

# Why are we here?

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 1.1

# But first: What's a Bootcamp?



# But really: what's bootcamp about?

- Remolding the recruit (that's you)
- Learning discipline, self-control
- Learning to master challenging tasks
- Personal growth



# What does this mean for us?

- Goal: learn to write *well-designed* programs
- Discipline: don't just spew out code
- We will give you a recipe (a discipline!) for creating well-designed programs.
  - Learn what questions to ask
  - Learn how to answer them

# Programming is a people discipline

- It's not just about writing code. It requires people skills, too.
- You will learn:
  - How to document your choices
  - How to explain your programs to others
  - How to defend your choices

# Our learning principles

*You are responsible for your own learning*

- Immersion
- Active reading
- Mastering small skills first
- Practice makes perfect

# Immersion

- You will work very hard
  - median is about 20 hours/week.
- By working so hard, you will develop good programming habits, and unlearn many of the bad habits that you have developed.
- You will dream about this course!

# Active Reading

- Reading is not a passive activity
- Read every word
- Read with pencil in hand
- Write down questions as you read
  - If you don't have any questions, you haven't read carefully enough!
- Look things up
  - We will not be spoon-feeding you
- Do the exercises



# Mastering small skills first

- Like writing:
  - First sentences
  - Then paragraphs
  - Then compositions
- We will start with very small programs
  - But you have to get them right first.
  - And learn to get them right every time

# Practice Makes Perfect

- Mastery comes through repetition!



# Summary

- Goal: learn to write *well-designed* programs
- Goal: learn to explain and defend your designs
- Our learning principles:
  - Immersion
  - Active Reading
  - Mastering Small Skills First
  - Practice Makes Perfect

# The Design Recipe

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 1.2

## The Six Principles

1. Programming is a People Discipline
2. Represent Information as Data; Interpret Data as Information
3. Programs should consist of functions and methods that consume and produce data
4. Design Functions Systematically
5. Design Systems Iteratively
6. Use State Only for Sharing Values

*Write this down, in your own handwriting!*

## The Four Slogans

1. Stick to the recipe!
2. Practice makes perfect.
3. The Shape of the Data Determines the Shape of the Program.
4. Test suites are your friend.

*Write this down, too!*

# The Design Recipe

## The Design Recipe

1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples
4. Design Strategy
5. Function Definition
6. Tests

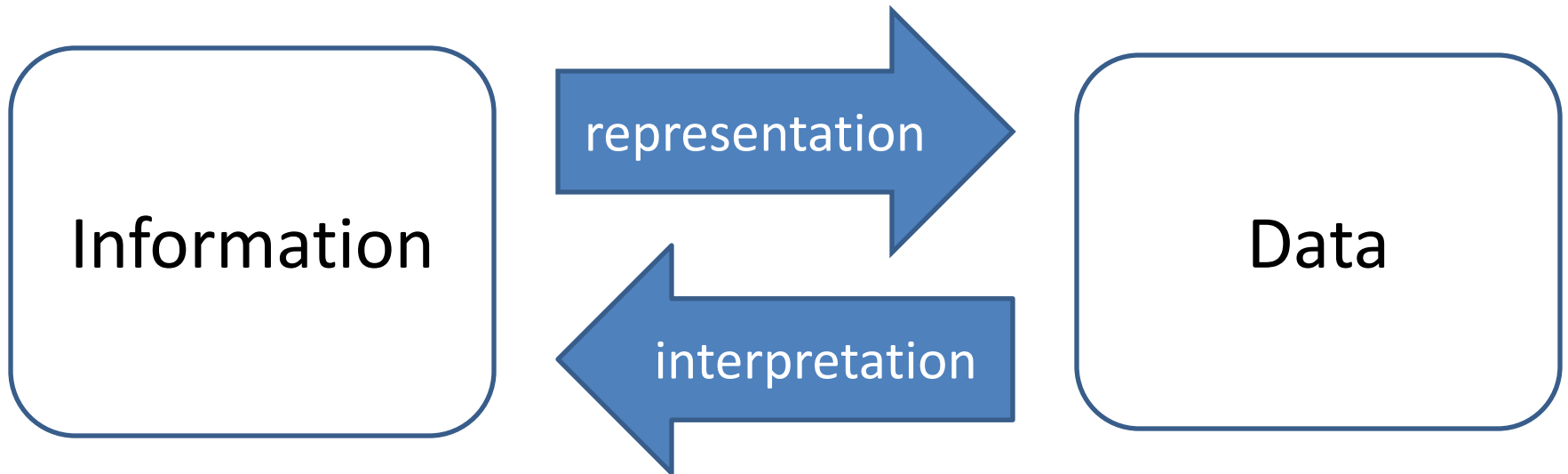
*This is important. Write it down, in your own handwriting. Keep it with you at all times. Put it on your mirror. Put it under your pillow. I'm not kidding!*

# Information Analysis and Data Design

- Information is what lives in the real world
- Need to decide *what part* of that information needs to be represented as data.
- Need to decide *how* that information will be represented as data
- Need to document how to *interpret* the data as information



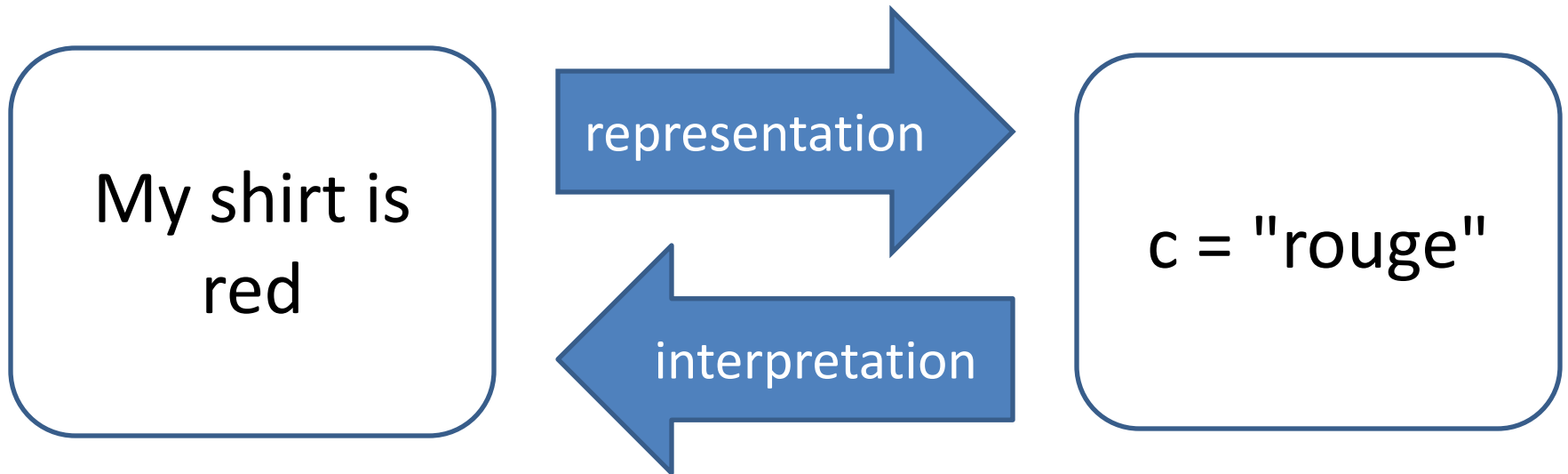
# Information and Data



This is not a pipe



# Information and Data: Example



**Interpretation:**

c = the color of my shirt, as a string, in French

# Information Analysis and Data Design

- Deliverables:
  - definition of data structures (“data definition”)
    - Including interpretation (vital!)
  - templates
    - for each kind of data, an outline of how to write a function that manipulates that data
    - will see examples of this later.

