

Data Abstraction

- Abstract Data
- Hierarchical Data
- Representing Abstract Data

Abstract Data

- Data Abstraction: Who, What, Where, When, Why
- Example: two-dimensional vectors
- `cons`, `car`, `cdr`
- `list`
- `quote`

Data Abstraction

- First section of course: building compound procedures
- Now: Compound Data
- Elevate the level at which we design programs
- Increase modularity
- Enhance expressive power

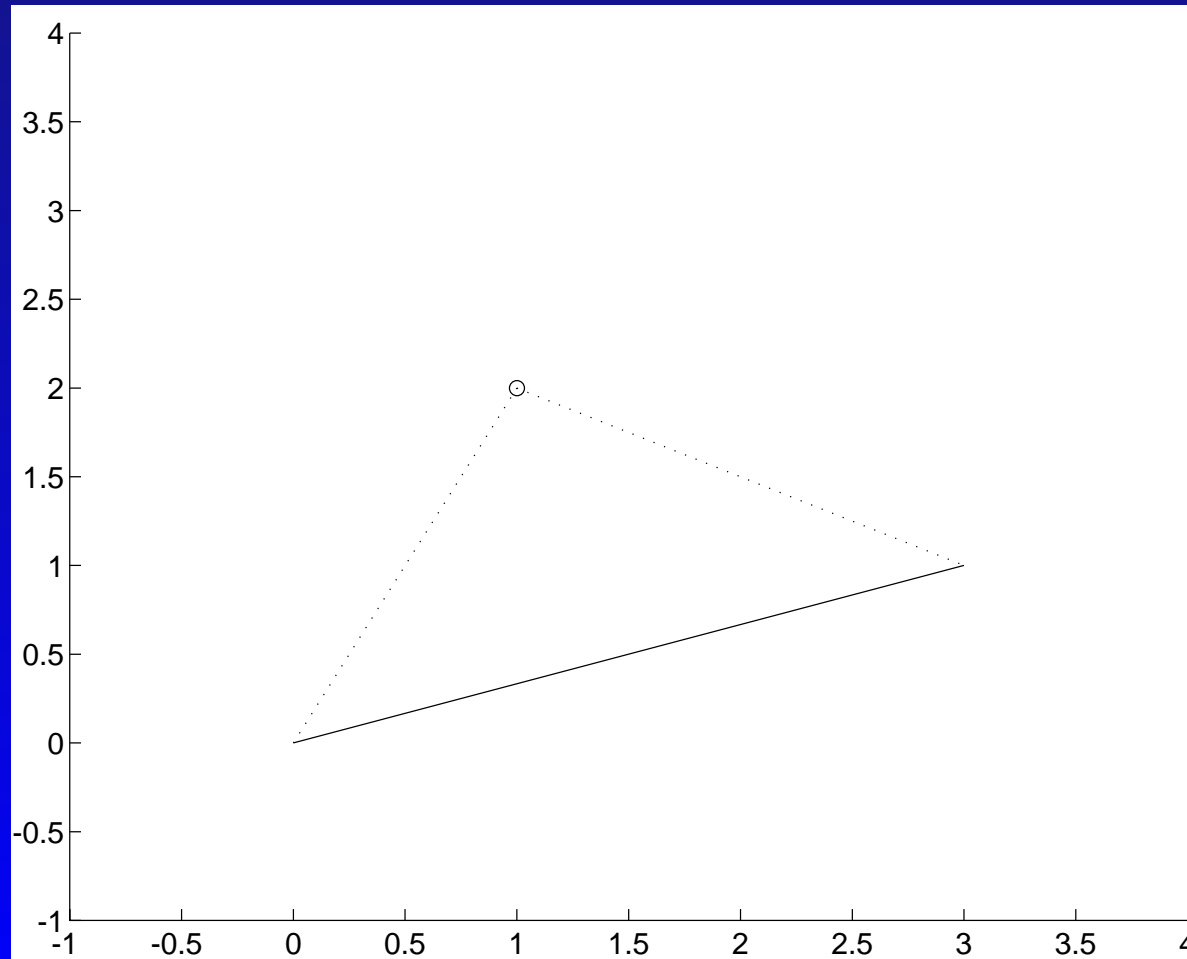
Compound Data

- Basic idea: structure programs that use compound data so that they operate on “abstract data” i.e., so they make no unnecessary assumptions about the data
- At the same time, there must be a “concrete” representation of the data that is independent of the programs using the data

