Introduction to Universe Programs

CS 5010 Program Design Paradigms
"Bootcamp"

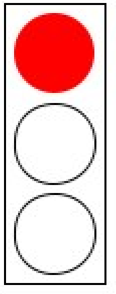
Lesson 2.1

The 2htdp/universe module

- Provides a way of creating and running an interactive machine.
- Machine will have some *state.*
- Machine can respond to *inputs*.
- Response to input is described as a *function*.
- Machine can show its state as a *scene.*
- We will use this to create interactive animations.

Traffic Light Example

- The traffic light is the machine, its state is compound information:
  - its current color AND # of ticks until next change
- Inputs will be the time (ticks)
- The traffic light can show its state as a scene, perhaps something like this:



The Falling Cat: Problem Statement

- We will produce an animation of a falling cat.
- The cat will starts at the top of the canvas, and fall at a constant velocity.
- If the cat is falling, hitting the space bar should pause the cat.
- If the cat is paused, hitting the space bar should unpause the cat.
- Demo: falling-cat.rkt

The Falling Cat: Information Analysis

- The state of the machine will consist of:
  - a number describing the position of the cat.
  - a boolean describing whether or not the cat is paused

Falling Cat: Data Design

```
(define-struct world (pos paused?))
;; A World is a (make-world Number Boolean)
;; Interpretation:
;; pos describes how far the cat has fallen, in pixels.
;; paused? describes whether or not the cat is paused.

;; template:
;; world-fn : World -> ??
;(define (world-fn w)
;  (... (world-pos w) (world-paused? w)))
```

Falling Cat 1: Information Analysis, part 2

- What inputs does the cat respond to?
- Answer: it responds to *time passing* and to *key strokes*

What kind of Key Events does it respond to?

```
;; We put in another data definition to indicate how KeyEvents
   should be interpreted:

;; A FallingCatKeyEvent is a KeyEvent that is one of
;; -- " "                   (interp: pause/unpause)
;; -- any other KeyEvent (interp: ignore)

;; template:
;; falling-cat-kev-fn : FallingCatKeyEvent ->
;(define (falling-cat-kev-fn kev)
;  (cond
;    [(key=? kev " ") ...]
;    [else ...]))
```

Many times you will need to do a "custom enumeration data". Here's how to set it up.

Look in Help Desk for definition of KeyEvent. What are the operations on KeyEvents?

This should go near the beginning of the file, with the other data definitions

Next, make a wishlist

- What functions will we need for our application?
- Write contracts and purpose statements for these functions.

Wishlist (1): How does it respond to time passing?

We express the answer as a function:

```
;; world-after-tick: World -> World
;; produce the world that should
;; follow the given world after a
;; tick.
```

we will describe responses to other inputs similarly

Wishlist (2): How does it respond to key events?

```
;; world-after-key-event : World KeyEvent -> World
;; produce the world that should follow the given world
;; after the given key event.
;; on space, toggle paused?-- ignore all others
```

Wishlist (3)

- We also need to *render* the state as a scene:

```
;; world->scene : World -> Scene
;; produce an Scene that portrays the given
   world.
```

Another response described as a function!

Wishlist (4): Running the world

```
;; main : Number -> World
;; starts the simulation with the cat in the given
   position and falling
(define (main initial-pos)
  (big-bang (make-world initial-pos false)
            (on-tick world-after-tick 0.5)
            (on-key world-after-key-event)
            (on-draw world->scene)))
```

secs/tick

names of events

functions for responses

Next: develop each of the functions

```
;; world-after-tick : World -> World
;; produce the world that should follow the given
   world after a tick

;; examples:
;; cat falling:
;; (world-after-tick (make-world 20 false))
;;  = (make-world (+ 20 CATSPEED) false)
;; cat paused:
;; (world-after-tick (make-world 20 true))
;;  = (make-world 20 true)
```

Choose strategy to match the data

- World is compound, so use structural decomposition:

```
;; strategy: structural decomposition [World]
(define (world-after-tick w)
  (... (world-pos w) (world-paused? w)))
```

- What goes in `...` ?
- It's complicated, so make it a separate function

Helper function

```
;; world-after-tick-helper : Number Boolean -> World
;; given a position and paused?, returns the next World
;; strategy: function composition
(define (world-after-tick-helper pos paused?)
  (if paused?
      (make-world pos paused?)
      (make-world (+ pos CATSPEED) paused?)))
```

- Don't need separate tests for helper functions except for debugging.

fields used more than once: a sure sign you need a help function

Tests

```
(define unpaused-world-at-20 (make-world 20 false))
(define paused-world-at-20   (make-world 20 true))
(define unpaused-world-at-28 (make-world (+ 20 CATSPEED) false))
(define paused-world-at-28   (make-world (+ 20 CATSPEED) true))

(define-test-suite world-after-tick-tests
  (check-equal?
    (world-after-tick unpaused-world-at-20)
    unpaused-world-at-28
    "in unpaused world, the cat should fall CATSPEED pixels and world should
   still be unpaused")

  (check-equal?
    (world-after-tick paused-world-at-20)
    paused-world-at-20
    "in paused world, cat should be unmoved"))

(run-tests world-after-tick-tests)
```

How does it respond to key events?

```
;; world-after-key-event : World KeyEvent -> World
;; produce the world that should follow the given world
;; after the given key event.
;; on space, toggle paused?-- ignore all others
;; examples: see tests below
;; strategy: structural decomposition [Enumeration on
   KeyEvent]

(define (world-after-key-event w kev)
  (cond
    [(key=? kev " ")
     (world-with-paused-toggled w)]
    [else w]))
```

Helper Function

```
;; world-with-paused-toggled : World -> World
;; produce a world just like the given one, but with
   paused? toggled
;; strategy: structural decomposition [w : World]
(define (world-with-paused-toggled w)
  (make-world
   (world-pos w)
   (not (world-paused? w))))

;; Don't need to test this separately, since tests for
   world-after-key-event already capture this.
```

Tests (1)

```
;; for world-after-key-event, we need 4 tests:
;; all combinations of:
;; a paused world and an unpaused world,
;; and a "pause" key event and a "non-pause" key
  event

;; Give symbolic names to "typical" values:
;; we have these for worlds,
;; now we'll add them for key events:
(define pause-key-event " ")
(define non-pause-key-event "q")
```

Tests (2)

```
(check-equal?
  (world-after-key-event
   paused-world-at-20
   pause-key-event)
  unpaused-world-at-20
  "after pause key, a paused world should become unpaused")

(check-equal?
  (world-after-key-event
    unpaused-world-at-20
    pause-key-event)
    paused-world-at-20
  "after pause key, an unpaused world should become paused")
```

Tests (3)