# Contract and Purpose Statement

- Contract: specifies the kind of input data and the kind of output data
- Purpose: a short declarative statement of how the output depends on the input
  - “Given XXX, returns<sup>s</sup> (or produces<sup>s</sup>) YYY”
  - “Returns<sup>s</sup> the YYY of the given XXX”
  - When possible, phrase in terms of information, not data

# Contract and Purpose Statement

**;; square : Number -> Number**

**;; Given a number, returns its square.**

**;; even? : Number -> Boolean**

**;; Given a number, returns true iff it is even.**

Not much difference between information and data here

# Contract and Purpose Statement

`;; f2c: Number -> Number`

`;; Given a temperature in Fahrenheit, returns  
the equivalent in Celsius.`

# Contract and Purpose Statement

;; f2c: **Number** -> Number

;; Given a **temperature in Fahrenheit**, returns  
the equivalent in Celsius.

data



information



# Examples

- Show sample arguments and results, to make clear what is intended.
- **$(f2c\ 32) = 0$**
- **$(f2c\ 212) = 100$**

# Design Strategy

- A short description of how to get from the purpose statement to the function definition
- We will have a menu of strategies. Here they are:

The Four Strategies
Domain Knowledge
Function Composition
Structural Decomposition
General Recursion

- We'll talk about the first three today; general recursion will come later.

# Design Strategy

- For f2c, the strategy we use is “domain knowledge”
  - meaning that there’s no real programming knowledge used

# Function Definition

- We apply some domain knowledge:

$$f2c(x) = ax + b$$

$$f2c(32) = 0$$

$$f2c(212) = 100$$

$$a \times 32 + b = 0$$

$$a \times 212 + b = 100$$

apply a little linear algebra

$$a = 5/9$$

$$b = -160/9$$

# Function Definition

- And we write the code.
  - We transcribe the formula
  - Racket has rational numbers

```
(define (f2c x)  
  (+ (* 5/9 x) -160/9))
```

# Tests

```
(check-equal? (f2c 32) 0  
  "32 Fahrenheit should be 0 Celsius")  
(check-equal? (f2c 212) 100  
  "212 Fahrenheit should be 100 Celsius")
```

Usually each test will have an associated text that describes what we are testing. More on this later...

# Summ

## The Design Recipe

1. Information Analysis and Data Design
2. Contract and Purpose Statement

## The Design Recipe

1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples

4. Design Strategy

5. Function Definition

6. Tests

## The Design

1. Information Analy

2. Contract and Purp

3. Examples

4. Design Strategy

5. Function Definition

6. Tests

## The Design Recipe

1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples

4. Design Strategy

5. Function Definition

6. Tests

– The single

is course!!

## The Design Re

1. Information Analysis ar

2. Contract and Purpose S

3. Examples

4. Design Strategy

5. Function Definition

6. Tests

4. Design Strategy

5. Function Definition

6. Tests

## Design Recipe

1. Information Analysis and Data Design

2. Contract and Purpose Statement

3. Examples

4. Design Strategy

5. Function Definition

6. Tests

# Information and Data

CS 5010 Program Design Paradigms

“Bootcamp”

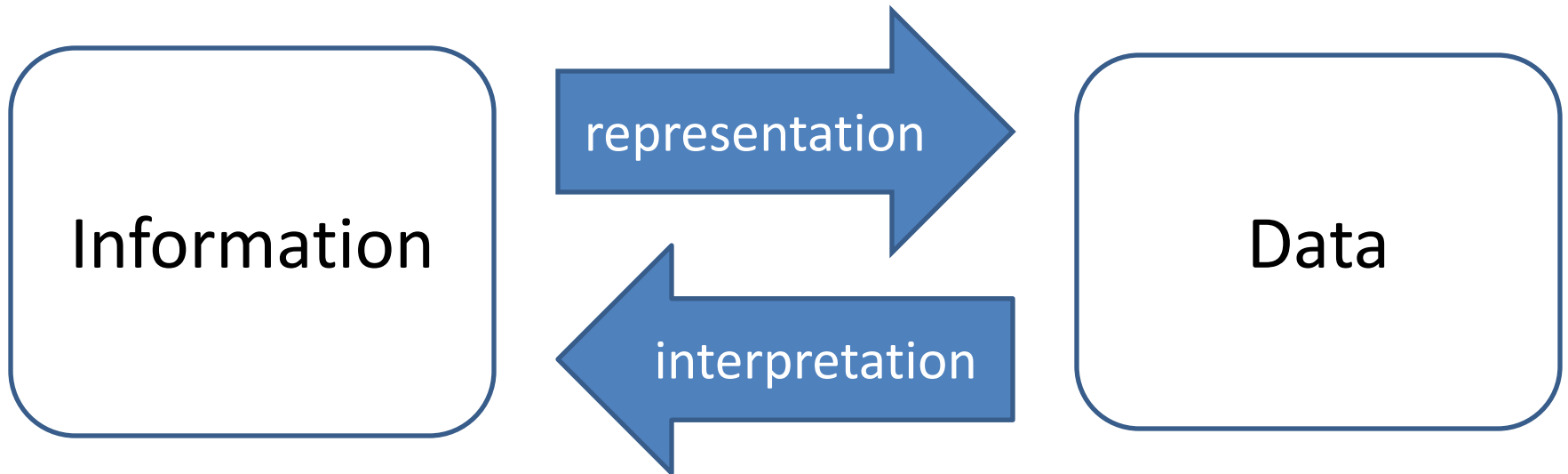
Lesson 1.3



# Information Analysis and Data Design

- Information is what lives in the real world
- Need to decide *what part* of that information needs to be represented as data.
- Need to decide *how* that information will be represented as data
- Need to document how to *interpret* the data as information

# Information and Data



# Information Analysis and Data Design

- Deliverables:
  - definition of data structures (“data definition”)
  - interpretation
  - templates
    - for each kind of data, an outline of how to write a function that manipulates that data
    - will see examples of this later.

# Information Analysis

- Of all the information in the world, which parts do we need to represent?
- Choice depends on the application

# Information about a classroom

- Location
- Capacity
- Schedule
- Arrangement of blackboards
- Who's in it now?
- What color are their clothes?
- Current temperature
- Number of Windows
- Color of seats
- Status of projector
- Status of internet connection
- Number of pieces of chalk in the tray...