Compound Data

- The interface between abstract representation and concrete representation
 - Constructors: Creating instances of the data
 - Selectors: Accessing pieces of the data

Example: Two-Dimensional Vectors



Example: Two-Dimensional Vectors

- Let's build a system for working with two-dimensional vectors
- Let's assume we already have a way of constructing a vector from its components
- Let's also assume we have a way of extracting the components from a vector

Example: Two-Dimensional Vectors

- The constructor and selector procedures

`(make-vector $\langle x \rangle$ $\langle y \rangle$)` returns a vector whose coordinates are $\langle x \rangle$ and $\langle y \rangle$

`(x-coord $\langle v \rangle$)` returns the first coordinate of vector $\langle v \rangle$

`(y-coord $\langle v \rangle$)` returns the second coordinate of vector $\langle v \rangle$

Example: Two-Dimensional Vectors

- What sorts of procedures would we want in this system?
- What sorts of things do we want to do with vectors

Example: Two-Dimensional Vectors

- What sorts of procedures would we want in this system?

Example: Two-Dimensional Vectors

- What sorts of procedures would we want in this system?
- What sorts of things do we want to do with vectors
 - Add and subtract them
 - Test them for equality
 - Multiply a vector by a scalar
 - Take the dot product of two vectors
 - Compute the length of a vector
 - Print a vector

Example: Two-Dimensional Vectors

- What sorts of procedures would we want in this system?
 - (add-vector $\langle v \rangle$ $\langle w \rangle$)
 - (sub-vector $\langle v \rangle$ $\langle w \rangle$)
 - (vector= $\langle v \rangle$ $\langle w \rangle$)
 - (scalar*vector $\langle c \rangle$ $\langle v \rangle$)
 - (dot $\langle v \rangle$ $\langle w \rangle$)
 - (length $\langle v \rangle$)
 - (print $\langle v \rangle$)

Example: Two-Dimensional Vectors

- Do we need to know anything about the vectors besides the constructors and selectors
- Should we use any information about the vectors besides the constructors and selectors
- Why or why not?

Cooperative Exercise

- Implement the two-dimensional vector package
- Each team take one procedure

Concrete Representation of Two-Dimensional Vectors

- Even though we can implement the vector package using just constructors and selectors
- We need an actual concrete representation

Concrete Representation of Two-Dimensional Vectors

- Scheme provides one basic compound data type (which is itself abstract) called a *pair* with constructor `cons` and selectors `car` and `cdr`
 - `(cons <x> <y>)`
 - `(car <p>)`
 - `(cdr <p>)`

Concrete Representation of Two-Dimensional Vectors

- The data in a pair can be any Scheme data, including procedures and other pairs

Pairs

- Example

```
1 ]=> (define x (cons 1 2))
```

```
;Value: x
```

```
1 ]=> (car x)
```

```
;Value: 1
```

```
1 ]=> (cdr x)
```

```
;Value: 2
```

Pairs

- Example

```
1 ]=> (define x (cons 1 2))
```

```
1 ]=> (define y (cons 3 4))
```

```
1 ]=> (define z (cons x y))
```

```
1 ]=> (car (cdr z))
```

```
1 ]=> (car (car z))
```

Pairs

- Example

```
1 ]=> (define z (cons x y))
```

```
;Value: z
```

```
1 ]=> (car (cdr z))
```

```
;Value: 3
```

```
1 ]=> (car (car z))
```

```
;Value: 1
```