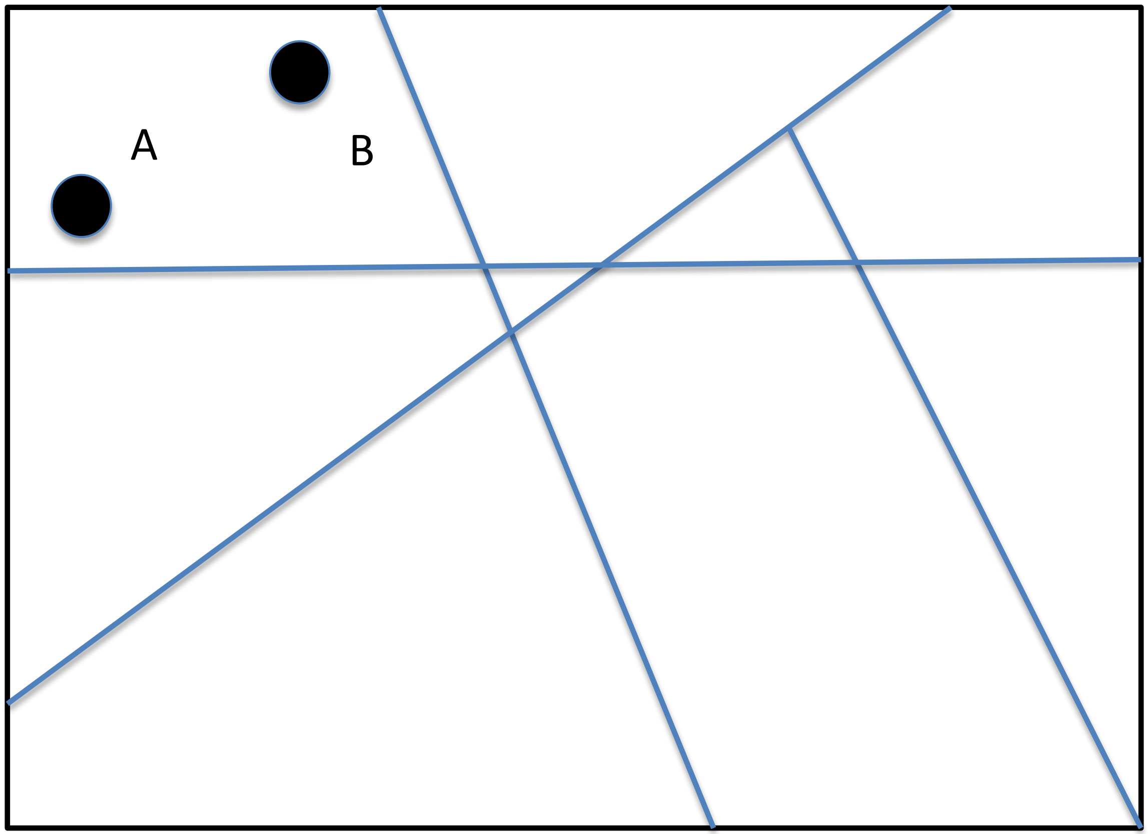
```
(check-equal?
  (world-after-key-event
    paused-world-at-20
    non-pause-key-event)
  paused-world-at-20
  "after a non-pause key, a paused world should be
   unchanged")

(check-equal?
  (world-after-key-event
    unpaused-world-at-20
    non-pause-key-event)
  unpaused-world-at-20
  "after a non-pause key, an unpaused world should be
   unchanged")
```

A Fancy Name for what we just did: Equivalence Partitioning

- Possible arguments to your function typically fall into classes for which the program yields similar results.
- Example: f2c had only 1 partition.
  - simple linear relation
- Example: world-after-key-event had 4 partitions:
  - cat paused or not
  - key is a "pause" key or not

Equivalence Partitioning

Regions of similar behavior



If the program works for input A, it will probably work for input B

Choosing test cases

- Choose tests that cover each equivalence partition.
- Choose mnemonic names for the input and output values in each partition.

What else is on our wishlist?

```
;; world->scene : World -> Scene
;; produce an Scene that portrays the given world.
;; example: (world->scene (make-world 20 ??))
;;          = (place-image CAT-IMAGE 100 20
;;                         EMPTY-CANVAS)
;; strategy: structural decomposition [World]
(define (world->scene w)
  (place-image CAT-IMAGE 100
               (world-pos w)
               EMPTY-CANVAS))
```

Tests

```
;; an image showing the cat at Y = 20
;; check this visually to make sure it's what you want
(define image-at-20 (place-image CAT-IMAGE CAT-X-COORD 20 EMPTY-CANVAS))

;; note: these only test whether world->scene calls place-image properly.
;; it doesn't check to see whether that's the right image!
;; these are not very good test strings!

(define-test-suite world->scene-tests
  (check-equal?
    (world->scene unpaused-world-at-20)
    image-at-20
    "test of (world->scene unpaused-world-at-20)")

  (check-equal?
    (world->scene paused-world-at-20)
    image-at-20
    "test of (world->scene paused-world-at-20)"))
```

Demo

- 01-10-falling-cat.rkt
- Question: how would you make the cat stop at the bottom of the screen?

Summary

- The universe module provides a way of creating and running an interactive machine.
- Machine will have some *state.*
- Machine can respond to *inputs*.
- Response to input is described as a *function*.
- Machine can show its state as a *scene.*
- We use this to create interactive animations.

How to Design Worlds

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 2.2

Review: the 2htdp/universe module

- Provides a way of creating and running an interactive machine.
- Machine will have some *state.*
- Machine can respond to *inputs*.
- Response to input is described as a *function*.
- Machine can show its state as a *scene.*
- We will use this to create interactive animations.

Recipe for Designing Worlds

How to Design Universe Programs
1. Information Analysis • What events should the world respond to? • What information changes in response to an event? • What information doesn't change in response to an event?
2. From your information analysis, write out the constant definitions and data definitions.
3. From your list of events, write a wish list of functions to be designed
4. Design the functions on your wishlist (use the design recipe!)

What's in the state?

- The state consists of the information that changes in response to a stimulus.
- Other things are constants.

Traffic Light Example

- Number of ticks until next change:
  - this changes on every tick.
  - So: State
- Current Color
  - changes when countdown timer runs out
  - So: State
- How often each color lasts before changing
  - doesn’t change
  - So: constant

Information Analysis for falling-cat

- dimensions of canvas
- x pos of cat
- y pos of cat
- current speed of cat
- image/size of cat
- acceleration of gravity g

which of these belong in the world?

which should be constants?

which need not be represented?

