COMP 210, Spring 2001
Lecture 21: Vectors and Mutable State

Reminders:

1. Big programming assignment, Missionaries and Cannibals, due April 6 (a week from next Wednesday) in class.

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)))
```

We 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 function is similar to one built-in to Scheme called `list-ref`, which consumes a `(list-of T) loi` and a number `n` and returns the `n`th element in `loi`, 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 ½ players to return an answer (assuming that the input requests are uniformly distributed over the numbers from 1 to 100). If ITF expanded its ranking table to include the top 1000 players, we would inspect at 500 ½ players on average to return an answer. We should be able to find the specified player in constant time independent of the number of players included in the ranking.

One big hint should come from the fact that we have a fixed set of players. This problem cries out for a structure rather than a list, since we know in advance the number of data items that will be managed. Let's redefine the **`ranking`** data type:

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

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

```
; 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)
  (cond [(= player-num 1) p1]
        [(= player-num 2) p2]
        [(= player-num 3) p3]
        …
        [(= player-num 100) p100]))
```

This program is on the right track. We can directly select a player with a given ranking, but we have no mechanism for mapping a number to the corresponding selector function without performing a sequence of tests. The program above evaluates 50 ½ comparisons on average to select a specified player.

Finger exercise: can we map numbers to selectors more efficiently than we did in the code above which runs in time O(N) where N is the number of selectors.

What we need is a structure where the selector names are numbers. A Scheme **`vector`** is precisely such a structure. A vector consists of a fixed number of fields indexed by consecutive natural numbers starting with 0 (just like the **`list-ref`** function accesses lists). Hence the fields of a vector of 100 elements are indexed by the natural numbers 0, 1, …, 99. As in a structure, the cost of accessing any element of a vector is the same, independent of which element we access.

Essentially every programming language includes this form of data structure; in Fortran, Algol, Pascal, C, C++, Java, this form of data structure is called an *array*.

Scheme provides several different primitive operations for constructing vectors. For example, we could define a vector containing the names of the core courses in the Computer Science major:

```
(define core-courses
  (vector `Comp210 `Comp212 `Comp 280 'Comp320 'Comp314))
```

Given this vector, we can perform several operations on it:

1. Find out how many components it contains:

```
     (vector-length core-courses) = 5
```

Just like list-ref, except it doesn't have to walk the entire vector. (Efficiency)

2. Retrieve the kth component (counting from 0)

```
     (vector-ref core-courses 2) = 'Comp280
```

3. Build a vector of length **n** containing the elements **(f 0)**, **(f 1)**, …, **(f (- n 1))** generated by a function **f**

```
     (build-vector n f)
```

For example,

```
     (build-vector 5 square) =
     (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 is the ideal 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 important, and
3. numbers are a convenient mechanism for indexing 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 should we create a ranking? We could use a huge invocation of the **`vector`** operation

```
 (vector (make-player `Sampras `Florida 3)

   . . . )
```

but this is clumsy and error prone. How do we check that the correct player is listed in 49th postion? In addition, we need a mechanism for updating a ranking as the season unfolds. What can we do to initialize the vector more simply and update it when new information come is?

Mutate (destructively modify) the ranking vector!

Scheme includes two more operations on vectors:

1. The operation

   ```
   (make-vector N)
   ```

   constructs a vector containing **`N`** elements. Each element of the constructed vector is initialized to 0.

2. The operation

   ```
   (vector-set! V n val)
   ```

   sets (modifies!) the **`n`**th element of **`V`** to **`val.`** For example, given the vector **`core-courses`** defined above,

   ```
   (vector-set! core-courses 4 `Comp312)
   ```

   changes the value of **`core-courses`** from

   ```
   (vector `Comp210 `Comp212 `Comp 280 ‘Comp320 ‘Comp314)
   ```

   to

   ```
   (vector `Comp210 `Comp212 `Comp 280 ‘Comp320 ‘Comp312)
   ```

Mutating a vector destructively changes the contents of the vector object; the old value of the updated element is removed.

Given these operations, we can construct a ranking as follows:

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

```
; rank-player! : ranking number player → void
;   fills 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))
```

When a Scheme operation destructively modifies a data object, we must document this change using an **`effect`** comment.

Other Mutation Operators

The **`vector-set!`** operation is not unique. Scheme includes operators for modifying the values of fields of structures and for modifying the values of variables. For each field of a structure, there is a field update operator. For example, given the structure definition

```
(define-struct Pair (left right))
```

Scheme creates update operators **`set-Pair-left!`** and **`set-Pair-right!`** That modify the contents of a **`Pair`** structure. These operators behave just like **`vector-set!.`** For example,

```
(define p (make-Pair 1 2))
(set-Pair-left! p 2)
p
```

evaluates to

```
(make-Pair 2 2)
```

The **`set!`** operator is different. The first argument to **`set!`** must be a variable name; it cannot be an expression because variable names are not *values.* The **`set!`** operator does not evaluate its first argument. Hence, **`set!`** is not a Scheme function; it is a *special form* like **`define`** and **`cond`**. Consider the following simple example:

```
(define x 1)
(set! x 7)
x
```

evaluates to

```
7
```

A More Interesting Example

```
(define p (make-Pair 1 2))
(define q p)
(set-Pair-left! p 2)
```

```
q
```

evaluates to

```
(make-Pair 2 2)
```

A Brief Warning

We have avoided mutation operations like **`vector-set!`** and **`set!`** up to this point in the course precisely because it makes reasoning about the behavior of programs much more difficult. Mutation operations lets us write programs (and expressions) whose results depend on things that happened earlier in the computation. (Perhaps, earlier in another computation…) This makes the simple rewriting rules for Scheme that we have used so far in the course somewhat more complex. It doesn't completely change the way that things work, but it does require that we keep track of much more context–-a subtle and difficult task, at best.

Let's show how our existing evaluation rules break! …

To reason about what a program does, in a world that includes mutation operators, we need to keep track of what happens any time a mutation occurs during execution. In addition, we must keep track of sharing relationships among data structures that may be mutated. This is a lot more complex than just copying over the arguments, textually, as we make successive calls. You can write a program that goes deep into some recursion, does a mutation operation, and returns. If that mutation operation changed the value of a variable that is used elsewhere in the computation, you might not recognize it, or, even, be aware of it.

Consider the following simple program:

```
; mystery: number -> number
;   performs some inscrutable computation
(define mystery
  (local
      [(define memory 1)]
    (lambda (x)
      (begin
        (set! memory (add1 memory))
        (* memory x  3/4)))))
```

Note: the expression

```
(begin M1 … Mn)
```

abbreviates

```
((lambda (x1 … xn) xn) M1 … Mn)
```

What does the **`mystery`** program do? It is hard to derive its operation by calling it with a few trial arguments!

```
(mystery 1)   =  3/4
(mystery 2)   =  3
(mystery 3)   =  27/4
(mystery 100) =  300
(mystery 100) =  375
```
… and so on, …

Thus, you should only use **`vector-set!`** and other mutation operations in carefully chosen and carefully planned ways. The next several lectures will address some of those issues. Remember, the **`!`** is a warning–to both the programmer and the reader.