# Classroom Scheduling Application

- Location
- Capacity
- Schedule
- Arrangement of blackboards
- Who's in it now?
- What color are their clothes?
- Current temperature
- Number of Windows
- Color of seats
- Status of projector
- Status of internet connection
- Number of pieces of chalk in the tray...

# HVAC Application



(Heating, Ventilation,  
and Air-Conditioning)

- Location
- Capacity
- Schedule
- Arrangement of blackboards
- Who's in it now?
- What color are their clothes?
- Current temperature
- Number of Windows
- Color of seats
- Status of projector
- Status of internet connection
- Number of pieces of chalk in the tray...

# Building Maintenance Application

- Location
- Capacity
- Schedule
- Arrangement of blackboards
- Who's in it now?
- What color are their clothes?
- Current temperature
- Number of Windows
- Color of seats
- Status of projector
- Status of internet connection
- Number of pieces of chalk in the tray...



# Kinds of Information

- Scalar information
- Partition Information
- Itemization Information
- Compound Information
- Mixed Information

# Scalar Information

- Information that is simple numbers or strings
- Examples:
  - Lengths, temperatures, distances, prices
- Racket has many other kinds of scalar information
  - Examples: Images, functions

# Partition Information

- information that is scalar information, but has different interpretations for different values
- Examples:
  - Taxable income
  - Keyboard inputs (sometimes)

# Itemization Information

- information that has only a few different values or a few different kinds of values.
- Examples:
  - A traffic light may be red, yellow, or green.
  - A coffee type may be weak, strong, or decaf.

# Compound Information

- information that has multiple parts
- Example: An inventory record in a bookstore may include
  - the author,
  - the title,
  - the number of copies on hand,
  - the cost of the book from the publisher,
  - the price that the bookstore charges for the book.

# Mixed Information

- Itemization information where one or more of the cases is compound information.
- Example: a wine bar order might be one of
  - Cup of coffee (compound information: size and type)
  - Glass of wine (compound information: vineyard and vintage)
  - Cup of Tea (scalar information)

# Kinds of Information: Summary

Kind of Information	Example
Scalar	Temperature
Partition	Taxable Income
Itemization	Traffic Light state (red, yellow, <b>OR</b> green)
Compound	Book (author, title, copies, cost, <b>AND</b> price)
Mixed	BarOrder (coffee ( <b>compound</b> ), <b>OR</b> wine ( <b>compound</b> ) <b>OR</b> tea (scalar))

# Representing Information as Data

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 1.4



# Data Definition

- For each kind of information we want to manipulate, we must choose a representation.
- The *data definition* documents our choice of representation.
- Always shows how to construct a value of the right kind.
- This serves as a reference as we design the rest of our program.

# Scalar Information

- Simple data, eg numbers, strings, etc.
- These are already values in Racket.
- You can represent anything as a number, but that's not likely to be a good representation
  - 404? -1?
- You can represent anything as a string, but that's not likely to be a good representation, either.

# Itemization Information

- Data that is one of a few values, or a few kinds of values.
- The data definition lists the possible values and their interpretation.

# Data Definition for Itemization Information

A TLState is one of

- "red"
- "yellow"
- "green"



We use CamelCase for names of kinds of data

interpretation is obvious, so not necessary here. We'll get to more complicated examples later.

Itemization Data was called “Enumeration Data” in HtDP 1/e.

# Partition Information

- The data definition must show
  - What kind of data is being partitioned into cases
  - Exactly which values are in each case
  - The interpretation of each case
- The cases must be
  - Mutually exclusive
  - Exhaustive
  - ➔ each value of the data falls into one and only one case.

# Data Definitions for Partition Data: Examples

```
;; example:  tax tables
```

```
;; A TaxableIncome is a positive Number,
```


```
;; which is one of:
```

```
;; [0-10000)
```

```
;; [10000,20000)
```

```
;; [20000,infinity)
```

Every positive number  
falls into exactly one  
of these categories



```
;; An FallingCatKeyEvent is a KeyEvent,
```

```
;; which is one of:
```

```
;; -- " "
```

```
      (interp: pause/unpause)
```

```
;; -- any other KeyEvent (interp: ignore)
```

Every KeyEvent falls into exactly one  
of these categories



# Compound Information

- Data that consists of two or more quantities.
- In Racket, we represent this as a **struct**

# define-struct

```
(define-struct book (author title copies))
```

This just defines a set of functions in Racket. It is not a data definition

This defines the following functions:

**make-book**

A constructor— given 3 arguments, returns a **book** with the given fields

**book-author**

**book-title**

**book-copies**

Selectors: given a **book**, return the indicated field.

**book?**

A Predicate: given any value, returns true iff it is a **book**



# Data Definition

The struct definition



```
(define-struct book (author title copies))
```

A Book is a (make-book String String Number)

Interpretation:

author is the author's name

title is the title

copies is the number of copies on hand

The Data  
Definition



# Data Definition

The name of this kind of data

How to construct this kind of data

The kind of data in each field

```
(define-struct book (author title copies))
```

A Book is a (make-book String String Number)

Interpretation:

author is the author's name

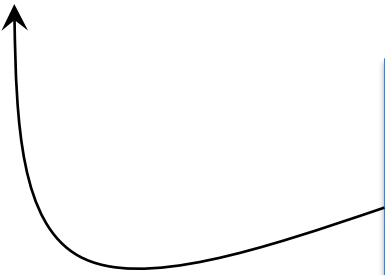
title is the title

copies is the number of copies on hand

The interpretation of each piece of data

# Data Definition

```
(define-struct ring (inner outer))  
;; A Ring is a (make-ring Number Number)  
;; inner is the inner radius of the ring, in  
   cm.  
;; outer is the outer radius of the ring, in  
   cm.  
;; INVARIANT: (< inner outer) is true.
```



Not every pair of Numbers is a valid Ring. Here we document any additional constraints. A function that takes a Ring as an argument can rely on this, but every function that creates a Ring must guarantee that the condition is true.

# Mixed Information

- Represent like itemization data, but with structs for each alternative that's a compound.
- Data definition must show the interpretation of each alternative
- Data definition must show the interpretation of each field

# Data Definition for mixed data: example

```
(define-struct wine-order (type year))
(define-struct coffee-order (type milk?))

;; A BarOrder is one of
;; -- (make-coffee-order CoffeeType Boolean)
;;      Interpretation:
;;      type is the type of coffee
;;      milk? is true iff the coffee should be served with
;;      milk in it
;; -- (make-wine-order String Number)
;;      Interpretation:
;;      type is the vineyard
;;      year is the vintage
;; -- "tea"
```

# Summary

- Information lives in the world, data lives in the computer
- Choice of information to be represented depends on the application
- Represent information as data; interpret data as information
- Kinds of data
  - scalar, partition, enumeration, compound, mixed
- Data definition: must include interpretation!

# Templates

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 1.5

# Templates give us the shape of the program

- For each data definition (except scalar data), we write a template.
- The template for a data definition tells us how to go about writing a function that manipulates values from that data definition.