Info analysis

either CATSPEED or 0; represented by paused?

	falling-cat-1	falling-cat-2 (pausable)	drag w/ mouse	model gravity
dimensions of canvas	constant	constant	constant	constant
x pos of cat	constant	constant	world	world if draggable
y pos of cat	world	world	world	world
current speed of cat	constant	world	world	world
image/size of cat	constant	constant	constant	constant
acceleration	not represented	not represented	not represented	constant

Wishlist

You can use any name you like: the big-bang clauses specify which function goes with which event

```
world-after-tick:  World -> World
Purpose: given a world, produces the world state that
   should follow after a clock tick

world-after-mouse-event: World Nat Nat MouseEvt -> World
Purpose: given a world, produces the world state that
   should follow after the given mouse event

world-after-key-event: World KeyEvt -> World
Purpose: given a world, produces the world state that
   should follow after the given key event

world-to-scene : World -> Scene
Purpose: produces a scene that depicts the given world
```

Summary

- Information that can change after an event goes into the world state
- Info that doesn't change is represented by constants
- Manage your project with a wishlist
- Wishlist must include contracts and purpose statements for each function

Draggable-Cat

CS 5010 Program Design Paradigms
"Bootcamp"

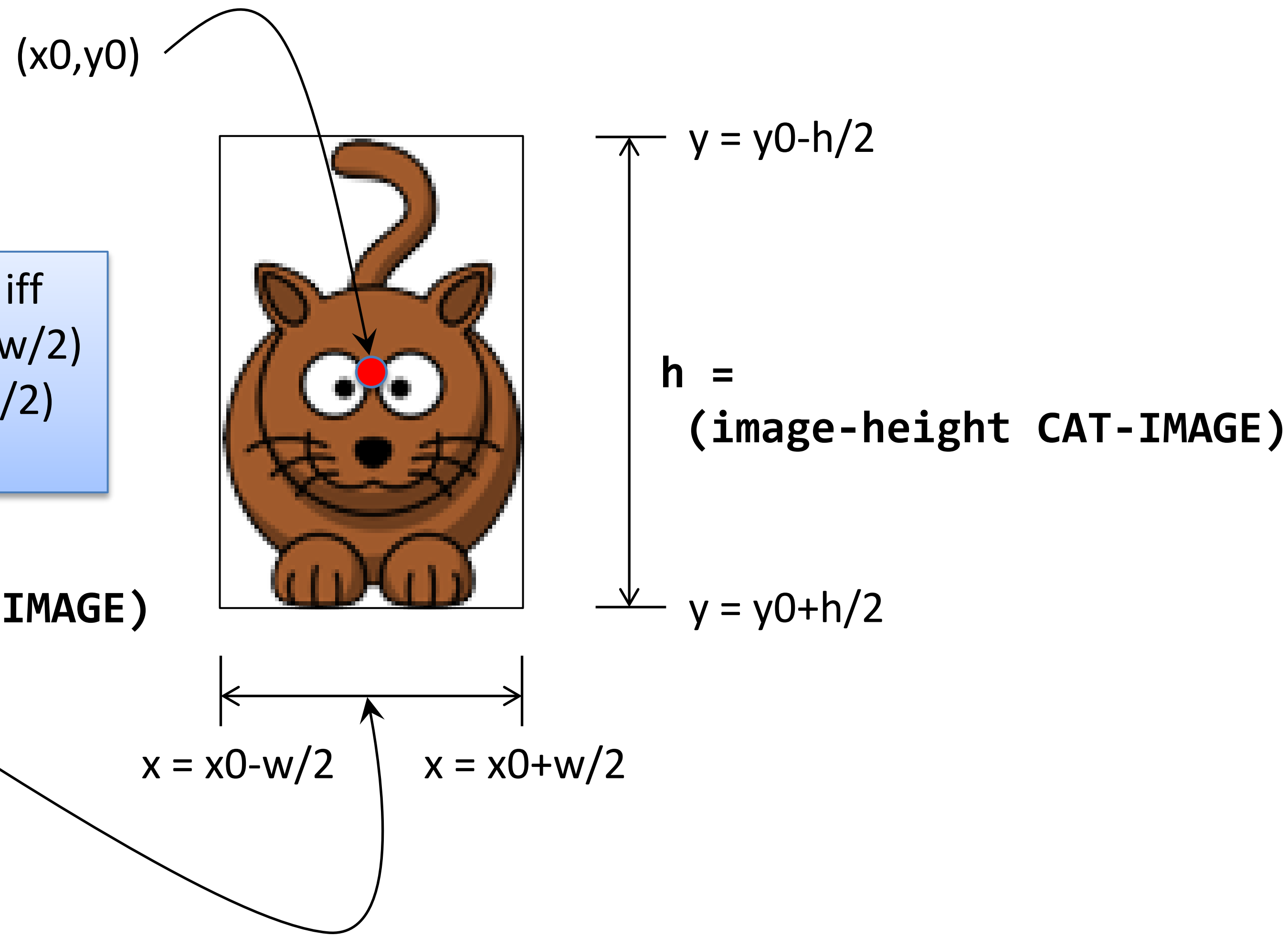
Lesson 2.3

Requirements

- Like falling cat, but user can drag the cat with the mouse.
- button-down to select, drag to move, button-up to release.
- A selected cat doesn't fall. When unselected, cat resumes its previous pausedness
  - if it was falling, it will continue to fall when released
  - if it was paused, it will remain paused when released
- Demo: draggable-cat.rkt

in-cat? relies on Bounding Box

(x,y) is inside the rectangle iff
(x0-w/2) <= x <= (x0 + w/2)
and (y0-h/2) <= y <= (y0+h/2)