Concrete Representation of Two-Dimensional Vectors

- Implement `make-vector`, `x-coord`, and `y-coord` using pairs

Concrete Representation of Two-Dimensional Vectors

- Implement `make-vector`, `x-coord`, and `y-coord` using pairs

```
(define (make-vector x y) (cons x y))
```

```
(define (x-coord v) (car v))
```

```
(define (y-coord v) (cdr v))
```

Concrete Representation of Two-Dimensional Vectors

- Implement `make-vector`, `x-coord`, and `y-coord` using `pairs`

```
(define make-vector cons)
```

```
(define x-coord car)
```

```
(define y-coord cdr)
```

Concrete Representation of Two-Dimensional Vectors

- Is there any “right” answer?
- What is the *key* issue?

Concrete Representation of Two-Dimensional Vectors

- Is there any “right” answer?
- What is the *key* issue?
 - That the constructors and selectors work like they are supposed to

Abstraction Barriers

- From the example you can see the notion of abstraction barriers
- A program using our vector package would only use the operators we provide
- The operators use only constructors and selectors we provide
- The constructors and selectors use Scheme primitives

Abstraction Barriers

- Each functional interface separates levels of abstraction in the program construction and represent *abstraction barriers*
- Breaking through an abstraction barrier is an *abstraction violation*

Abstraction Violation

- Example

```
(define (vector+ v w)
  (cons
    (+ (car v) (car w))
    (+ (cdr v) (cdr w))))
```

- Would this work in our implementation?
- Why is this bad?

Abstraction Violation

- Example

```
(define (vector+ v w)
  (cons
    (+ (car v) (car w))
    (+ (cdr v) (cdr w))))
```

- Would this work in our implementation?
 - Maybe
- Why is this bad?
 - Changes in concrete implementation would break it

Abstraction Violation

- Rule of thumb
- Your program should never depend on (or use) what is on the other side of an abstraction barrier
- **Warning**
- Graders will be looking for abstraction violations

Pairs

- For instance, it doesn't matter how `cons`, `car`, and `cdr` are implemented
- In fact, we could implement these using only procedures!

Pairs

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else
           (error "Bad arg: cons" m)))))
```

```
(define (car x) (x 0))
```

```
(define (cdr x) (x 1))
```

Pairs

- Huh?!?

Pairs

- Claim: These definitions are just fine because they do what they are supposed to.

- Evaluate

```
1 ]=> (define x (cons 8 9))
```

```
1 ]=> x
```

```
1 ]=> (car x)
```

```
1 ]=> (cdr x)
```

Pairs

- Claim: These definitions are just fine because they do what they are supposed to