# Itemization or Partition Data

- Recipe for template for itemization data:

Question	Answer
Does the data definition distinguish among different subclasses of data?	Write a cond expression with a clause for each case.
How do the subclasses differ from each other?	Write predicates that distinguish the cases.

# Example:

Always start with  
contract

```
;; tls-fn : TLState -> ??  
(define (tls-fn a-tls)  
  (cond  
    ← [(string=? a-tls "red")  
       ...]  
    [(string=? a-tls "yellow")  
       ...]  
    [(string=? a-tls "green")  
       ...])))
```

Write it without  
comments, then  
comment it

# Compound Data

- Write a skeleton definition with the selectors.

# Data Definition

```
(define-struct book (author title copies))
```

A Book is a (make-book String String Number)

Interpretation:

author is the author's name

title is the title

copies is the number of copies on hand

# Template for Book

```
;; book-fn : Book -> ??
```

```
(define (book-fn b)
```

```
  (...)
```

```
    (book-author b)
```

```
    (book-title b)
```

```
    (book-copies b)))
```

Take apart the  
pieces

This is the  
"inventory"

And combine  
them functionally

# Recipe for template: compound data

Question	Answer
Does the data definition distinguish among different subclasses of data?	Write a cond expression with a clause for each case.
How do the subclasses differ from each other?	Write predicates that distinguish the cases.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the template.

# Mixed Data

- Like itemization data, but one or more of the alternatives are compound data
- Template: like for itemization data, but add selectors for the compound cases

# Data Definition for mixed data: example

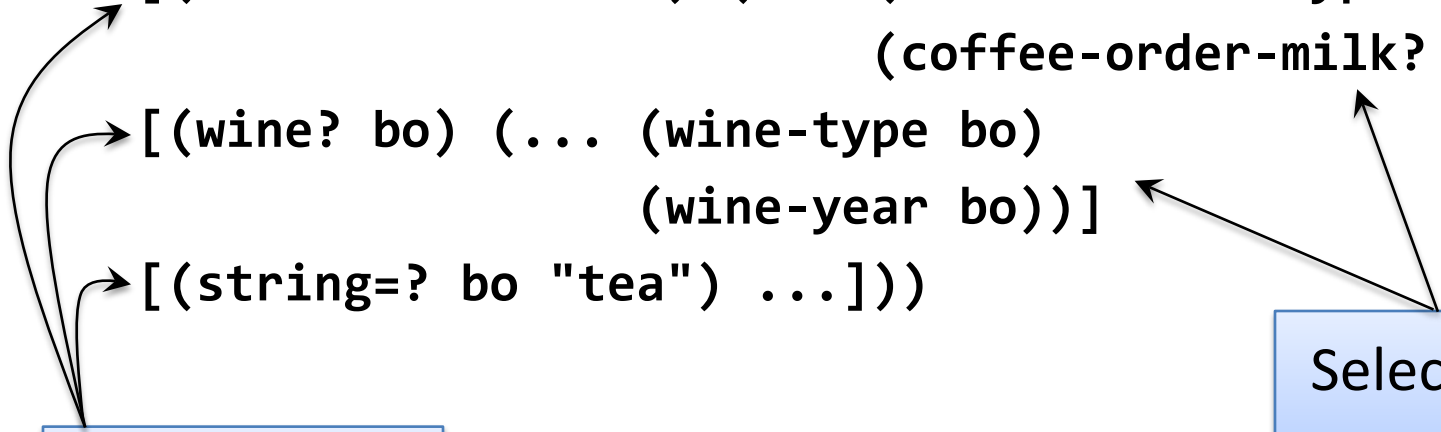
```
(define-struct wine-order (type year))  
(define-struct coffee-order (type milk?))  
  
;; A BarOrder is one of  
;; -- (make-coffee-order CoffeeType Boolean)  
;; Interpretation:  
;; type is the type of coffee  
;; milk? is true iff the coffee should be served with  
;; milk in it  
;; -- (make-wine-order String Number)  
;; Interpretation:  
;; type is the vineyard  
;; year is the vintage  
;; -- "tea"
```

3 alternatives; 2  
are compounds



# Template for mixed data

```
;; bar-order-fn : BarOrder -> ??  
(define (bar-order-fn bo)  
  (cond  
    [(coffee-order? bo) (... (coffee-order-type bo)  
                               (coffee-order-milk? bo))]  
    [(wine? bo) (... (wine-type bo)  
                     (wine-year bo))]  
    [(string=? bo "tea") ...]))
```



3 alternatives

Selectors for the  
compound cases

# The Recipe for Templates (again)

Question	Answer
Does the data definition distinguish among different subclasses of data?	Write a cond expression with a clause for each case.
How do the subclasses differ from each other?	Write predicates that distinguish the cases.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the template.

# Domain Knowledge and Function Composition

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 1.6

# Programs are Functions

- Functions take an argument and return a result.
- Contract says what kind of data the argument and result are.
- Purpose statement describes how the result depends on the argument
- Design strategy: a short description of how to get from the purpose statement to the code

# The Four Design Strategies

## The Four Strategies

Domain Knowledge

Function Composition

Structural Decomposition

General Recursion

Today's topics



The diagram consists of a vertical stack of four light blue rectangular boxes. The top box is dark blue with the text 'The Four Strategies' in white. The three boxes below it are light blue and contain the text 'Domain Knowledge', 'Function Composition', and 'Structural Decomposition' respectively. A red rounded rectangle encloses these three boxes. A red arrow points from a red box at the bottom right, labeled 'Today's topics', to the right side of the red rounded rectangle.

