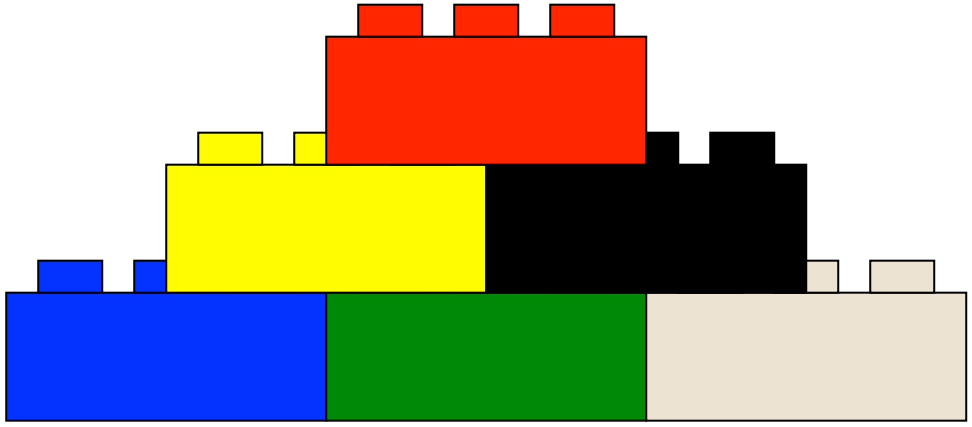
CS 4
Fundamentals of Computer Programming

Lecture 7: January 21, 2015

Compound data types



Last time

- Functions as return values of functions
- Lambda shielding
- New Scheme features: `let`, `display`, `begin`

Today

- Introduce compound data structures in Scheme
  - `cons`
  - `car`
  - `cdr`
- Introduction to *data abstraction*

Previously

- Functions
- Evaluation model
- Recursion
- First-class functions
- Types:
  - numbers, booleans, functions
- A **sufficient** set to compute anything
  - but is it convenient?

Compound data

- We start by motivating:
  - Composite data
  - Data abstraction
- Introduce **`cons`**, **`car`**, and **`cdr`** functions
- Data abstraction examples
  - Complex numbers
  - Complex quadratic equation

Complex Numbers

- Consider:
  - x = a + b*i*

$$i = \sqrt{-1}$$

- How do we represent and use them?
- For motivation, let's just use what we know…

Complex Numbers

- Need both numbers (a=real) and (b=imaginary) for each complex number
- Define some operations…
  - add
  - subtract
  - negate
  - multiply
  - magnitude

Complex Addition

- (ar+ai×*i*) + (br+bi×*i*)
- = (ar+br) + (ai+bi)×*i*
- Add real components to get real component of sum
- Add imaginary components to get imaginary component of sum

Defining `complex-add`

```
(define (complex-add ar ai br bi) ...)
```

- What does it return?
  - Can only return a single number so far
  - But we need two results:
    - real part of result
    - imaginary part of result
  - So… have to write two separate functions?!

complex-add

- Break into two functions:

1) Return real part

```
(define (complex-add-real ar ai br bi)
  (+ ar br))
```

2) Return imaginary part

```
(define (complex-add-imag ar ai br bi)
  (+ ai bi))
```

Complex Multiplication

- $(ar+ai\times i) \times (br+bi\times i)$
- $= (ar\times br + ai\times bi\times i^2) + (ar\times bi\times i + ai\times br\times i)$
- $= (ar\times br - ai\times bi) + (ar\times bi + ai\times br)\times i$
- Real part of result: $(ar\times br - ai\times bi)$
- Imaginary part of result: $(ar\times bi + ai\times br)$

Defining `complex-multiply`

- Real part of result: (ar×br - ai×bi)

```
(define (complex-multiply-real ar ai br bi)
  (- (* ar br) (* ai bi)))
```

- Imaginary part of result: (ar×bi + ai×br)

```
(define (complex-multiply-imag ar ai br bi)
  (+ (* ar bi) (* ai br)))
```

Consider

- foo(A,B,C) = A+B*C (where A, B, C complex)

```
(define (foo-real  ar ai  br bi  cr ci)
   (complex-add-real  ar ai
     (complex-multiply-real  br bi  cr ci)
     (complex-multiply-imag  br bi  cr ci)))
```

```
(define (foo-imag  ar ai  br bi  cr ci)
   (complex-add-imag  ar ai
     (complex-multiply-real  br bi  cr ci)
     (complex-multiply-imag  br bi  cr ci)))
```

Yuck!

- Have to handle every component separately
  - Makes it hard to understand
  - Exposes lots of underlying complexity
- Have to *return* each component separately
- Have to write separate operations

  (perform lots of operations redundantly)

Redundant operations

- foo(A,B,C) = A+B*C

```
(define (foo-real ar ai br bi cr ci)
  (complex-add-real ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
```

```
(define (foo-imag ar ai br bi cr ci)
  (complex-add-imag ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
```

Redundant operations

- foo(A,B,C) = A+B*C

```
(define (foo-real ar ai br bi cr ci)
  (complex-add-real ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
```

```
(define (foo-imag ar ai br bi cr ci)
  (complex-add-imag ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
```

We need…

- …some way to bundle numbers together
  - then we can treat them as a single item
- Ideally should be able to write:

```
(define (foo a b c)
   (complex-add a (complex-multiply b c)))
```

- In other words:
- We need a way to combine things

Scheme vs. Python etc.