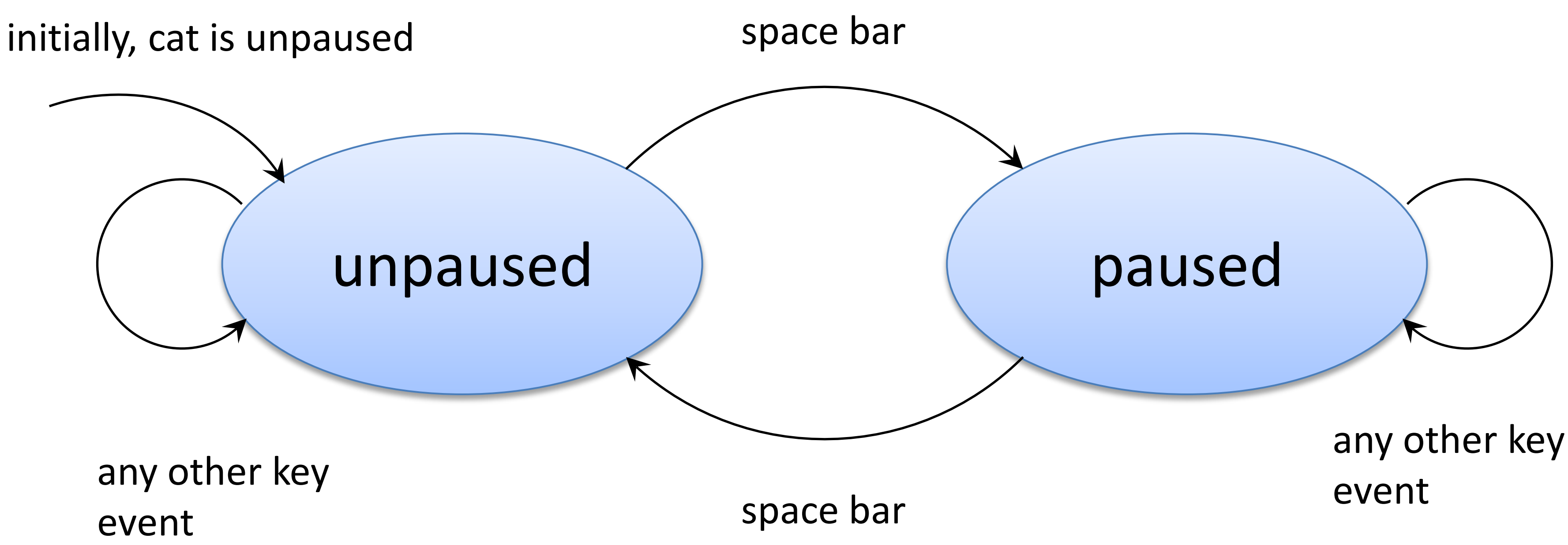


w = (image-width CAT-IMAGE)

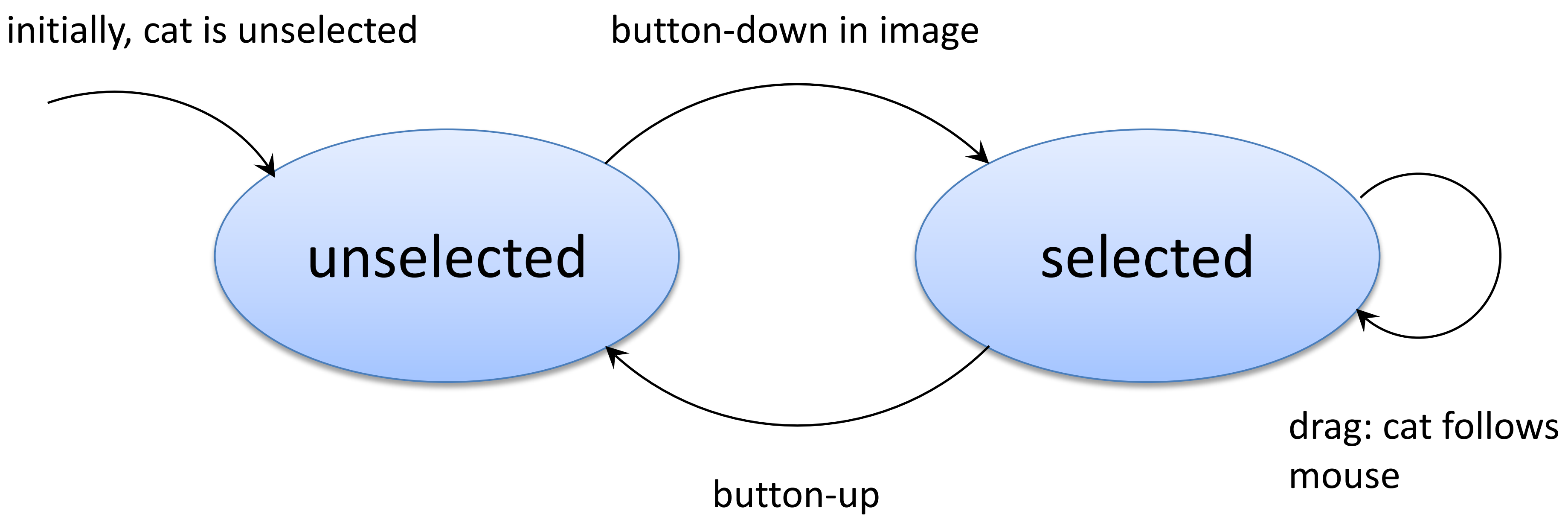
Information Analysis

- What are the possible behaviors of the cat?
  - as it falls?
  - as it is dragged?