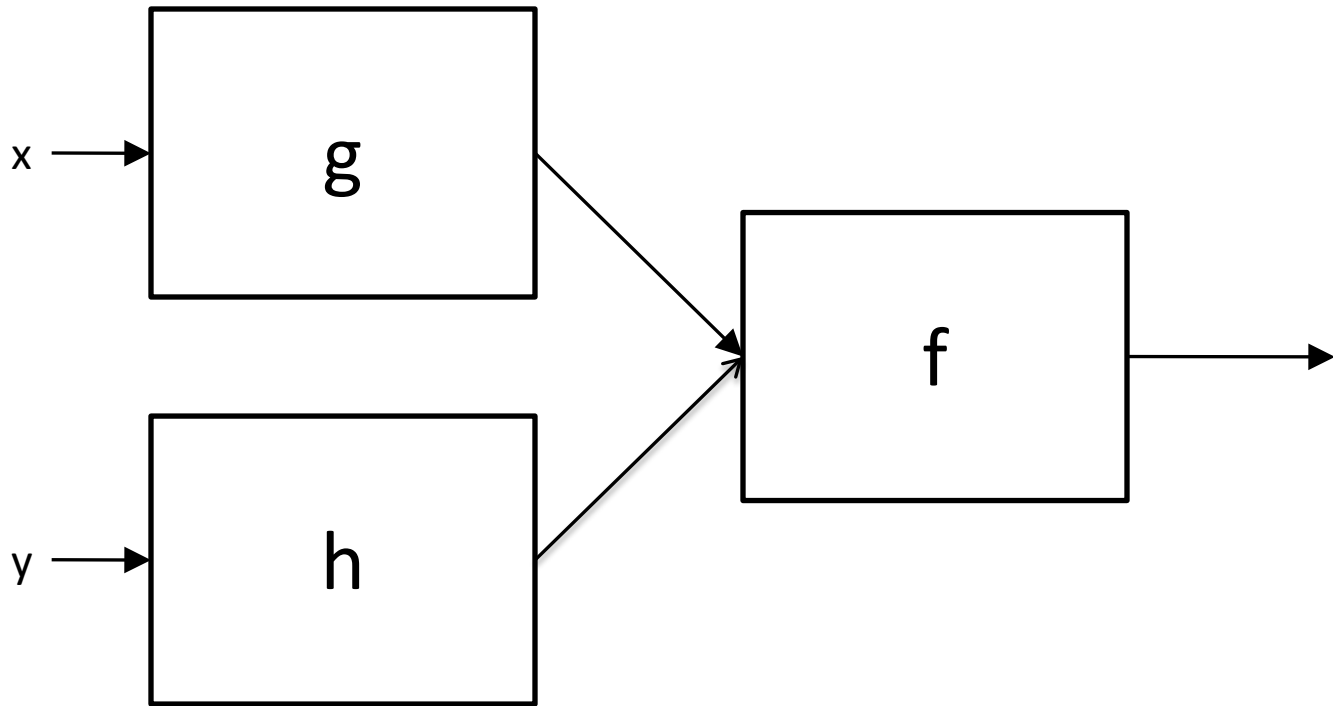
# Domain Knowledge

- The code is produced by transcribing a mathematical formula.
- No "programming knowledge" required.
- Usually used for scalar data (e.g., numbers)
- The code produced is typically very short
- It does not call any functions except for those built into the programming language
- Demo: 01-1-velocity.rkt

# Function Composition

- Used when the answer can be explained as a combination of simpler computations.
- Demo: `area-of-ring1.rkt`

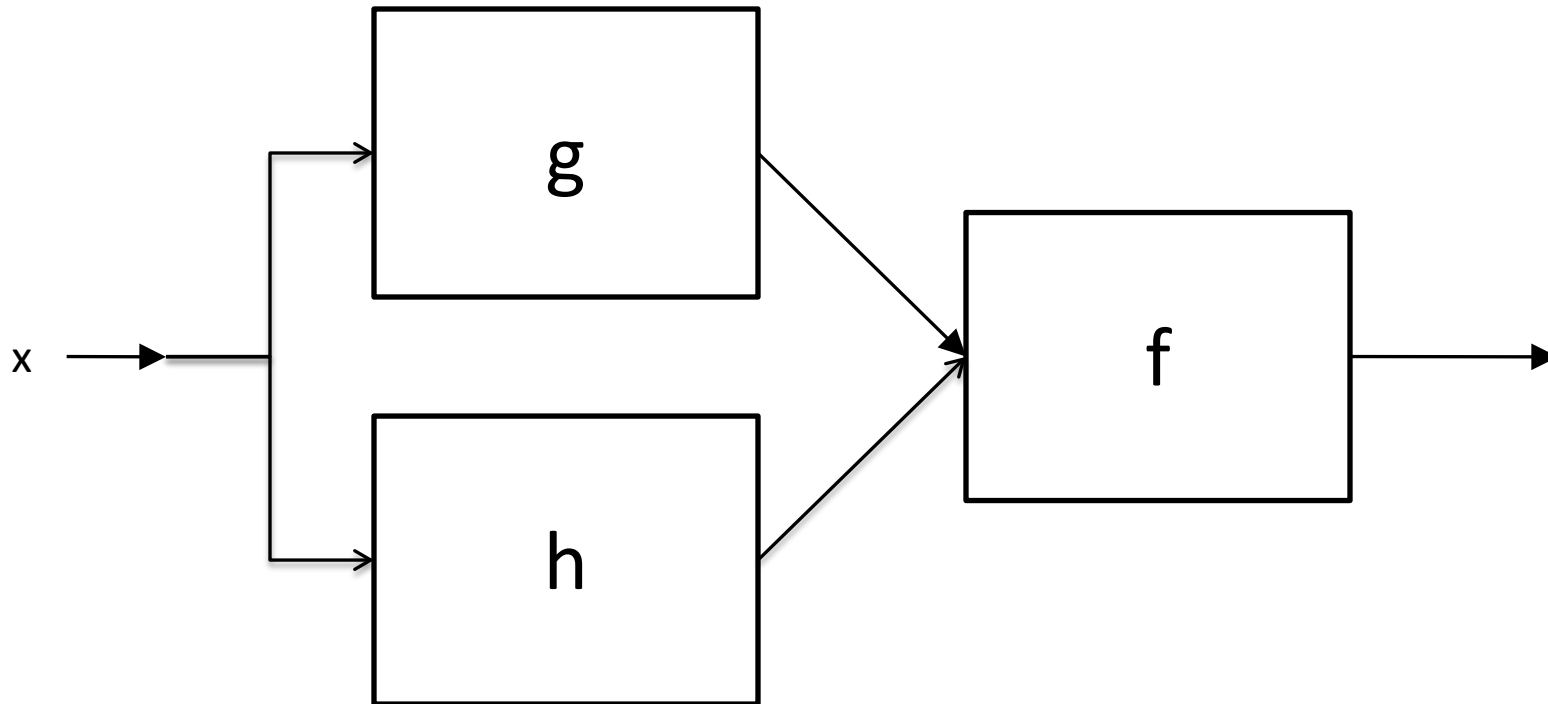
# Patterns of Function Composition (1)