- In Python, we can combine things using...
  - lists
  - tuples
  - dictionaries
  - objects
- Scheme has these too, but we'll follow the book's approach...
- ...what is the *minimum* number of operations we need to combine things?

Compound Data

- Attaching procedure: **`cons`**
  - takes two data items
  - **`cons`**tructs a composite data item
    - …that **contains** the original two
- *e.g.*

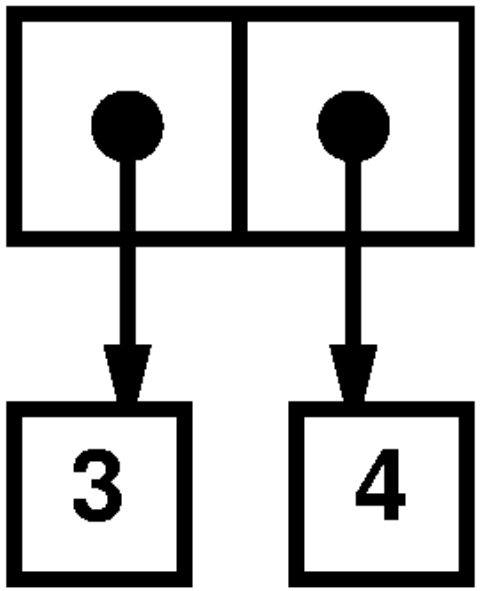
```
(cons 3 4)
;; result: '(3 . 4)
```

- Result of **`cons`** operation called a "cons cell" or "cons pair"

Box and Pointer Diagrams

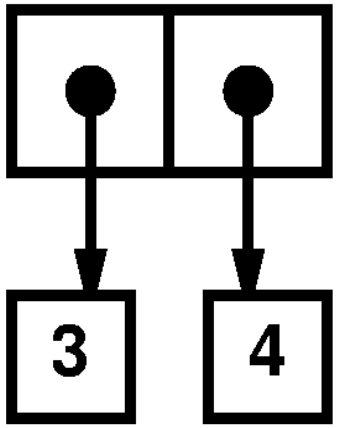
Numbers


```
(cons 3 4)
```



Three ways to describe a `cons` pair

1. With code…
   - `(cons 3 4)`
2. With the way Scheme displays it…
   - `'(3 . 4)  ;; "dotted pair"`
3. With diagrams…



Decomposing

- Can retrieve components of a cons pair with two operations:
  - **`car`**
    - Returns the first item
  - **`cdr`**
    - Returns the second item

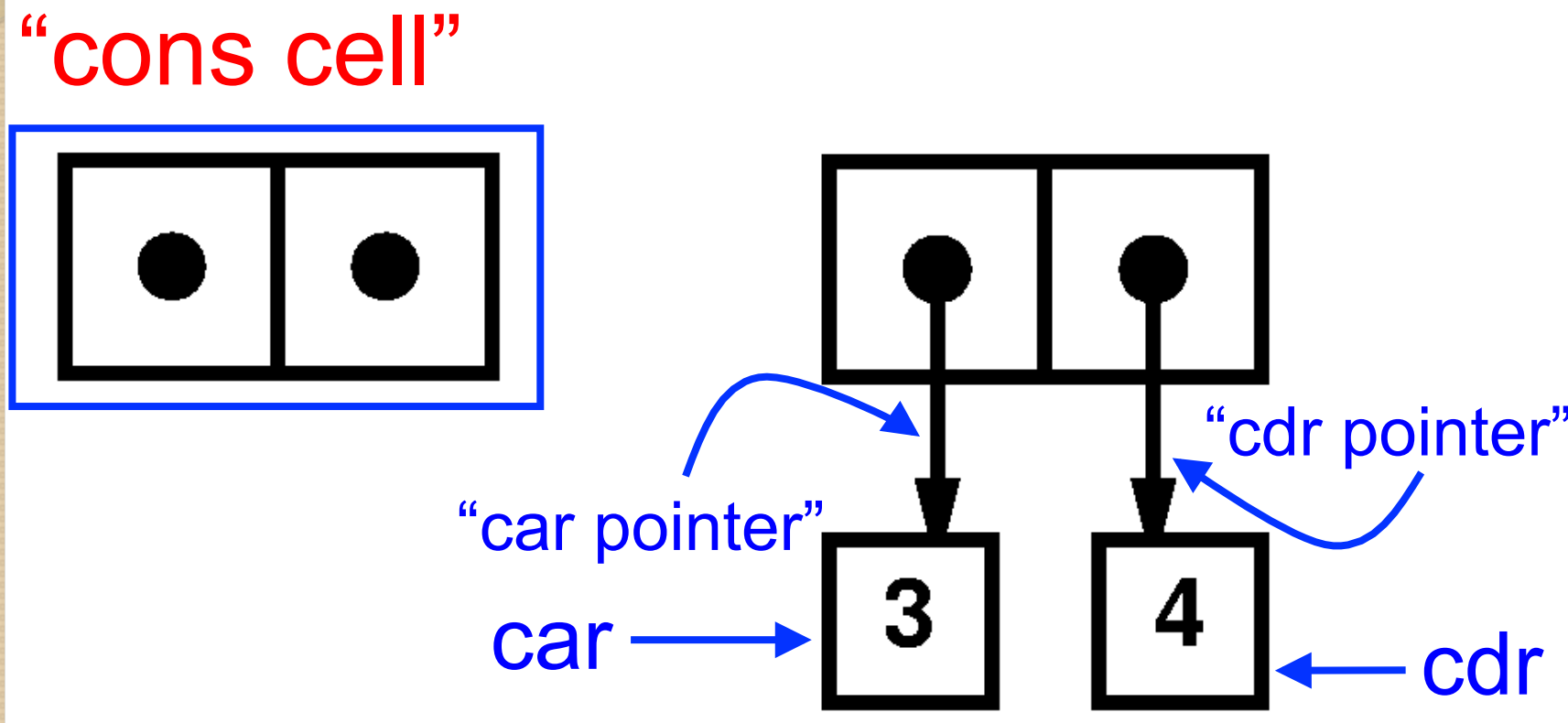
car? cdr?