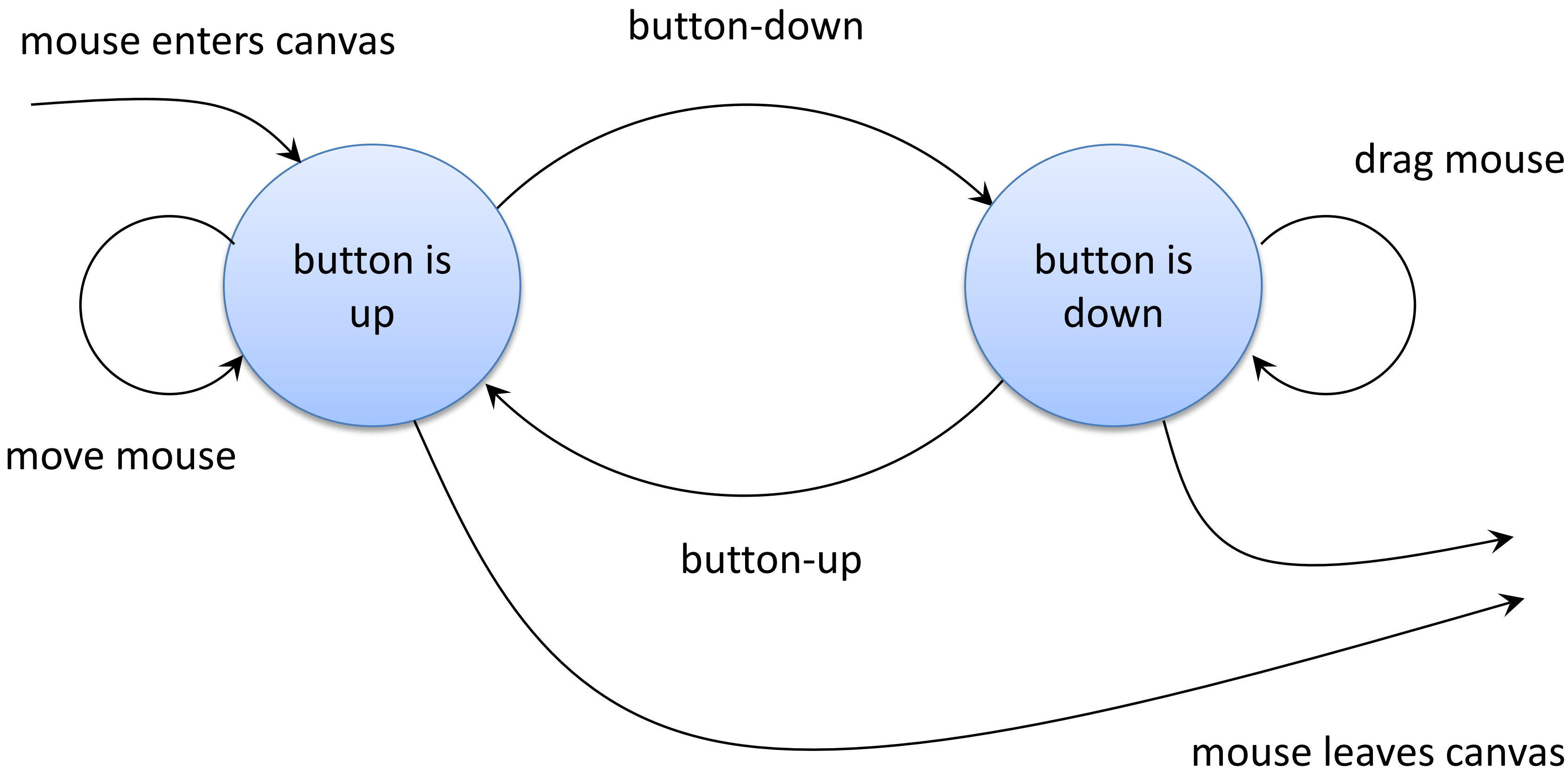
Life Cycle of a falling cat



Life Cycle of a dragged cat



Life Cycle of Mouse Movements



Data Design (1)

```
(define-struct world (x-pos y-pos paused? selected?))
;; A World is a (make-world Number Number Boolean Boolean)
;; Interpretation:
;; x-pos, y-pos give the position of the cat.
;; paused? describes whether or not the cat is paused.
;; selected? describes whether or not the cat is selected.

;; template:
;; world-fn : World -> ??
;(define (world-fn w)
; (... (world-x-pos w) (world-y-pos w)
;      (world-paused? w) (world-selected? w)))
```

Data Design (2): Mouse Events

```
;; A FallingCatMouseEvent is a MouseEvent that is
one of:
;; -- "button-down"   (interp: maybe select the cat)
;; -- "drag"          (interp: maybe drag the cat)
;; -- "button-up"     (interp: unselect the cat)
;; -- any other mouse event (interp: ignored)

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

STOP!

- Next, get all your old functions working with the new data definitions.
- Make sure your old tests work
  - Don't change your tests!
  - We used mostly symbolic names for the test inputs and test outputs, so you should just change those definitions
  - The tests themselves should work.

Testing your old functions

```
(define unpaused-world-at-20
  (make-world CAT-X-COORD 20 false false))
(define paused-world-at-20
  (make-world CAT-X-COORD 20 true false))
(define unpaused-world-at-28
  (make-world CAT-X-COORD 28 false false))
(define paused-world-at-28
  (make-world CAT-X-COORD 28 true false))
...
(check-equal?
  (world-after-key-event paused-world-at-20 pause-key-event)
  unpaused-world-at-20
  "after pause key, a paused world should become unpaused")
```

adjusted values

same tests

Everything OK?

- Good. Now we are ready to move on to the new features.

Responding to Mouse Events

```
(big-bang ...
  (on-mouse world-after-mouse-event))
```

```
world-after-mouse-event :
  World Integer Integer MouseEvent -> World
```

Look in the Help Desk for details about **`on-mouse`**

world-after-mouse-event

```
;; world-after-mouse-event :
;;    World Integer Integer FallingCatMouseEvent
;;    -> World
;; produces the world that should follow the given mouse event
;; examples:  See slide on life cycle of dragged cat
;; strategy: struct decomp on mouse events
(define (world-after-mouse-event w mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (world-after-button-down w mx my)]
    [(mouse=? mev "drag")
     (world-after-drag w mx my)]
    [(mouse=? mev "button-up")
     (world-after-button-up w mx my)]
    [else w]))
```

How to test this function?

- 3 mouse events (+ a test for the else clause)
- cat selected or unselected
  - mouse works the same way whether the cat is paused or not.
- event inside cat or not.
- 3 x 2 x 2 = 12 tests
- plus test for else clause
- plus: cat remains paused or unpaused across selection.
- Demo: draggable-cat.rkt

Recipe for Adding a Feature

Adding a New Feature to an Existing Program
1. Perform information analysis for new feature
2. Modify data definitions as needed
3. Update existing functions to work with new data definitions
4. Write wishlist of functions for new feature
5. Design new functions following the Design Recipe

Two Draggable Cats

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 2.4

Requirements

- Like draggable-cat, except:
- We have 2 cats in the scene
- Each cat can be individually selected, as in draggable-cat
- Space pauses or unpauses the entire animation
- Demo: two-draggable-cats

Information Analysis

- The world has two cats and a paused?
  - it is the whole world that is paused or not

Data Definitions: World

```
(define-struct world (cat1 cat2 paused?))
;; A World is a (make-world Cat Cat Boolean)
;; cat1 and cat2 are the two cats
;; paused? describes whether or not the world
;; is paused

;; template:
;; world-fn : World -> ??
;; (define (world-fn w)
;;   (... (world-cat1 w)
;;        (world-cat2 w)
;;        (world-paused? w)))
```

Information Analysis

- Each cat has x-pos, y-pos, and selected?
- What about paused?
  - cats aren't individually paused
  - it's the whole thing that is paused or not.

Data Definitions: Cat

```
(define-struct cat (x-pos y-pos selected?))
;; A Cat is a
;;   (make-cat Number Number Boolean)
;; Interpretation:
;; x-pos, y-pos give the position of the cat.
;; selected? describes whether or not the cat is
;; selected.

;; template:
;; cat-fn : Cat -> ??
;(define (cat-fn c)
; (... (cat-x-pos w)
       (cat-y-pos w)
       (cat-selected? w)))
```

Data Design Principles

- Every value of the information should be represented by some value of the data
  - otherwise, we lose immediately!
- Every value of the data should represent some value of the information
  - no meaningless or nonsensical combinations
  - if each cat had a paused? field, then what does it mean for one cat to be paused and the other not?

world-after-tick

```
;; world-after-tick : World -> World
;; produces the world that should follow the
given world after a tick
;; strategy: structural decomposition on
;;   w : World
(define (world-after-tick w)
  (if (world-paused? w)
    w
    (make-world
      (cat-after-tick (world-cat1 w))
      (cat-after-tick (world-cat2 w))
      false)))
```

(world-cat1 w) is a cat, so call a cat function on it

cat-after-tick

```
;; cat-after-tick : Cat -> Cat
;; produces the state of the given cat after a tick in an
;; unpaused world.