**$(f (g x) (h y))$**

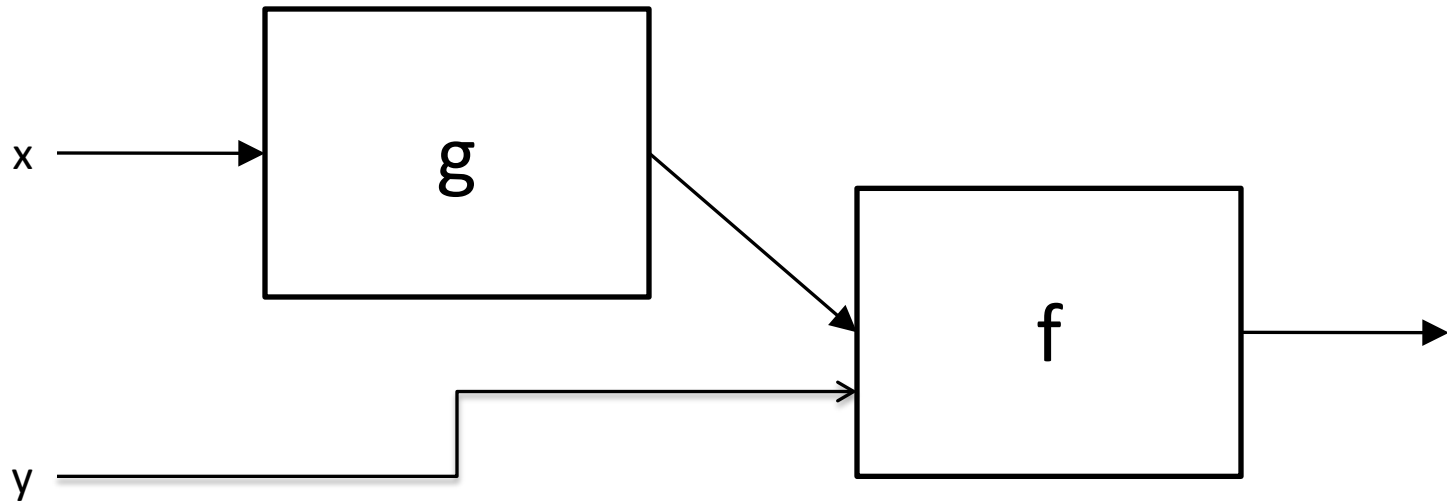


# Patterns of Function Composition (2)



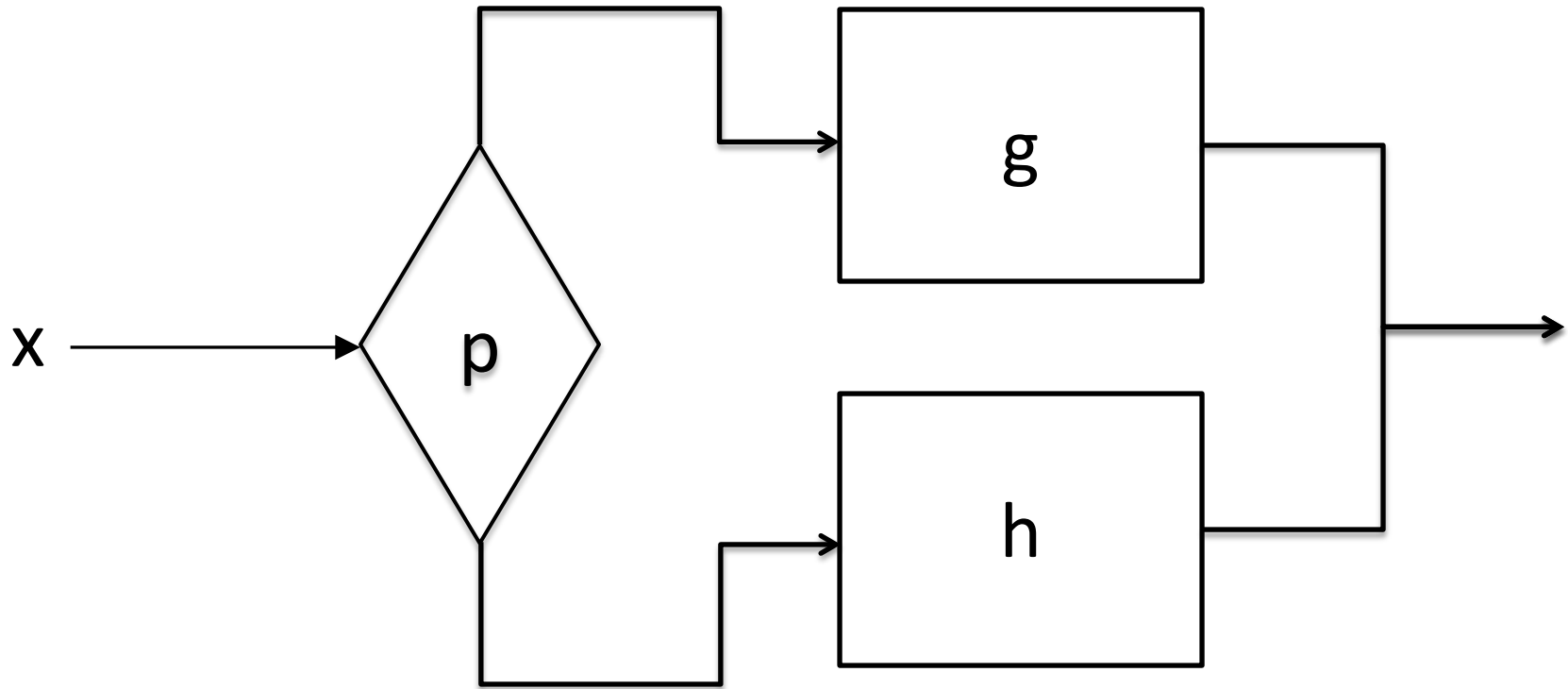
$(f \ (g \ x) \ (h \ x))$

# Patterns of Function Composition (3)



$(f (g x) y)$

# Patterns of Function Composition (4)



**`(if (p x) (g x) (h x))`**

# Keep it short!

- Domain knowledge, functional composition is for very short definitions only.
- If you have complicated junk in your function, you must have put it there for a reason. Turn it into a separate function so you can explain it and test it.

# Bad Example

```
;; ball-after-tick : Ball -> Ball
;; strategy: structural decomposition
(define (ball-after-tick b)
  (if
    (and
      (<= YUP (where b) YLO)
      (or (<= (ball-x b) XWALL
            (+ (ball-x b)
               (ball-dx b))))
      (>= (ball-x b) XWALL
            (+ (ball-x b)
               (ball-dx b)))))
    (make-ball
      (- (* 2 XWALL)
          (ball-x (straight b 1.)))
      (ball-y (straight b 1.))
      (- (ball-dx (straight b 1.))
          (ball-dy (straight b 1.)))
      (straight b 1.)))
```

```
;; ball-after-tick : Ball -> Ball
;; strategy: function composition
(define (ball-after-tick b)
  (if
    (ball-would-hit-wall? b)
    (ball-after-bounce b)
    (ball-after-straight-travel b)))
```

# What's the difference between domain knowledge and function composition?

- If a function has pieces that can be given meaningful contracts and purpose statements, then break it up and use function composition.

# Structural Decomposition

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 1.7

# Structural Decomposition

- Used when the problem can be solved by examining a piece of non-scalar data.
- Slogan:

*The shape of the data  
determines the shape of  
the program.*



# From Template to Function Definition

## Recipe for Structural Decomposition

1. Do the first four steps of the design recipe first!!
2. Make a copy of the template and uncomment it
3. Fill in the function name and add more arguments if needed
4. Fill in the blanks in the template with functional compositions of the arguments and the fields.