- The names **`car`** and **`cdr`** are historical accidents
- Original Lisp implementation:

  `car` = "**C**ontents of **A**ddress **R**egister"

  `cdr` = "**C**ontents of **D**ecrement **R**egister"
- Better names: "**`first`**" and "**`second`**"
  - or are they? (Come back to this later)

Box and Pointer Diagrams

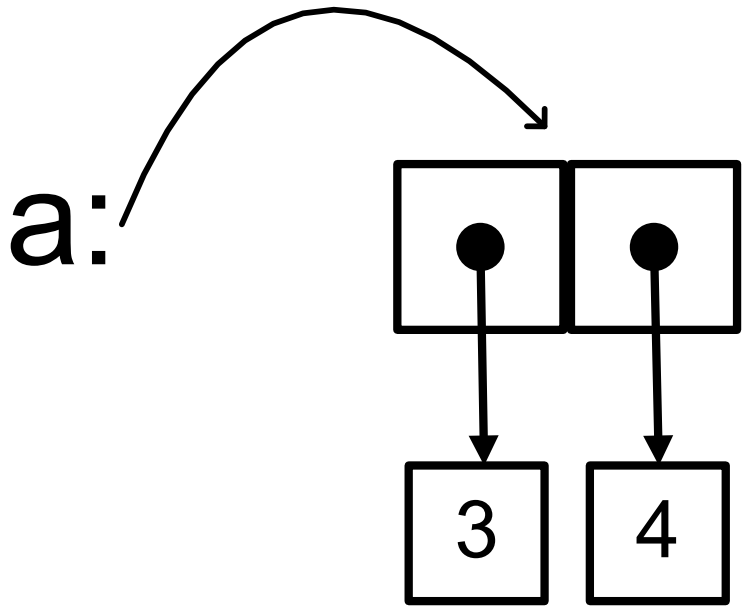


Scheme Example

- `(define a (cons 3 4))`
- `(car a)`
- → `3`
- `(cdr a)`
- → `4`

Box and Pointer Diagrams

```
(define a (cons 3 4))
```



Operational Property of `cons`

- How **`cons`** is implemented doesn't actually matter
  - (you'll see one (weird) way to do it in lab 3)
- What <u>does</u> matter:
  1. **`(car (cons a b))`** ≡ **`a`**
  2. **`(cdr (cons a b))`** ≡ **`b`**

Note

- **`cons`** works on all types:

```
(cons 3 4)
(cons #t #f)
(cons (lambda (x) x)
      (lambda (x) (/ 1 x)))
(cons (cons 3 4) (cons 4 5))
```

- etc. (all are OK)

`cons` arguments

- `cons` does not require arguments to be of the same type
- These are all valid:

```
(cons 3 #t)
(cons 3 (lambda (x) (+ 3 x)))
(cons 3 (cons 4 (lambda (x) x)))
```

- etc.
- (We'll take advantage of this when we introduce lists)

Other things about `cons`/`car`/`cdr`

- **`cons`** must take *exactly* two arguments

```
(cons 1 2 3)  ;; error!
(cons 1)      ;; error!
(cons 1 2)    ;; OK, makes cons pair
```

- **`cons`**, **`car`**, **`cdr`** are ordinary procedures
  - *not* special forms

Question:

- How can **`cons`**/**`car`**/**`cdr`** solve the previous problem (complex numbers)?