;; examples:
;; cat selected
;; (cat-after-tick selected-cat-at-20) = selected-cat-at-20
;; cat paused:
;; (cat-after-tick unselected-cat-at-20) = unselected-cat-at-28

;; strategy: structural decomposition on c : Cat

(define (cat-after-tick c)
  (cat-after-tick-helper
    (cat-x-pos c) (cat-y-pos c) (cat-selected? c)))
```

cat-after-tick-helper

```
;; cat-after-tick-helper
;;   : Number Number Boolean -> Cat
;; produces the cat that should follow one in the given
;; position in an unpaused world
;; strategy: function composition
(define (cat-after-tick-helper x-pos y-pos selected?)
  (if selected?
    (make-cat x-pos y-pos selected?)
    (make-cat
      x-pos
      (+ y-pos CATSPEED)
      selected?)))
```

world-to-scene

```
;; world-to-scene : World -> Scene
;; produces a Scene that portrays the
;;   given world.
;; strategy: structural decomposition
;;   on w : World
(define (world-to-scene w)
  (place-cat
    (world-cat1 w)
    (place-cat
      (world-cat2 w)
      EMPTY-CANVAS)))
```

The pieces are cats, so create a wishlist function to place a cat on a scene

place-cat

```
;; place-cat : Cat Scene -> Scene
;; returns a scene like the given one, but with
;; the given cat painted on it.
;; strategy : structural decomposition
;;  on c : Cat
(define (place-cat c s)
  (place-image
    CAT-IMAGE
    (cat-x-pos c) (cat-y-pos c)
    s))
```

Lists

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 2.5

How to represent info of arbitrary size?

- a phone book with many listings
- a space-invaders game with many invaders
- a presentation with many slides

- Each of these can be represented as a sequence of information items.
- There may be better ways for some of these, but we will start with sequences

Lists: A Handy Construct for Sequences

- Sequences of data items arise so often that Racket has a standard way of doing them.
  - ListOfNumbers
  - ListOfDigits
  - ListOfStrings
  - ListOfBooks

Lists of Numbers

```
A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)
```

Examples of LONs

```
                              empty
                    (cons 11 empty)
          (cons 22 (cons 11 empty))
(cons 33 (cons 22 (cons 11 empty)))
                    (cons 33 empty)
```

```
A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)
```

Lists of Digits

```
A Digit is one of
 "0" | "1" | "2" | ... | "9"

A List of Digits (LOD) is one of:
-- empty
-- (cons Digit LOD)
```

Examples of LODs

```
                              empty
                    (cons "3" empty)
          (cons "2" (cons "3" empty))
(cons "4" (cons "2" (cons "3" empty)))
```

- Not LODs:

```
  (cons 4 (cons "2" (cons "3" empty)))
    (cons (cons "3" empty)
          (cons "2" (cons "3" empty)))
```

```
A List of Digits (LOD) is one of:
-- empty
-- (cons Digit LOD)
```

Lists of Books

```
A Book is a (make-book ...) .

A List of Books (LOB) is one of:
-- empty
-- (cons Book LOB)
```

Examples of LOBs

```
(define book1 (make-book ...))
(define book2 (make-book ...))
(define book3 (make-book ...))
```

```
                                      empty
                         (cons book1 empty)
            (cons book2 (cons book1 empty))
(cons book2 (cons book2 (cons book1 empty))
```

- Not a LOB:

```
    (cons 4 (cons book2 (cons book1 empty))
```

A List of Books (LOB) is one of:
-- empty
-- (cons Book LOB)

This data definition is *self-referential*

```
A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)
```

We also call this a *recursive* data definition

This one is self-referential, too

```
A Digit is one of
 "0" | "1" | "2" | ... | "9"

A List of Digits (LOD) is one of:
-- empty
-- (cons Digit LOD)
```

The General Pattern

```
A ListOf<X> is one of
-- empty
   interp: a sequence with no elements
-- (cons X ListOf<X>)
   interp: (cons x lst) represents a sequence
   whose first element is x and whose
   other elements are represented by lst.

A LON is a ListOf<Number>
A LOD is a ListOf<Digit>
A LOB is a ListOf<Book>
```

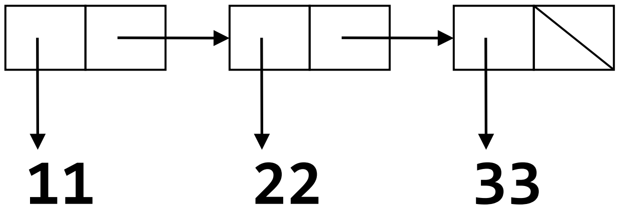
List Notation

Constructor notation:

```
(cons 11
  (cons 22
     (cons 33
           empty))))
```

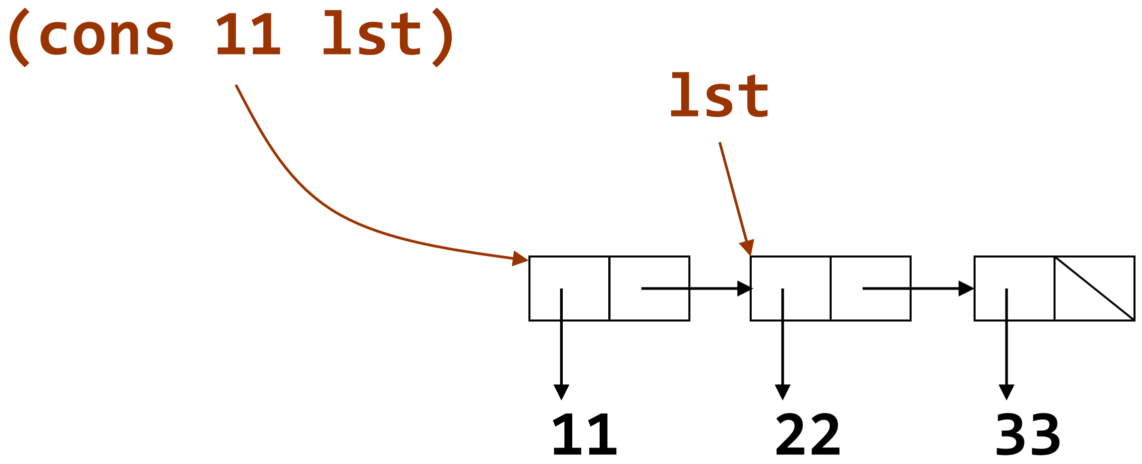
List notation: `(list 11 22 33)`

Internal representation:



`write`-style (output only): `(11 22 33)`

Implementation of **cons**




```
          lst  =    (list 22 33)