```
1 ]=> x
```

```
;Value: #[compound-procedure 3]
```

```
1 ]=> (car x)
```

```
;Value: 8
```

```
1 ]=> (cdr x)
```

```
;Value: 9
```

Pairs

- Consider

```
1 ]=> (car x)
;      (x 0)
;      ((lambda (m)
;         (cond ((= m 0) 8)
;               ((= m 1) 9))))
;      0)
;Value: 8
```

Pairs

- Is `cons` a special form?

Pairs

- Is `cons` a special form?
 - Why or why not?
 - How can you tell?

Lists

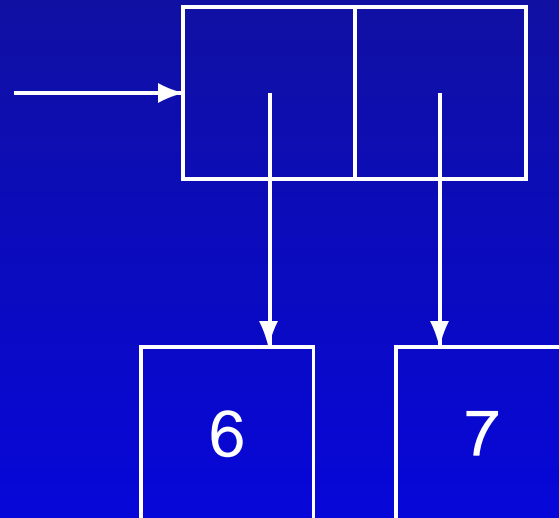
- Early in the course, we said LISP = LISt Processing
- Whence lists?
- Built up from pairs!

Box and Pointer Notation

- We can represent pairs graphically with boxes and pointers
- A box is used to represent the pair
- The pointers point to the data in the pair
- These are only graphical pointers — don't confuse them with the notion of pointers in C

Box and Pointer Notation

- Box and pointer diagram for (cons 6 7)

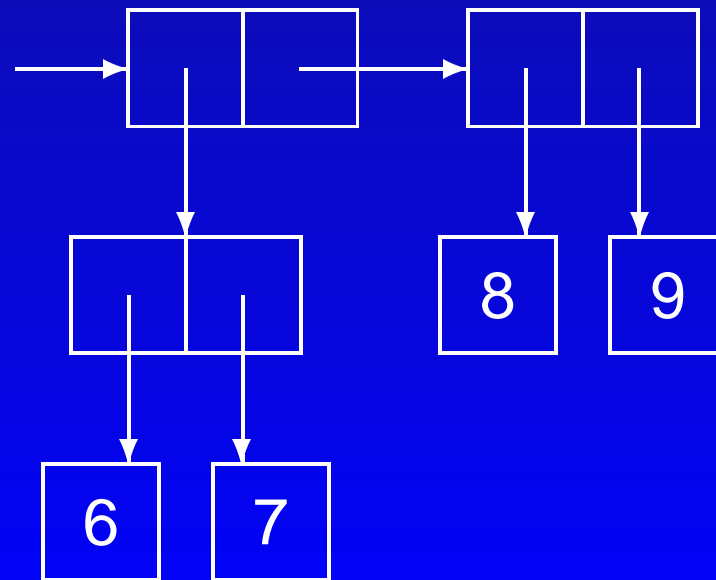


Box and Pointer Notation

- Box and pointer diagram for `(cons (cons 6 7) (cons 8 9))`

Box and Pointer Notation

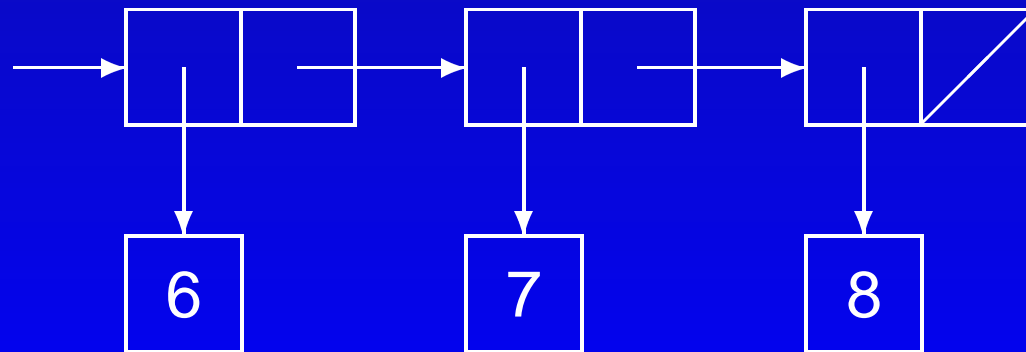
- Box and pointer diagram for `(cons (cons 6 7) (cons 8 9))`



Lists

- Lists are sequences of pairs (`nil` is the empty list/pair)

```
(cons 6  
  (cons 7  
    (cons 8 nil)))
```



Lists

- Scheme provides a primitive `list` for constructing lists
- Previous example