Complex numbers, take 2

- Define an *abstraction* of a complex number:

```
(define (make-complex real imaginary)
  (cons real imaginary))
(define (get-real complex) (car complex))
(define (get-imag complex) (cdr complex))
```

Constructors and accessors

- **`make-complex`**: "*constructor* "
  - constructs new complex numbers
- **`get-real`**, **`get-imag`**: "*accessors* "
  - provides access to contents of complex numbers
  - in this case: real, imaginary parts
- Question: Why not just use **`cons`**/**`car`**/**`cdr`** directly? (more about this later)

Complex numbers: operations

Define our operations in terms of this abstraction:

-
```
(define (complex-add a b)
  (make-complex
    (+ (get-real a) (get-real b))
    (+ (get-imag a) (get-imag b))))
```

Sample Evaluation

- `(define a (make-complex 3 4))`
  → `a` is now `'(3 . 4)`
- `(define b (make-complex 1 2))`
  → `b` is now `'(1 . 2)`
- `(complex-add a b)`
  →
```
(make-complex
    (+ (get-real (3 . 4)) (get-real (1 . 2)))
    (+ (get-imag (3 . 4)) (get-imag (1 . 2))))
```

Continuing Evaluation

(not strict evaluation order here)

```
(make-complex
  (+ (get-real '(3 . 4)) (get-real '(1 . 2)))
  (+ (get-imag '(3 . 4)) (get-imag '(1 . 2))))

(make-complex (+ (car '(3 . 4)) (car '(1 . 2)))
              (+ (cdr '(3 . 4)) (cdr '(1 . 2))))

(make-complex (+ 3 1) (+ 4 2))

(cons 4 6)
```

→ `'(4 . 6)`

Note:

- The single quote in:

  `'(4 . 6)`

- is short for the Scheme **`quote`** special form
  - which we will discuss at length in a few lectures
- The dot syntax is a notational convenience
  - can just write `(cons 4 6)` instead

Complex numbers: multiply

Now define operations in terms of abstraction:

```
(define (complex-multiply a b)
    (make-complex
      <real part of result>
      <imaginary part of result>))
```

Complex numbers: multiply

Now define operations in terms of abstraction:

```
(define (complex-multiply a b)
  (make-complex
    (- (* (get-real a) (get-real b))
       (* (get-imag a) (get-imag b)))
    (+ (* (get-real a) (get-imag b))
       (* (get-imag a) (get-real b)))))
```

Complex numbers: magnitude

- "Magnitude" of a complex number is defined to be

$$magnitude(x + y \times i) = \sqrt{x^2 + y^2}$$

- *N.B.* magnitude is a real number

Complex numbers: magnitude

$$magnitude(x + y \times i) = \sqrt{x^2 + y^2}$$

```
(define (complex-magnitude a)
  (sqrt (+ (square (get-real a))
           (square (get-imag a)))))
```

Revisiting `foo`

- foo(A,B,C) = A+B*C
- ```
  (define (foo a b c)
     (complex-add
        a
        (complex-multiply b c)))
  ```
- Simple, direct reflection of computation we had in mind


  ```

Claim:

- Compound data packing makes the code easier to understand.

Comparison

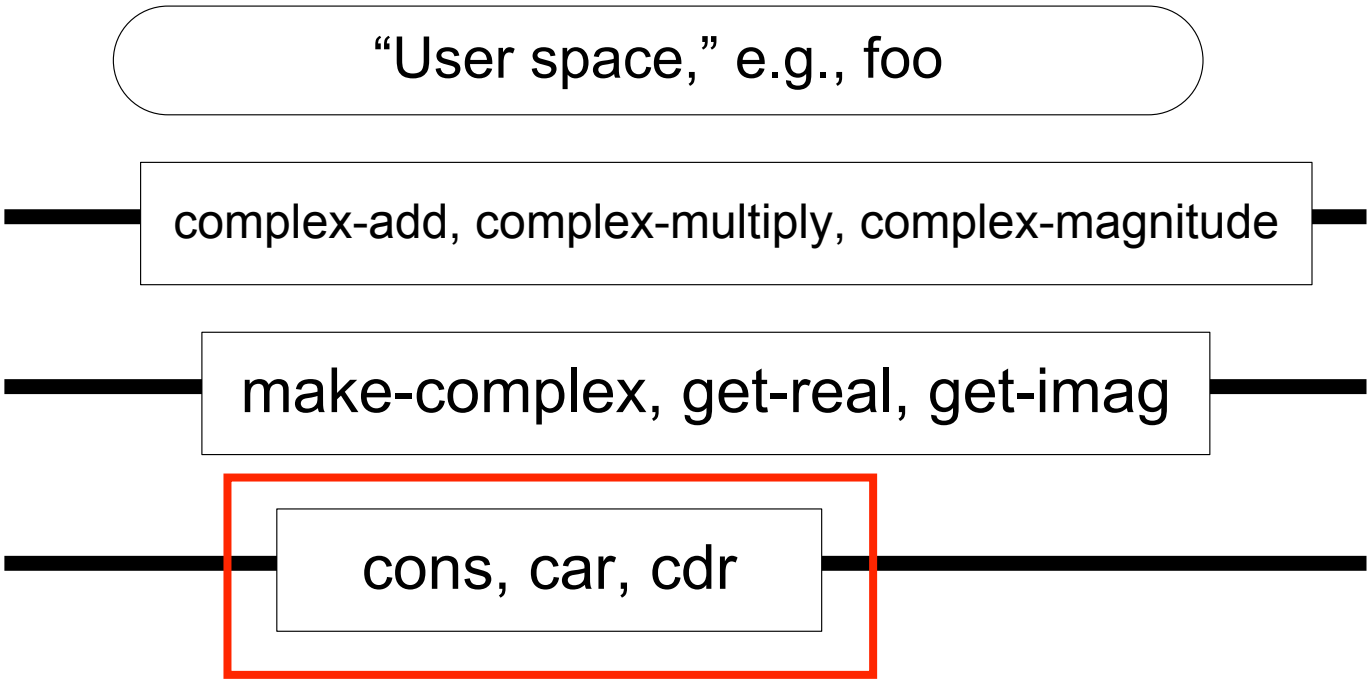
```
(define (foo-real ar ai br bi cr ci)
  (complex-add-real ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
(define (foo-imag ar ai br bi cr ci)
  (complex-add-imag ar ai
    (complex-multiply-real br bi cr ci)
    (complex-multiply-imag br bi cr ci)))
```

VS.

