Administrative Notes

Extra credit homework

- Due Today

Third exam

- Available now
- Due next Wednesday, April 24, at 5 pm *outside my office*
- Closed notes, closed book
- Material since second exam, through today

Next Week

Of course, I cannot test you on next week's material

- Look at imperative programming
  - → We've indoctrinated you into the functional style
  - → We've (finally) let you use set! *(+ set-structure! & vector-set!)*
- The functional patterns work in imperative programming
  - → They produce working code
  - → The code can be inefficient

  Assignment is often an efficiency hack
- We'll study quicksort
  - → Rewrite it to use vectors *(more practice with vector)*
  - → Rewrite it to use set! well *(thinking in imperative terms)*
  - → Rewrite it in C *(introduce you to C)*

Last Class

Wrote code to maintain rankings for the ITF

- Limited number of data items (100 players)
- Need for efficient random access to data on players
- Led to vectors

Today

- Couple of applications for vectors
- Brief review for exam

Vectors

Interface

- vector is analogous to list

  (define KeithFavorites (vector ‘COMP412 ‘CAAM460 ‘ENGL317))

- vector is supported by several functions
  - → vector-length (vector-length KeithFavorites) ⇒ 3
  - → vector-ref (vector-ref KeithFavorites 2) ⇒ ‘ENGL317
  - → vector-set! (vector-set! KeithFavorites 0 ‘COMP210)

- Initializer: build-vector: num (num->num) -> vector

  (build-vector 5 (lambda(x)(* x x))) ⇒ (vector 0 1 4 9 16)

Applications of Vector

Linear Algebra

- Vectors are a common abstraction in mathematics
  - → What's the common name for Math 212?
- A vector is a k-tuple of scalars (*numbers*)
  - → Specifies a point in vector space
- Important operations on vectors
  - → Scalar arithmetic : $s \times v$ or $s + v$
  - → Vector arithmetic : $v \times w$ or $v + w$

Applications of Vector

Scalar arithmetic

- Scalar-vector addition

```
;; scalar-add : number  vector of number -> vector of number
;; Purpose:  compute the sum of a scalar and a vector
(define (scalar-add a-num a-vec)
   (build-vector (vector-length a-vec) (lambda(i)(* a-num (vector-ref a-vec i)))))
```

Need to create a vector to hold the result

Must be "conformable" to input vector

Do the actual work in an anonymous initialization function !

Applications of Vector

Scalar arithmetic

- Scalar-vector addition

```
;; scalar-add : number  vector of number -> vector of number
;; Purpose:  compute the sum of a scalar and a vector
(define (scalar-add a-num a-vec)
  (build-vector (vector-length a-vec) (lambda(i)(+ a-num (vector-ref a-vec i)))))
```

- Scalar-vector multiplication

```
;; scalar-mult : number  vector of number -> vector of number
;; Purpose:  compute the product of a scalar and a vector
(define (scalar-mult a-num a-vec)
   (build-vector (vector-length a-vec) (lambda(i)(* a-num (vector-ref a-vec i)))))
```

Code is quite similar ⇒ *Create an abstract function*

Applications of Vector

Abstracting scalar-add and scalar-mult

- Scalar arithmetic

```
;; scalar-arith : num  vector of num (num num -> num) -> vector of num
;; Purpose:  apply function argument to vector and scalar, elementwise
(define (scalar-arith a-num a-vec an-op)
  (build-vector (vector-length a-vec)
                (lambda(i)(an-op a-num (vector-ref a-vec i)))))
```

What changed?

And we can rewrite scalar-add & scalar-mult appropriately …

```
;; scalar-add : num  vector of num  -> vector of num
(define (scalar-add s v) (scalar-arith s v +))

;; scalar-mult : num  vector of num -> vector of num
(define (scalar-mult s v) (scalar-mult s v *))
```

Applications of Vector

Vector arithmetic

- Follows in a straightforward fashion

```
;; vector-arith: vector  vector  (num num -> num) -> vector
;; Purpose:  apply function argument to two vectors
(define (vector-arith vec1 vec2 an-op)
  (build-vector (vector-length vec1)
                      (lambda(i) (an-op (vector-ref vec1 i) (vector-ref vec2 i)))))
```

Assume that vec1 & vec2 are conformable

And we can write vector-add & vector-mult …

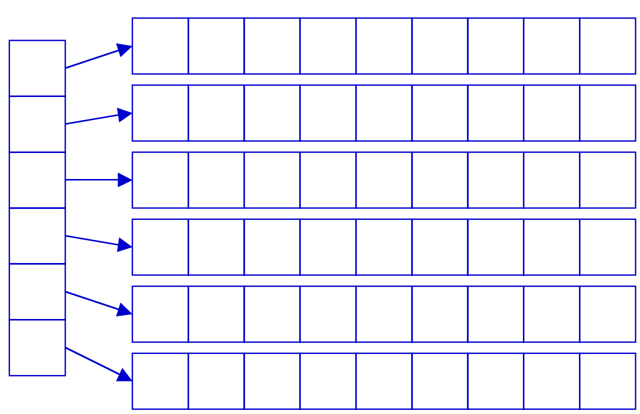
```
;; vector-add : vector of num  vector of num  -> vector of num
(define (vector-add v1 v2) (vector-arith v1 v2 +))

;; vector-mult : vector of num  vector of num -> vector of num
(define (vector-mult v1 v2) (vector-mult v1 v2 *))
```


What about Arrays?

Array is either

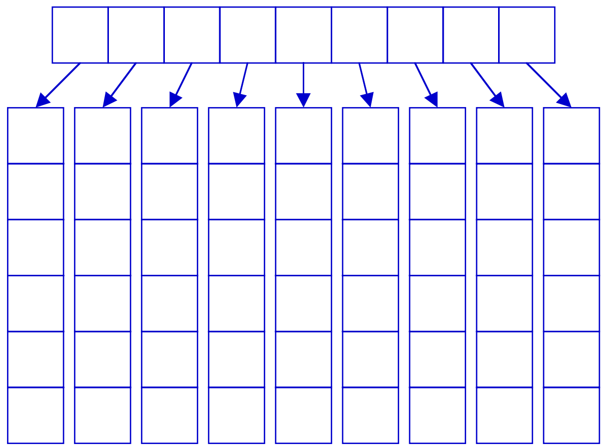
- Vector of columns, where column is vector



What about Arrays?

Array is either

- Vector of columns, where column is vector, *or*
- Vector of rows, where row is a vector



What about Arrays?

Array is either

- Vector of columns, where column is vector, *or*
- Vector of rows, where row is a vector

Clever student can build arrays using the initializer

- Call build-vector inside build-vector
- Must use nested vector-ref and vector-set! operations
  - → A little awkward, but you can write your own interface

This is the way that Java does it
(early C did this, too)

Material for the Exam

You are responsible for:

- Contents of lecture — both class and lab lecture
- Sections 25 to 43 in the book (*as it relates to lecture*)

- All lecture notes are online, except John's lecture on binary search
- Fall 2000, Exam 3 is online
- Lab lecture notes are up-to-date online

Every test (so far) has had

- Question on each major topic
- Question drawn from lab lectures

Material for the Exam

The major topics since the second exam include:

- Generative recursion
  - → Binary search, find-flights, …
- Accumulators
  - → Reverse, max (*not accumulators on trees*)
- Local state
  - → Memo-functions, address-book
- Data-hiding and abstraction
  - → Address-book generator
- Equality (equal? versus eq?)
- Vectors

Too many topics
- *4 questions on 4 topics*
- *Extra credit*

Adds a little suspense …