*Don't do anything else!*

# Following the template means filling in the blanks

```
;; template:
;; f : TaxableIncome -> ??
(define (f amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))    ...]
    [(and (<= 10000 amt) (< amt 20000)) ...]
    [(<= 20000 amt) ...]))
```

Fill in the function name, contract,  
arguments, and strategy

```
:: tax-on : TaxableIncome -> Number  
:: produces the tax on the given TaxableIncome  
:: examples: ....  
:: strategy: structural decomposition on amt  
:: [TaxableIncome]  
(define (tax-on amt)  
  (cond  
    [(and (<= 0 amt) (< amt 10000))    ...]  
    [(and (<= 10000 amt) (< amt 20000)) ...]  
    [(<= 20000 amt) ...]))
```

# Now fill in the blanks with functional compositions of the arguments

```
;; tax-on : TaxableIncome -> Number
;; produces the tax on the given TaxableIncome
;; examples: ....
;; strategy: structural decomposition on amt
;; [TaxableIncome]
(define (tax-on amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))
     0]
    [(and (<= 10000 amt) (< amt 20000))
     (* 0.10 (- amt 10000))]
    [(<= 20000 amt)
     (* 0.20 (- amt 20000))]))
```

That's all you need to do!

That's all you are allowed to do!

# Racket Demos

- 01-3-traffic-light.rkt

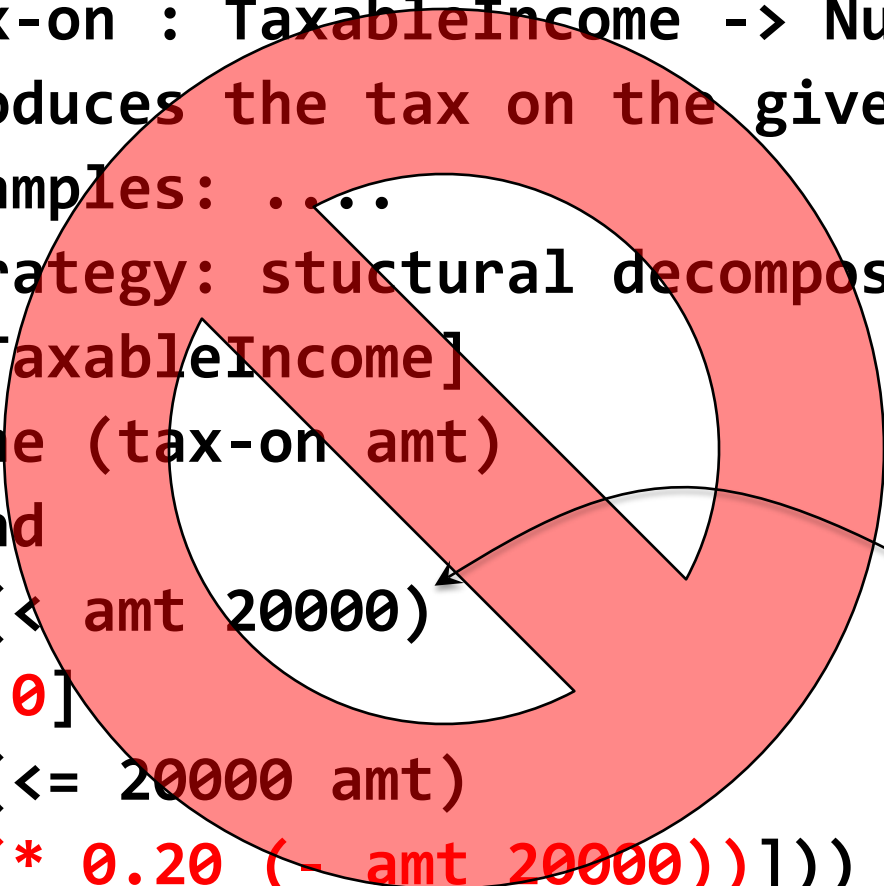
# This Definition Doesn't Follow the Template

```
;; tax-on : TaxableIncome -> Number
;; produces the tax on the given TaxableIncome
;; examples: ....
;; strategy: structural decomposition on amt
;; [TaxableIncome]
(define (tax-on amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))
     0]
    [(<= 20000 amt)
     (* 0.20 (- amt 20000))]
    [(and (<= 10000 amt) (< amt 20000))
     (* 0.10 (- amt 10000))]
  ))
```

Cases in different  
order

# Neither does this

```
;; tax-on : TaxableIncome -> Number
;; produces the tax on the given TaxableIncome
;; examples: ....
;; strategy: structural decomposition on amt
;; [TaxableIncome]
(define (tax-on amt)
  (cond
    [(< amt 20000)
     0]
    [(<= 20000 amt)
     (* 0.20 (- amt 20000))]))
```



Predicates different from template

# Structural decomposition vs. function composition


- In function composition, you can't take look at a piece of data; all you can do is pass it around.
- So `(if (inside-ball? ball x y) ... ...)` is function composition
  - here we're just passing `ball` to `inside-ball?`, which will look at the coordinates of the ball.
- `(if (< (ball-x ball) 0) ... ...)` is structural decomposition, not function composition.



# Bad Example

```
;; ball-after-tick : Ball -> Ball
;; strategy: structural decomposition
(define (ball-after-tick b)
  (if
    (and
      (<= YUP (where b) YLO)
      (or (<= (ball-x b) XWALL
            (+ (ball-x b)
               (ball-dx b)))
          (>= (ball-x b) XWALL
              (+ (ball-x b)
                 (ball-dx b)))))
    (make-ball
      (- (* 2 XWALL)
          (ball-x (straight b 1.)))
      (ball-y (straight b 1.))
      (- (ball-dx (straight b 1.))
          (ball-dy (straight b 1.)))
      (straight b 1.))
    (straight b 1.)))
```

```
;; ball-after-tick : Ball -> Ball
;; strategy: function composition
(define (ball-after-tick b)
  (if
    (ball-would-hit-wall? b)
    (ball-after-bounce b)
    (ball-after-straight-travel b)))
```



Each of these will use structural decomposition on ball

# You can only do structural decomposition on one thing at a time.

- If you need to do structural decomposition on more than one argument, decompose one argument first and pass the results on to a suitable help function or functions.
- Let's look at an example (01-4-ball-after-mouse.rkt)

# Data Definition: Ball

```
(define-struct ball (x y radius selected?))

;; A Ball is a (make-ball Number Number Number Boolean)
;; x and y are the coordinates of the center of the ball,
;; in pixels, relative to the origin of the scene.
;; radius is the radius of the ball, in pixels
;; selected? is true iff the ball has been selected for dragging.

;; TEMPLATE:
;; (define (ball-fn b)
;;   (...
;;     (ball-x b) (ball-y b) (ball-radius b)(ball-selected? b)))
```

# Data Definition: BallMouseEvent

```
;; A BallMouseEvent is a MouseEvent, which is one of
;; -- "button-down"      interp: select the current ball
;; -- "drag"             interp: drag the current ball
;; -- "button-up"        interp: deselect the current ball
;; -- any other MouseEvent  interp: ignored

;; MouseEvent is defined in the 2htdp/universe module.

;; TEMPLATE:
;; (define (bme-fn mev)
;;   (cond
;;     [(mouse=? mev "button-down") ...]
;;     [(mouse=? mev "drag") ...]
;;     [(mouse=? mev "button-up") ...]
;;     [else ...]))
```