```
(define (foo a b c)
  (complex-add
    a
    (complex-multiply b c)))
```

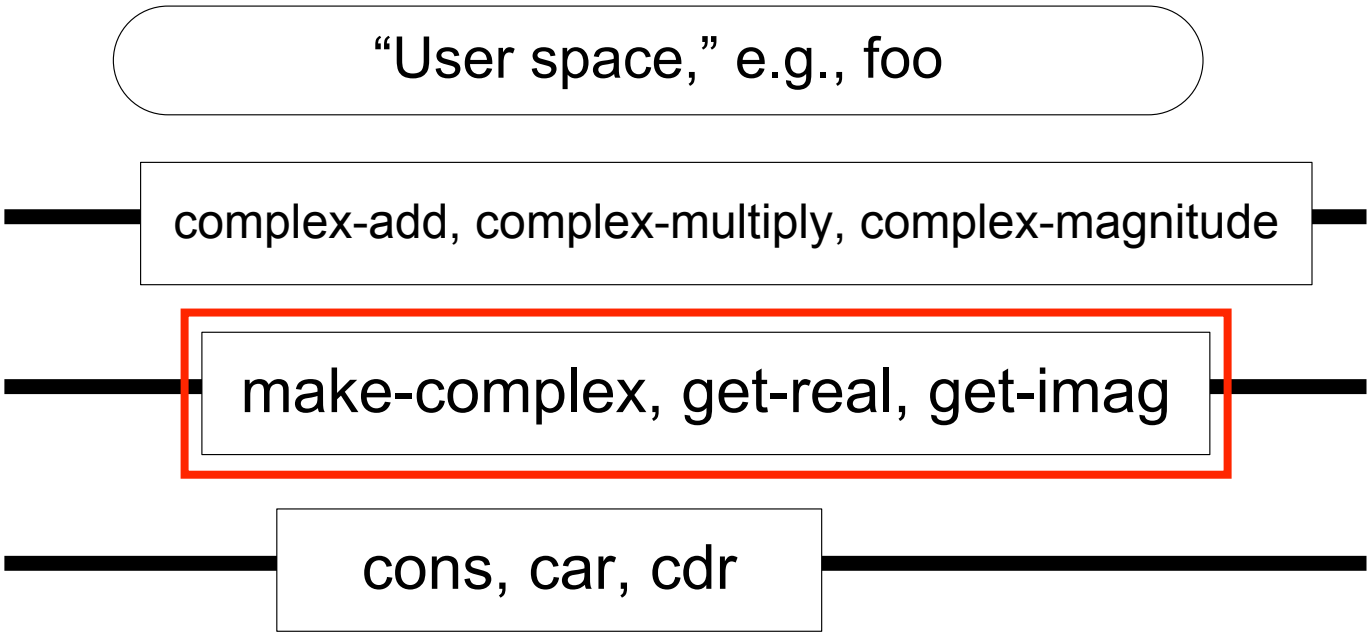
Notice

- In developing the solution:
  - Maintained a strict *abstraction hierarchy*:



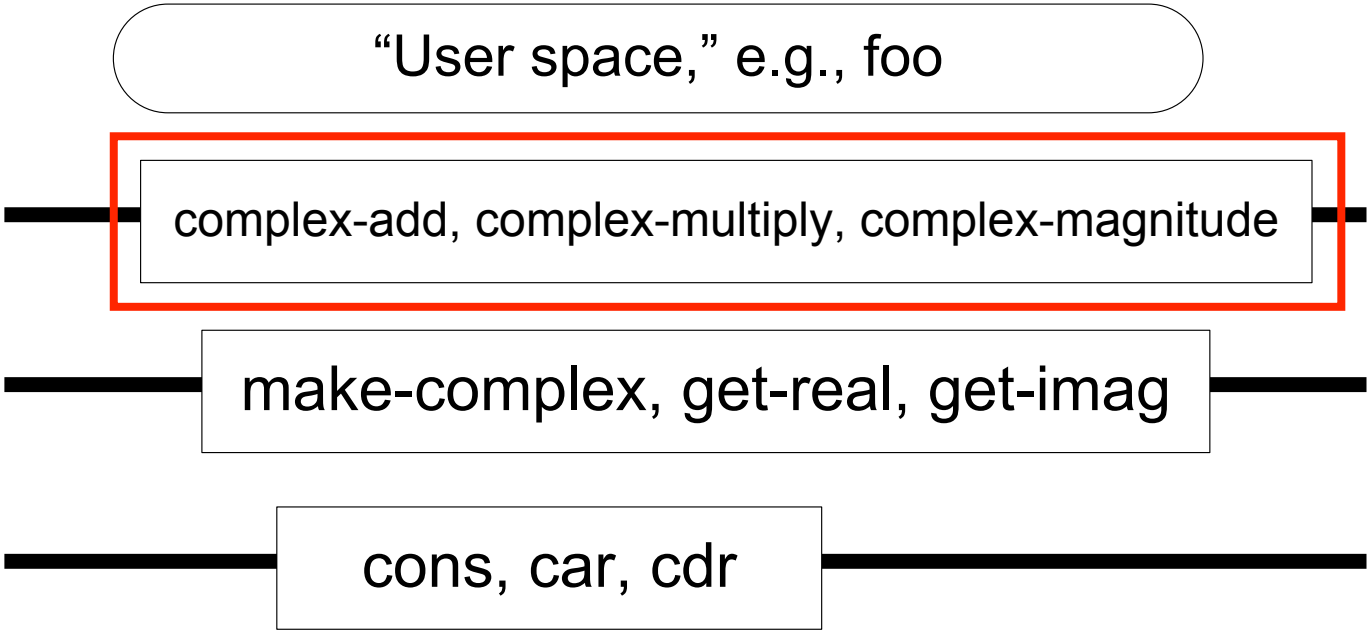
Notice

- In developing the solution:
  - Maintained a strict *abstraction hierarchy*:



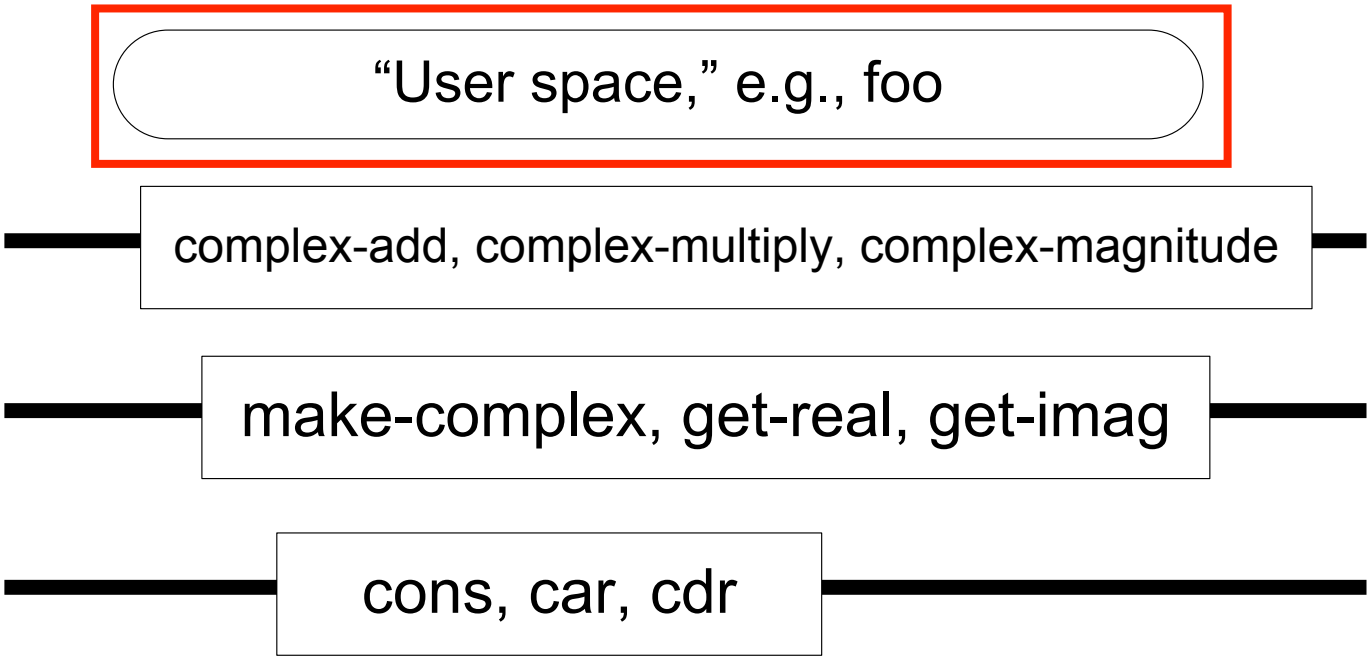
Notice

- In developing the solution:
  - Maintained a strict *abstraction hierarchy*:



Notice

- In developing the solution:
  - Maintained a strict *abstraction hierarchy*:



"Strict" abstraction hierarchy

- Means: each level only depends on level *immediately* below it