(cons 11 lst) = (list 11 22 33)
```

Operations on Lists

empty? : ListOf<X> -> Boolean

Given a list, returns true iff the list is empty

Operations on Lists

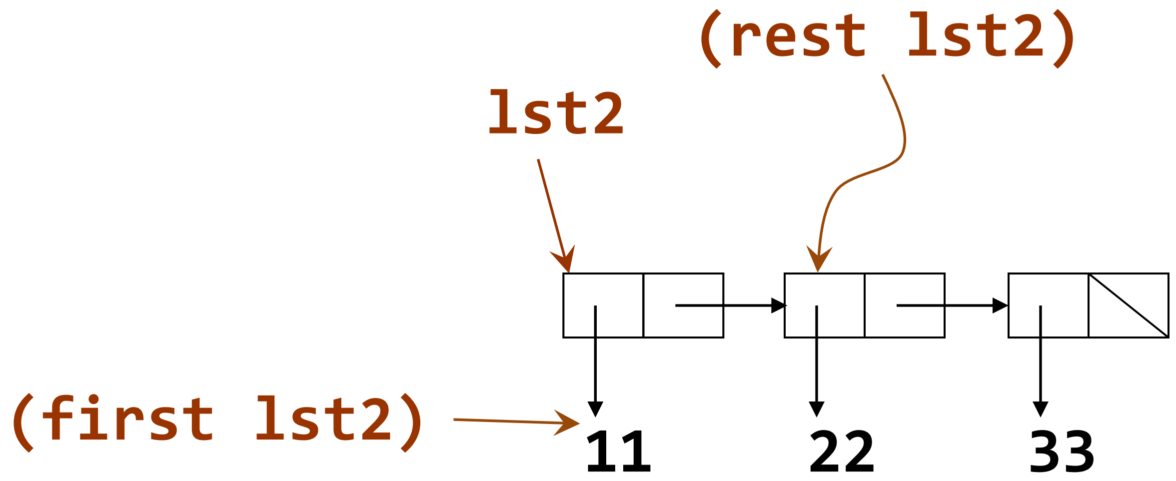
If you use the template for lists, the invariant will always be satisfied when you call the function.

```
first  : ListOf<X> -> X
Given a list, returns its first
element.
INVARIANT: the list is non-empty

rest   : ListOf<X> -> ListOf<X>
Given a list, returns the list of all
its elements except the first.
INVARIANT: the list is non-empty
```

Examples

```
(empty?                      empty)    = true
(empty?           (cons 11 empty))  = false
(empty? (cons 22 (cons 11 empty))) = false

(first (cons 11 empty)) = 11
(rest  (cons 11 empty)) = empty

(first (cons 22 (cons 11 empty))) = 22
(rest  (cons 22 (cons 11 empty))) = (cons 11 empty)

(first empty)  → Error! (Precondition failed)
(rest  empty)  → Error! (Precondition failed)
```

Implementation of **`first`** and **`rest`**




```
        lst2  = (list 11 22 33)
(first lst2) = 11
(rest  lst2) = (list 22 33)
```

Properties of **`cons`**, **`first`**, and **`rest`**

```
(first (cons v l)) = v

(rest (cons v l)) = l
```

If **`l`** is non-empty, then

```
(cons (first l) (rest l)) = l
```

Using the List Template

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 2.6

This data definition is *self-referential*

```
A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)
```

We also call this a *recursive* data definition

Template for List data

```
;; list-fn : ListOf<X> -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (list-fn (rest lst)))]))
```

Observe that **`lst`** is non-empty when **`first`** and **`rest`** are called, so their invariants are satisfied.

This template is *self-referential*

```
;; list-fn : ListOf<X> -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (list-fn (rest lst)))]))
```

`(rest lst)` is a **`ListOf<X>`**, so call **`list-fn`** on it

Self-reference in the data definition leads to self-reference in the template.

The template questions

```
;; list-fn : ListOf<X> -> ??
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (list-fn (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: lon-length

```
lon-length : ListOf<Number> -> Number
Given a ListOf<Number>, returns its length
(lon-length empty) = 0
(lon-length (cons 11 empty)) = 1
(lon-length (cons 33 (cons 11 empty))) = 2
Strategy: Structural Decomposition on lst :
  ListOf<Number>
```

Example: lon-length

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (lon-length (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: lon-length

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
               (lon-length (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

The code is self-referential, too

```
lon-length : LON -> Number
Given a LON, find its length
(define (lon-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
               (lon-length (rest lst)))]))
```

Self-reference in the data definition leads to self-reference in the template;
Self-reference in the template leads to self-reference in the code.

Example: lon-sum

```
lon-sum : LON -> Number
Given a LON, returns its sum
(lon-sum empty) = 0
(lon-sum (cons 11 empty)) = 11
(lon-sum (cons 33 (cons 11 empty))) = 44
(lon-sum (cons 10 (cons 20 (cons 3 empty)))) = 33
Strategy: Structural Decomposition on lst : LON
```

Example: lon-sum

```
lon-sum : LON -> Number
(define (lon-sum lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (lon-sum (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: lon-sum

```
lon-sum : LON -> Number
(define (lon-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+   (first lst)
               (lon-sum (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Watch this work:

```
(lon-sum (cons 11 (cons 22 (cons 33 empty))))
= (+ 11  (lon-sum (cons 22 (cons 33 empty))))
= (+ 11  (+ 22    (lon-sum (cons 33 empty))))
= (+ 11  (+ 22    (+ 33    (lon-sum empty))))
= (+ 11  (+ 22    (+ 33    0)))
= (+ 11  (+ 22    33))
= (+ 11  55)
= 66
```

Example: double-all

```
double-all : LON -> LON
Given a LON, produces a list just like
  the original, but with each number
  doubled
(double-all empty) = empty
(double-all (cons 11 empty))
 = (cons 22 empty)
(double-all (cons 33 (cons 11 empty)))
 = (cons 66 (cons 22 empty))
```

Example: double-all

```
double-all : LON -> LON
Given a LON, produce a list just like the
  original, but with each number doubled
Strategy: Struc. Decomp. on lst: LON
(define (double-all lst)
  (cond
    [(empty? lst) empty]
    [else (cons (* 2 (first lst))
                (double-all (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: remove-evens

```
remove-evens : LON -> LON
Given a LON, produces a list just like the original, but
  with all the even numbers removed
(remove-evens empty) = empty
(remove-evens (cons 12 empty)) = empty
(define list-22-11-13-46-7
  (cons 22
   (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-evens list-22-11-13-46-7)
  = (cons 11 (cons 13 (cons 7 empty)))
```

Example: remove-evens

What's the answer for the empty list?

```
remove-evens : LON -> LON
Given a LON, produces a list just like the original, but
   with all the even numbers re
(define (remove-evens lst)
  (cond
    [(empty? lst) ...]
    [ else (... (first lst)
                (remove-evens (rest lst)))]))
```

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: remove-evens

What's the answer for the empty list?