# ball-after-mouse

```
;; ball-after-mouse :  
;;   Ball Number Number MouseEvent -> Ball  
;; Returns the ball after the given mouse event  
;; Strategy: Structural Decomposition on  
;;   mev : MouseEvent  
(define (ball-after-mouse b mx my mev)  
  (cond  
    [(mouse=? mev "button-down")  
     (ball-after-button-down b mx my)]  
    [(mouse=? mev "drag")  
     (ball-after-drag b mx my)]  
    [(mouse=? mev "button-up")  
     (ball-after-button-up b mx my)]  
    [else b])))
```

Wishlist:

1. ball-after-button-down
2. ball-after-drag
3. ball-after-button-up

# ball-after-drag

```
;; ball-after-drag : Ball Number Number -> Ball
;; Returns the ball after a drag event at the
given location.
;; strategy: structural decomposition on
;;   b : Ball
(define (ball-after-drag b x y)
  (if (ball-selected? b)
      (ball-moved-to b x y)
      b))
```

# ball-moved-to

```
;; ball-moved-to : Ball Number Number -> Ball
;; Returns a ball like the given one, except
;; that it has been moved to the given
;; coordinates.
;; strategy: structural decomposition on
;;   b : Ball
(define (ball-moved-to b x y)
  (make-ball x y
    (ball-radius b)
    (ball-selected? b)))
```

# would-balls-collide?

- Look at Examples/01-5-balls-collide.rkt



# Writing good definitions

- If your code is ugly, try decomposing things in the other order
- Remember: Keep it short!
  - If you have complicated junk in your function, you must have put it there for a reason. Turn it into a separate function so you can explain it and test it.

# Summary

- Three Design Strategies:
  - Domain knowledge
    - For very simple tasks, usually scalar data
  - Function Combination
    - Combine simpler functions in series or pipeline
    - Use with any kind of data
    - May involve domain knowledge as well
  - Structural Decomposition
    - Used for enumeration , compound, or mixed data
    - Template gives sketch of function
    - Our most important tool

Remember:

*The shape of the data  
determines the shape of  
the program.*

# Comparison of Strategies

Techniques Used	We call it:	Strategy
Domain Knowledge	➔	Domain Knowledge
Domain Knowledge + Function Calls	➔	Function Composition
Domain Knowledge + Function Calls + Inspection of Values	➔	Structural Decomposition

# Working with images and scenes

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 1.9

# Images are Scalar Data

- In Racket, images are scalar data
- Racket has:
  - Functions for creating images
  - Functions for combining images
  - Functions for finding properties of images

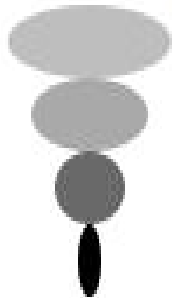
# Creating Images

- `(require 2htdp/image)`
- Functions for creating images
  - `bitmap : Filename -> Image`
  - `rectangle : Width Height Mode Color -> Image`
  - `circle : Radius Mode Color -> Image`
  - `line : x1 y1 Color -> Image`
  - `text : String Font-size Color -> Image`

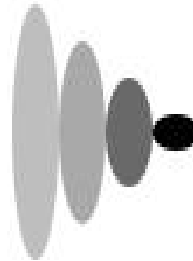
# Combining Images

- **beside**
- **above**

```
> (above (ellipse 70 20 "solid" "gray")  
          (ellipse 50 20 "solid" "darkgray")  
          (ellipse 30 20 "solid" "dimgray")  
          (ellipse 10 20 "solid" "black"))
```



```
> (beside (ellipse 20 70 "solid" "gray")  
          (ellipse 20 50 "solid" "darkgray")  
          (ellipse 20 30 "solid" "dimgray")  
          (ellipse 20 10 "solid" "black"))
```



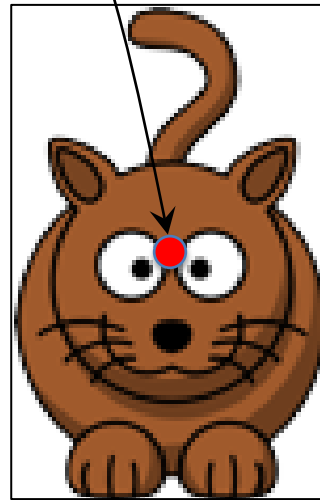
# Measuring Images

- Functions for determining image properties
  - **image-width** : **Image** -> **Number**
  - **image-height** : **Image** -> **Number**
  - **image?** : **Any** -> **Boolean**



# Bounding Box

$(x_0, y_0)$



$(x, y)$  is inside the rectangle iff  
 $(x_0 - w/2) \leq x \leq (x_0 + w/2)$   
and  $(y_0 - h/2) \leq y \leq (y_0 + h/2)$

$y = y_0 - h/2$

$h =$   
 $(\text{image-height CAT-IMAGE})$

$y = y_0 + h/2$

$w = (\text{image-width CAT-IMAGE})$

$x = x_0 - w/2$        $x = x_0 + w/2$

# Creating Scenes

- A **Scene** is an image whose origin is in the top-left corner
  - **(empty-scene width height)**
    - returns an empty scene with the given dimensions.
  - **(place-image img x y s)**
    - produces a scene just like **s**, except that image **img** is placed with its center at **(x,y)**
    - coordinates are relative to top-left of **s**

# Creating Scenes with Function Composition

- Use with function composition to create scenes with images in them
- Example: functional-scenes.rkt
- Run, then type **scene1** in the interaction window.
  - Building scene2 didn't change scene1.
  - We are always building new value; we never change an old value.

# Images and the Design Recipe: Examples

- In your examples, describe the image in words.
- EXAMPLE: Consider a function that produces an image of an arithmetic expression. You could write in your program

```
(define three-plus-two-expression
  (make-plus-expr 3 2))
```
- In your examples you could write:

```
;; (expr-to-image three-plus-two-expression)
;; = image of (+ 3 2)
```

# Images and the Design Recipe: Tests (1)

- Because of the differences in systems, using **check-equal?** on images is not reliable.
- So we need to work around this problem.

# Images and the Design Recipe: Tests

## (2)

- First, construct the correct image. Do NOT use the function you are testing to construct the image.
- EXAMPLE:  

```
(define correct-three-plus-two-image  
  (text "(+ 3 2)" 11 black))
```
- Check it visually to see that it's correct
  - Alas, this step is not automatable.

# Images and the Design Recipe: Tests

## (3)

- Test to see that the image produced by your function has the same properties (e.g. width, height) as the correct one.

**;; result of actually calling my function:**

```
(define my-three-plus-two-image  
  (expr-to-image three-plus-two-expression))
```

```
(check-equal?  
  (image-width my-three-plus-two-image)  
  (image-width correct-three-plus-two-image))
```

```
(check-equal?  
  (image-height my-three-plus-two-image)  
  (image-height correct-three-plus-two-image))
```

# Summary

- Images are ordinary scalar values
- Create and combine them using functions
- Scenes are a kind of image
  - create with empty-scene
  - build with place-image
- 2htdp/image has lots of functions for doing all this.
  - Go look at the help docs
- Can't compare images for equality
  - Test images by checking their properties