- Very convenient -- less to think about when writing new procedures
- Not always possible to have perfectly strict hierarchy, but desirable
- *Don't* want to depend on all levels below!

The point

- Once we've defined **`make-complex`**, **`get-real`**, **`get-imag`**
  - …in terms of **`cons`**, **`car`**, **`and cdr`**
  - we don't use **`cons`**, **`car`**, or **`cdr`** again!
  - All other operations on complex numbers are always in terms of **`make-complex`**, **`get-real`**, and **`get-imag`**
  - Those operations don't have to know how complex numbers are implemented

  → We can change the underlying implementation *without having to rewrite any other code*

Hiding implementation details

- Once we've defined **`complex-add`**, **`complex-multiply`**…
  - …we can define other operations (*e.g.* **`foo`**) in terms of those operations
  - We don't need to know anything about how to look inside complex numbers
  - We have successfully hidden the details of how to do complex operations
  - *e.g.* **`foo`** doesn't know or care that complex numbers implemented in terms of **`car`**/**`cdr`**

Building on complex number abstraction

- Now consider defining a quadratic equation in terms of complex numbers:
- $Q = A*x^2 + B*x + C$
  - where A, B, C, and x are all complex.
- How do we represent this equation?

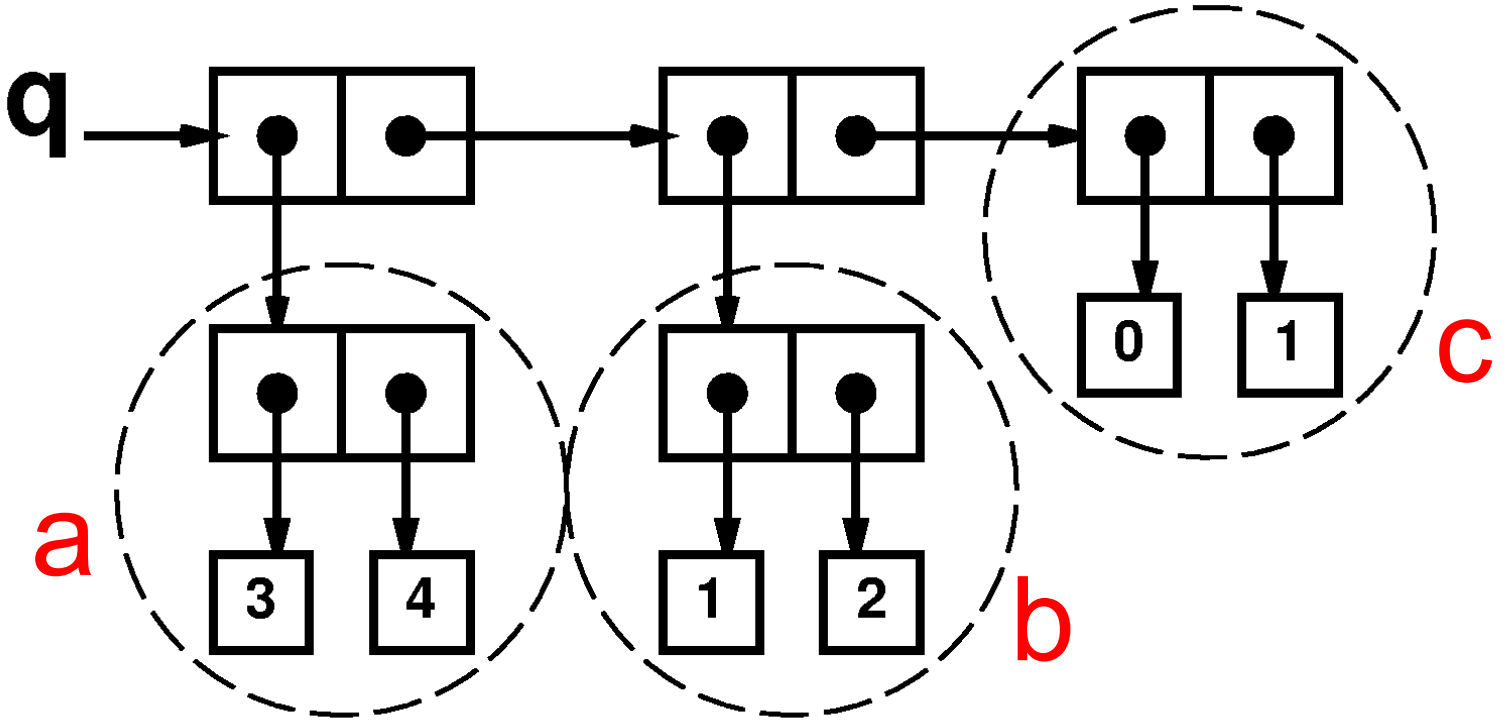
Represent complex quadratic