```
remove-evens : LON -> LON
Given a LON, produces a list just like the original, but
   with all the even numbers re
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [ else (... (first lst)
                (remove-evens (rest lst)))]))
```

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: remove-evens

```
remove-evens : LON -> LON
Given a LON, produces a list just like the original, but
   with all the even numbers removed
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (remove-evens (rest lst))
              (cons (first lst)
                    (remove-evens (rest lst))))]))

Add to wishlist:  even? : Number -> Boolean
```

Example: remove-first-even

```
remove-first-even : LON -> LON
Given a LON, produces a list just like the original, but
  with the first even number removed
(remove-first-even empty) = empty
(remove-first-even (cons 12 empty)) = empty
(define list-22-11-13-46-7
  (cons 22
   (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-first-even list-22-11-13-46-7)
  = (cons 11 (cons 13 (cons 46 (cons 7 empty))))
```

Why is this not a good test?

Example: remove-first-even

```
remove-first-even : LON -> LON
Given a LON, produces a list jus                        t
  with all the even numbers rem
(define (remove-first-even lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (remove-first-even (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: remove-first-even

```
remove-first-even : LON -> LON
Given a LON, produces a list jus                    t
  with all the even numbers rem
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [else (... (first lst)
               (remove-first-even (rest lst)))]))
```

What's the answer for the empty list?

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Example: remove-first-even

```
remove-first-even : LON -> LON
Given a LON, produces a list just like the original, but
   with all the even numbers removed
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (rest lst)
              (cons (first lst)
                    (remove-first-even
                      (rest lst))))]))
```

Here we don't recur; that's OK

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Lists of Structures

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 2.7

Lists of Structures

- Lists of structures occur all the time
- Important to have interpretation in the data definitions
- Programming with these is no different:
  - use the **`ListOf<X>`** template
  - Follow the Recipe!

Books, again

```
(define-struct book (author title on-hand price))

A Book is a
 (make-book String String Number Number)
Interpretation:
author is the author’s name
title is the title
on-hand is the number of copies on hand
price is the price in USD
```

Template for Book

```
book-fn : Book -> ??
(define (book-fn b)
  (... (book-author b)
       (book-title b)
       (book-on-hand b)
       (book-price b)))
```

ListofBooks

```
;; A ListOfBooks (LOB) is either
;; -- empty
;; -- (cons Book LOB)

;; lob-fn : LOB -> ??
;; (define (lob-fn lob)
;;   (cond
;;     [(empty? lob) ...]
;;     [else (...
;;             (first b)
;;             (lob-fn (rest lob)))]))
```

Example

```
(define lob1
  (list
    (make-book "Felleisen"   "HtDP/1" 20  7)
    (make-book "Wand"        "EOPL"    5 50)
    (make-book "Shakespeare" "Hamlet"  0  2)
    (make-book "Shakespeare" "Macbeth" 0 10)))
```

Exercise [books.rkt]

```
;; books-out-of-stock : LOB -> LOB
;; returns a list of the books that are out of
  stock in the given LOB
;; Example:
;; (books-out-of-stock lob1) =
;;  (list
;;    (make-book "Shakespeare" "Hamlet" 0 2)
;;    (make-book "Shakespeare" "Macbeth" 0 10))
;; Strategy: structural decomposition on lob : LOB
```

Exercise [books.rkt]

```
;; inventory-total-value : LOB -> Number
;; Returns the value of all the copies on hand
;; of all the books in the given LOB
;; EXAMPLE:
;; (inventory-total-value lob1) = 390
```

Summary: Self-Referential or Recursive Information

- Represent arbitrary-sized information using a *self-referential* (or *recursive*) data definition.
- Self-reference in the data definition leads to self-reference in the template.
- Self-reference in the template leads to self-reference in the code.
- Writing functions on this kind of data is easy: just Follow The Recipe!

Non-Empty Lists

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 2.8

Empty lists

- Most computations on lists make sense on empty lists
  - **`(sum empty) = 0`**
  - **`(product empty) = 1`**
  - **`(double-all empty) = empty`**
  - etc, etc.

Non-empty lists

- But some computations don't make sense for empty lists
  - min, max
  - average

Non-Empty Lists

- For these problems, the list template doesn't make sense, either
- For these problems, we use a different data definition and a different template that is tuned for dealing with lists that are always non-empty

Data Definition for Non-Empty List

```
;; A NonEmptyListOfSardines is one of
;; -- (cons Sardine empty)
;; -- (cons Sardine
;;          NonEmptyListOfSardines)
```

Template for Non-Empty List

```
;; nelist-fn : NonEmptyListOfSardines -> ??
(define (nelist-fn ne-lst)
  (cond
    [(empty? (rest ne-lst)) (... (first ne-lst))]
    [else (...
            (first ne-lst)
            (nelist-fn (rest ne-lst)))]))
```

(rest ne-lst) is a **NonEmptyListOfSardines** so call **nelist-fn** on it

Template Questions for Non-Empty Lists

```
;; nelist-fn : NonEmptyListOfSardines -> ??
(define (list-fn ne-lst)
  (cond
    [(empty? (rest ne-lst)) (... (first ne-lst))]
    [else (...
            (first ne-lst)
            (list-fn (rest ne-lst)))]))
```

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

What is the answer for a list of length 1?

Non-Empty Lists: The General Pattern

```
A NonEmptyListOf<X> is one of
-- (cons X empty)
   interp: a list with a single X
-- (cons X NonEmptyListOf<X>)
   interp: (cons x lst) represents a
   sequence whose first element is x
   and whose other elements are
   represented by lst.
```

Template Questions for Non-Empty Lists

```
;; nelist-fn : NonEmptyListOf<X> -> ??
(define (list-fn ne-lst)
  (cond
    [(empty? (rest ne-lst)) (.... (first ne-lst))]
    [else (...
            (first ne-lst)
            (list-fn (rest ne-lst)))]))
```

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

What is the answer for a list of length 1?

Example: max

```
;; list-max : NonEmptyListOfNumber -> Number
;; given a non-empty list of numbers, returns the largest
;; element
(define (list-max ne-lst)
  (cond
    [(empty? (rest ne-lst)) (first ne-lst)]
    [else (max
            (first ne-lst)
            (list-max (rest ne-lst)))]))
```

Example: average

```
lon-avg : LON -> Number
Given a non-empty LON, returns its average
(lon-avg (cons 11 empty)) = 11
(lon-avg (cons 33 (cons 11 empty))) = 22
(lon-avg (cons 33 (cons 11 (cons 11 empty)))) = 55/3
```

Example: average

```
;; lon-avg : NELON -> Number
;; Given a non-empty LON, returns its average
;; strategy: structural decomposition
(define (lon-avg ne-lst)
  (cond
    [(empty? (rest ne-lst)) (first ne-lst)]
    [else (..b..
            (first ne-lst)
            (lon-avg (rest ne-lst)))]))
```

If we knew the answer for the rest of the list, and we knew the first of the list, how could we combine them to get the answer for the whole list?

Oops...

- **`(lon-avg (list 33 11 11))`** = 55/3

  ➔ **`(... 33 11)`** = 55/3

  `11`

- **`(lon-avg (list 33   11))`** = 22

  ➔ **`(... 33 11)`** = 22

- Can't have both!

Function Composition to the Rescue!

```
lon-avg : NELON -> Number
Given a non-empty LON, returns its average
Strategy: functional composition
(define (lon-avg lst)
  (/ (lon-sum lst) (lon-length lst)))
```