Data abstraction

Constructor

Accessor/selector

Contract!

Operations

Pairs and list

`cons`

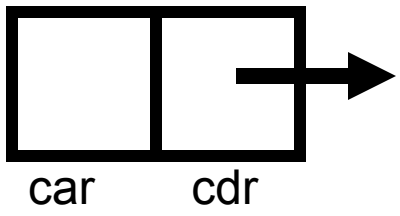
`car`

`cdr`

`list`

`'()`

box and pointer diagram




```
(define nil '())
```

Other accessors

Shortcuts: c?????r. ex:

```
(caddr X):(car (cdr (cdr X)))
```

List access:

```
first, second etc.
```

How could you define first, second, third, and fourth using the c???r function?

first: car

second: cadr

third: caddr

Order of growth?

```
(define (dummy x)
  (if (<= 1 x) x
      (+ (dummy (floor (/ x 2))
         (dummy (- x (floor (/ x 2)))))
```

Linear complexity!

2 recursive calls (would suggest 2^n)

But each reduces the problem to half the complexity, (log style)

Give a box/arrow diagram and the scheme printout

```
(cons 1 2)
(cons 1 (cons 2 nil))
(cons 1 nil)
(cons 1 (cons 2 3))
(cons (cons 1 2) nil)
(list 1 2 3 4)
(list 1 (cons 2 3) (list 4 5))
```

Draw a box and pointer diagram

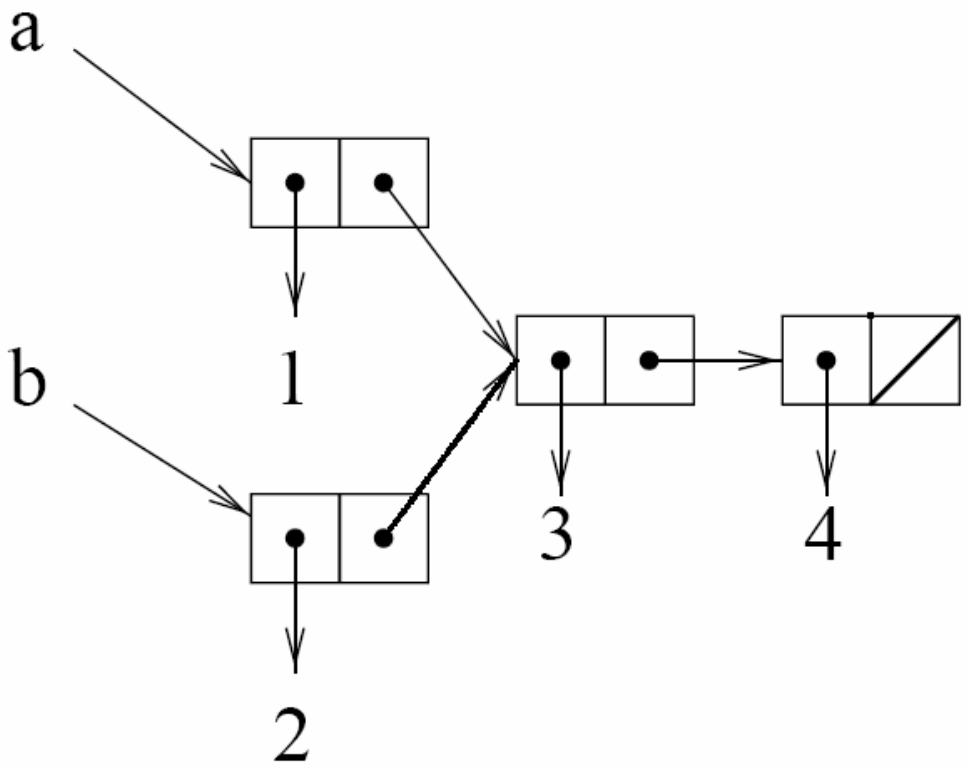
```
(define a (list 1 2 3 4))
(define b
  (list 5 (cdr (cdr a)) (cons 6 7)))
```

Write a Scheme expression that will print each of the following. Also draw box and pointer diagrams.

```
(list 1 2 3)
> (1 2 3)
(cons (cons 1 2) 2)
> ((1 . 2) . 2)
(cons (list 1 (list 2)) 3)
> ((1 (2)) . 3)
```

Write scheme expressions to get the following diagram

```
(define a (list 1 3 4))
(define b (cons 2 (cdr a)))
```



We saw that we have the primitive function pair? to see if an object is a pair. What if we wanted to write the function list? to see if an object is a list ?

Assume you have a predicate null? to test the empty list

```
(define list? (lambda (L)
  (or (null? L)
      (and (pair? L) (list? (cdr L))))))
```

What is the Order of Growth of pair? and list? ?

pair?: constant in time and space

list?: linear in time, constant in space.

Write a procedure to compute the length of a list

Recursive?

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

Iterative?

```
(define length (lambda(L)
  (define helper (lambda(L cur)
     (if (null? L) cur
          (helper (cdr L) (+ 1 cur)))))
  (helper L 0)))
```

Write a procedure square-list that takes a list of integers as input and returns the list of squared values

```
(define square-list (lambda(L)
  (if (null? L) ()
     (cons (square (car L))
           (square-list (cdr L))))))
```

What if we wanted to reference the element of a list? Write the function list-ref that takes a list x and an integer n and returns the nth element of the list x.

```
(define list-ref (lambda(L n)
  (if (null? L) (error "list too small")
     (if (= 0 n) (car L)
          (list-ref (cdr L) (- n 1))))))
```

Consider the procedure copy which takes a list and returns a copy of the list. How do each of the following differ?

```
(define (copy-ident x) x)
```

This one does not create a new copy. It just points to the old one !

```
(define (copy-recurse x)
  (if (null? x) nil
    (cons (car x)
      (copy-recurse (cdr x)))))
```

This one does the job. It does actually create a new list which contains the same values.

Notice that copy-recurse is a recursive process. Write an iterative copy! Warning, it's tough. See next slide.

Warning: the below is not copy!

```
(define (*copy-iter* x)
  (define (aux x ans)
    (if (null? x) ans
      (aux (cdr x)
           (cons (car x) ans)))
  (aux x nil) )
```

The above is not copy. Actually, it's reverse!

Now let's define copy using reverse:

```
(reverse (reverse x))
```