update its nth component to a new value

```
(vector-set! KeithFavorites 0 `COMP210)
KeithFavorites
> (vector `COMP210 `CAAM460 `ENGL314)
```

If we want to initialize the values in a vector using a function, Scheme provides us with a built-in function called **build** vector. It consumes a number (desired length) and a function that takes an index in the vector into a value. The program build-vector returns a vector of the specified length where each element is set to the value that the function returns for that element's ordinal index. Thus (build-vector n f) is equivalent to

```
(vector (f 0) (f 1) ... (f (- n 1))

(build-vector 5 (lambda (i) (* i i)))
(vector 0 1 4 9 16)
```

Since vector-ref uses the same amount of work to access every element, regardless of its position in the vector, a vector might be the right data structure for our program find-by-rank. In general, it makes sense to use vectors when:

1. the number of components is fixed,
2. uniform access to components is a serious benefit, and
3. numbers are a natural way to index the components.

Thus, vectors are good for problems involving rankings of fixed numbers of elements, such as our tennis organization problem. However, they are bad for problems such as address books, because the numbers are not a natural way to index the entries.

Let's redesign our rankings program using vectors. The data definition for players stays the same.

```
;; A ranking is a vector of 100 players

;; find-by-rank : ranking number[<=100] → player
;; returns the player with the given rank, starting from rank 1
(define (find-by-rank a-ranking player-num)
  (vector-ref a-ranking (sub1 player-num)))
```

Index starts at 0!

How would you create a ranking? We could write a function to create an empty ranking, and another to update a ranking with a particular player in a particular spot:

```
;; make-ranking : number → vector
;; Purpose:  creates a vector with all components
;;                initialized to false
(define (make-ranking size)
   (build-vector size (lambda (i) false)))

;; rank-player! : ranking number player → void
;; Purpose: fill the rank specified by the number argument with
;;               the player argument
;;
;; effect : changes value of ranking in position rank to player
(define (rank-player! a-ranking rank a-player)
  (vector-set! a-ranking rank a-player))
```