```
;; constructor:
(define (make-quadratic a b c)
  (cons a (cons b c)))
;; accessors:
(define (get-a quad) (car quad))
(define (get-b quad) (car (cdr quad)))
(define (get-c quad) (cdr (cdr quad)))
```

- After this, don't use **`cons`**/**`car`**/**`cdr`** any more in this problem!

Complex Quadratic Picture

```
(define q (make-quadratic
            (make-complex 3 4)
            (make-complex 1 2)
            (make-complex 0 1))
```



Evaluate Quadratic

- Evaluate a quadratic at a point:

```
(define (evaluate-quadratic quad point)
  (complex-add
    (complex-add
      (complex-multiply
        (get-a quad)
        (complex-multiply point point))
      (complex-multiply (get-b quad) point))
    (get-c quad)))
```

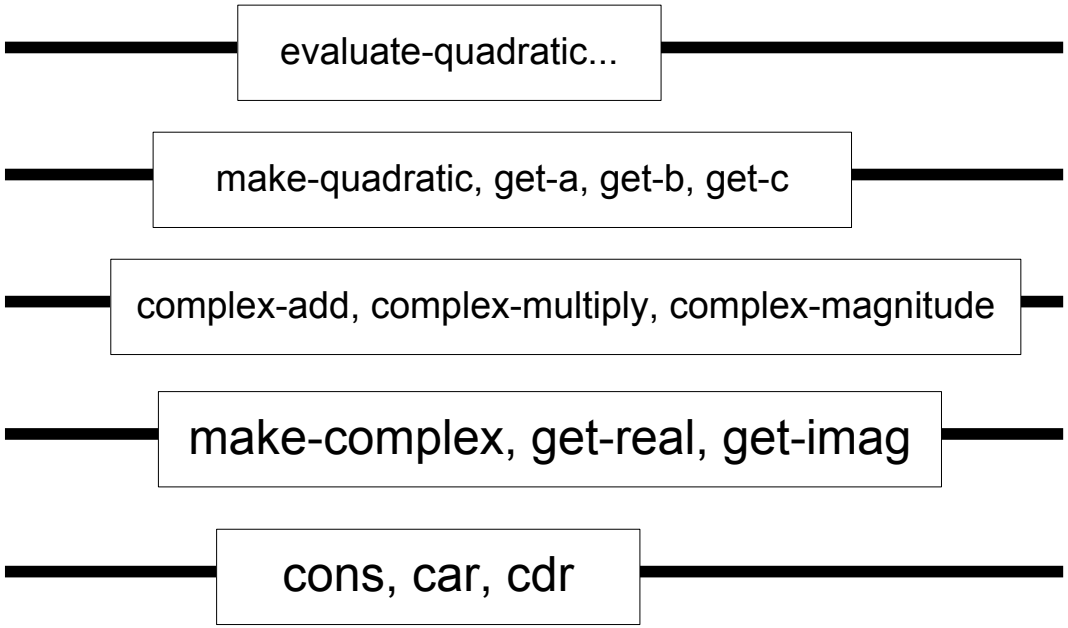
$A*X^2$

$B*X$

C

Notice

- Our **`evaluate-quadratic`** function does *not* need to know how…
  - …complex numbers are implemented
  - …operations on complex numbers work



Dependencies

- **`evaluate-quadratic`** depends on
  - **`get-a`**, **`get-b`**, **`get-c`** (1 level below)
  - **`complex-add`**, **`complex-multiply`** (2 levels below)
- So abstraction hierarchy is not *completely* strict
  - but certainly independent of **`car`**, **`cdr`**, **`cons`**

Change of representation

- One advantage of abstraction layers is that you can change lower-level representations without altering higher-level code
- Example: use Scheme vectors instead of cons cells
- Vector: a collection of items, indexed by location
  - like an array, or a Python list
  - *not* like a Scheme list (next class)

Vectors

```
> (define v (vector 1 2 3 4 5))
> (vector-ref v 0)
1
> (vector-ref v 1)
2
...
```

New representation

```
;; constructor:
(define (make-quadratic a b c)
  (vector a b c))
;; accessors:
(define (get-a quad)
  (vector-ref quad 0))
(define (get-b quad)
  (vector-ref quad 1))
(define (get-c quad)
  (vector-ref quad 2))
```

New representation

- Rest of code unchanged!
- Can change low-level representation (*e.g.* for efficiency) without worrying about other layers
- Works because strict abstraction hierarchy was set up

Abstraction Barriers

- Critical to controlling complexity, because:
  - Large objects…
    - equations...
    - airplanes...
  - …are implemented in terms of potentially many primitive data items

Abstraction Barriers

- *Without* abstraction:
  - the number of operations,
  - the number of ways to represent things, and
  - the number of ways to combine them
- …can grow **exponentially** w.r.t. the number of primitive data items
  - "Everything depends on everything else"
- Without abstraction, a change in any component requires everything to change

Abstraction Barriers

- A strict hierarchy helps **linearize** the problem:
  - Only need to know about a fixed (constant) number of things at each level
  - Can think about a single level at a time without thinking about (or even knowing about) what lies below
  - *e.g.*, can reason about quadratic equation in terms of `complex-multiply` w/out knowing how `complex-multiply` works or how complex numbers are implemented

Abstraction Barriers

- Abstraction barriers make code *manageable* and *comprehensible*
- Without them, code is a *"big ball of mud"* (BBOM) and every part potentially depends on every other part
- Understanding a BBOM is almost impossible, especially as system scales up
- Understanding a system with abstraction barriers is easy, because at any given point, only need to think about a small number of things!

Abstraction

- Decomposes a larger conceptual problem…
  - …into a series of *small* conceptual problems
- Also known as "divide-and-conquer"
  - Conquer conceptual complexity…
  - …by dividing into smaller, *independent* pieces

Different kinds of abstraction

- In previous lectures we've seen *procedural* abstraction
  - writing procedures that were abstracted versions of more specific procedures
- Today we see *data* abstraction
  - writing constructors and accessors for new kinds of data (*e.g.* complex numbers, quadratic equations) which are abstracted versions of more specific kinds of data (cons pairs, groups of cons pairs etc.)

Summary

- Compound data operators allow us to package up many data items into one
- They allow us to represent things with multiple components as single, abstract object
- Rigorous data abstraction allows us to contain, constrain, and deal with the complexity of compound objects and the layers of functionality we may pile on top

Next time

- Extend `cons`/`car`/`cdr` to create lists
- More data abstraction