```
(list 6 7 8)
```

Lists

- In general

```
(list < a1 > < a2 > ... < an >)
```

- Equivalent to

```
(cons < a1 >  
  (cons < a2 >  
    (cons ...  
      (cons < an > nil) ...)))
```

Lists

- Thus, to draw box and pointer diagram for a list
- First make a “backbone” of pairs, one pair for each element of the list
- Each datum in the list will be the `car` of one of the pairs

Lists

- Scheme prints lists as a sequence of elements, enclosed in parentheses

```
1 ]=> (list 6 7 8)
```

```
;Value: (6 7 8)
```

Cooperative Exercise

- Draw a box-and-pointer diagram for the vector `i` defined as

```
(define i (make-vector 1 0))
```

Cooperative Exercise

- Describe the result of calling the following procedure with a list as its argument.

```
(define (mystery lst)
  (mystery-helper lst nil))
```

```
(define (mystery-helper lst other)
  (if (null? lst)
      other
      (mystery-helper (cdr lst)
                       (cons (car lst) other))))
```

List Operations

- Conventional programming techniques for lists
- “`cdr`” down the list
- “`cons`” up a new list

List Operations

- Example: Get the n th element of a list
- Use wishful thinking

```
(define (nth n x)
  . . . )
```

List Operations

- Example: Get the n th element of a list
- Assume we can get $n - 1$ st element of a shorter list (the rest of the list)
- `(cdr x)` is the shorter list

```
(define (nth n x)
```

```
  . . .
```

```
    (nth (- n 1) (cdr x)))
```

List Operations

- Example: Get the n th element of a list
- Add the base case

```
(define (nth n x)
  (if (= n 0)
      (car x)
      (nth (- n 1) (cdr x))))
```

List Operations

- Example: Get the n th element of a list

```
(define (nth n x)
  (if (= n 0)
      (car x)
      (nth (- n 1) (cdr x))))
```

List Operations

- Example: Get the length of a list

```
(define (length x)
```

```
)
```

List Operations

- Example: Get the length of a list

```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))
```

List Operations

- Example: append a list y to a list x

```
1 ]=> (append (list 6 7 8)
           (list 1 2 3))
```

```
;Value: (6 7 8 1 2 3)
```

```
1 ]=> (append (list 1 2 3)
           (list 6 7 8))
```

```
;Value: (1 2 3 6 7 8)
```

List Operations

- Example: append a list y to a list x
- What is the wishful thinking step?

```
(define (append x y)
```

```
)
```

List Operations

- Example: append a list y to a list x
- What is the wishful thinking step?

```
(define (append x y)
```

```
  . . .
```

```
    (cons (car x) (append (cdr x) y)))
```

List Operations

- Example: append a list y to a list x
- What is the base case?

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Trees

- Natural generalization of lists: Lists of lists
- Lists of lists: *Trees*
- Elements of the sequence are branches
- Elements that are also sequences are subtrees

Trees

- Draw box and pointer diagram for

```
1 ]=> (cons (list 8 7) (list 6 5))
```

Trees

- Draw tree structure for

```
1 ]=> (cons (list 8 7) (list 6 5))
```

Working with Trees

- Operations on trees \rightarrow operations on their branches
- Operations on branches \rightarrow operations on subtrees
- *Recursion!*

Working with Trees

- To aid in working with trees, Scheme provides primitive predicate `atom?`
- `atom?` tests whether its argument is *atomic* (not decomposable, i.e., not a pair)

Tree Operations

- Similar to `length`, write a procedure that counts the atoms in a tree

```
(define (countatoms x)
```

```
)
```

Tree Operations

- Similar to `length`, write a procedure that counts the atoms in a tree
- What is the wishful thinking step?

```
(define (countatoms x)
  . . .
  (+ (countatoms (car x))
     (countatoms (cdr x))))
```

Tree Operations

- Similar to `length`, write a procedure that counts the atoms in a tree
- What is the base case?

```
(define (countatoms x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else
         (+ (countatoms (car x))
            (countatoms (cdr x))))))
```