

## Message passing paradigm

- A way of using procedure abstraction to implement data abstraction
- A procedure is used to represent an object
- A higher-order procedure is used to act as a constructor
- A message is passed to the object (value passed as input to the procedure) to act as a selector

## How pairs could be implemented

```
(define (cons x y)
  (define (dispatch message)
    (cond ((= message 0) x)
          ((= message 1) y)
          (else
           (error "bad message"
                  message))))
  dispatch)
```

## Implementing the selectors

```
(define (car z) (z 0))
(define (cdr z) (z 1))

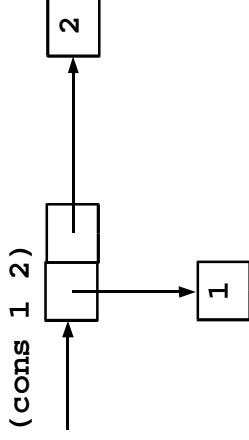
("Don't try this at home!")
```

## Alternate version of cons

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y)
          (else
           (error "bad message"
                  message)))))
```

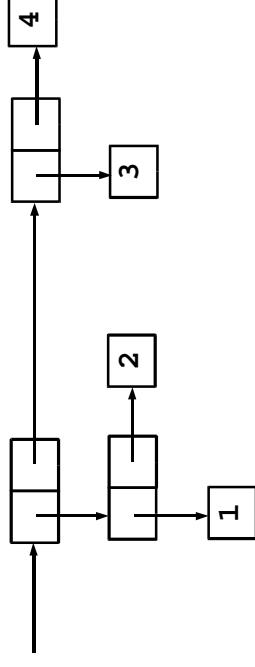
## Box and pointer notation

- Draw cdr pointers to the right
- Draw car pointers downward



## Another list structure

(cons (cons 1 2) (cons 3 4))



## The closure property

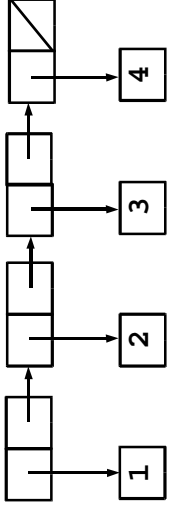
- A constructor has the **closure property** if it can take data of a certain type as input and return data of the same type
- cons is an example
- Such constructors can be used to build **hierarchical structures**

## Lists, a recursive data type

- The empty list is a list
- If  $x$  is any datum and  $y$  is a list, then (cons  $x$   $y$ ) is a list
- The empty list is denoted by () in DrScheme and by nil in the course textbook
- Whenever you see nil in the book, read ()

## What do lists look like?

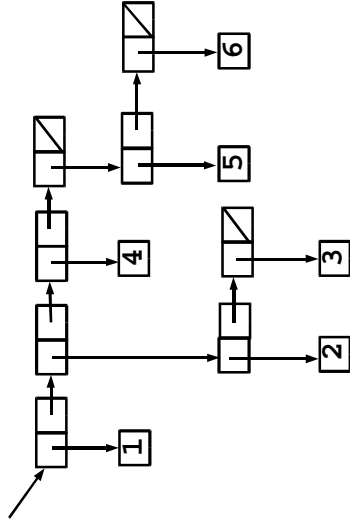
```
(cons 1 (cons 2 (cons 3 (cons 4
  ())))))
(1 2 3 4)
```



## Lists can contain lists

```
(cons 1
  (cons (cons 2 (cons 3 ()))
    (cons 4
      (cons (cons 5 (cons 6 ()))
        ())))))
(1 (2 3) 4 (5 6))
```

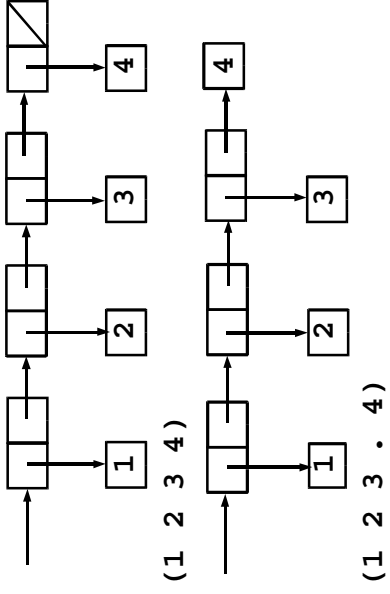
## Box and pointer representation



## Printing out list structures

Printed like lists, but if the last `cdr` in a `cdr` chain points to a primitive datum other than `()`, the primitive datum is printed with a dot in front of it

## A comparison



## Some service procedures for lists

```
(list 1 2 3 4) --> (1 2 3 4)
(define (list-ref list n)
  (if (= n 0)
      (car list)
      (list-ref (cdr list)
                 (- n 1))))
(list-ref (list 1 2 3 4) 0) --> 1
(list-ref (list 1 2 3 4) 2) --> 3
```

## More service procedures

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

## Append

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1)
                    list2))))
(append (list 1 2) (list 3 4))
--> (1 2 3 4)
```



## Mapping over lists

```
(define (list-of-squares x)
  (if (null? x)
      ()
      (cons (* (car x) (car x))
            (list-of-squares
              (cdr x)))))
(list-of-squares (list 1 2 3 4 5))
--> (1 4 9 16 25)
```

## A map procedure

```
(define (map proc list)
  (if (null? list)
      ()
      (cons (proc (car list))
            (map proc
                  (cdr list)))))
(define (list-of-squares list)
  (map (lambda (x) (* x x)) list))
```

## Scheme's map procedure

```
(map <procedure taking N arguments>
     <list 1>
     <list 2>
     ...
     <list N>)
```

All lists must have the same length.

## Using map

```
(define (vector-sum vec1 vec2)
  (map + vec1 vec2))
(vector-sum (list 1 2 3 4)
            (list 5 6 7 8))
--> (6 8 10 12)
```