Introduction to Universe Programs

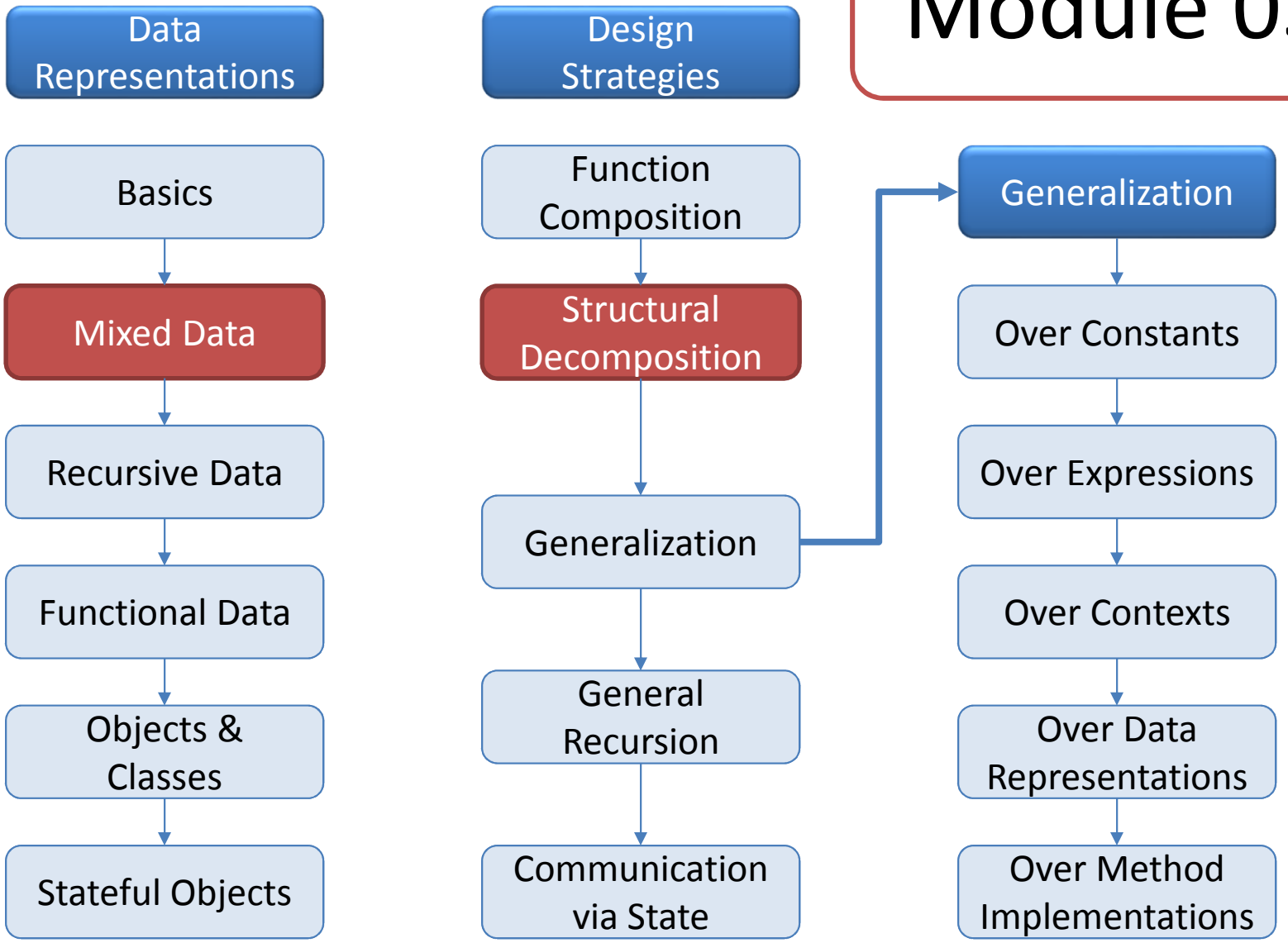
CS 5010 Program Design Paradigms "Bootcamp"

Lesson 3.1

Module Outline

- We will learn about the Universe module, which we will use to create interactive animations.
- We will learn how to do “iterative refinement”– that is, adding features to a program
- We will learn more about how to design data.
- We’ll do all this in the context of 3 versions of a simple system.

Module 03


Data Representations
Basics
Mixed Data
Recursive Data
Functional Data
Objects & Classes
Stateful Objects
Design Strategies
Function Composition
Structural Decomposition
Generalization
General Recursion
Communication via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

Let's see where we are

The Six Principles of this course
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 values
4. Design Functions Systematically
5. Design Systems Iteratively
6. Pass values when you can, share state only when you must.

The System Design Recipe
1. Write a purpose statement for your system.
2. Design data to represent the relevant information in the world.
3. Make a wishlist of main functions. Write down their contracts and purpose statements.
4. Design the individual functions. Maintain a wishlist (or wishtree) of functions you will need to write.

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
6. Repeat for the next new feature

Learning Objectives for this lesson

- In this lesson, you will learn about the 2htdp/universe module, which allows you to create interactive animations.
- At the end of this lesson, students should be able to:
  - Use the 2htdp/universe module to create a simple interactive animation, including:
    - Analyzing data to determine whether it should be constant or part of the world state,
    - Writing data definitions for worlds, key events, and mouse events, and
    - Writing code to handle the various events during the animation.
  - Explain the steps of the System Design Recipe

The 2htdp/universe module

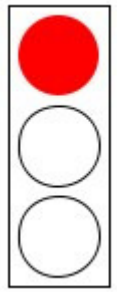
- Provides a way of creating and running an interactive machine.
- Machine will have some *state.*
- Machine can respond to *inputs*.
- Response to input is described as a *function*.
- Machine can show its state as a *scene.*
- We will use this to create interactive animations.

Example: a traffic light

- The light is a machine that responds to time passing; as time passes, the light goes through its cycle of colors.
- The state of the machine consists of its current color and the amount of time (in ticks) until the next change of color. At every tick, the amount of time decreases by 1.
- When the timer reaches 0, the light goes to its next color (from green to yellow, from yellow to red, from red to green), and the timer is reset to the number of ticks that light should stay in its new color.
- In addition, the traffic light should be able to display itself as scene, perhaps like the one on the slide.

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). At every tick, the timer is decremented.
- When the timer reaches 0, the light goes to its next color (from green to yellow, from yellow to red, from red to green), and the timer is reset to the number of ticks that light should stay in its new color.
- The traffic light can show its state as a scene, perhaps something like this:



A second example: 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.

falling-cat.rkt demo

YouTube link

The Falling Cat: Information Analysis

- There are the only two things that change as the animation progresses: the position of the cat, and whether or not the cat is paused. So that's what we put in the state:
- The state of the machine will consist of:
  - an integer describing the y-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 Integer 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?

- It responds to the space character, which is represented by the string " " that consists of a single space.
- All other key events are ignored.

Next, make a wishlist

- What functions will we need for our application?
- Write contracts and purpose statements for these functions.
- Then design each function in turn.

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

We express the answer as a function:

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

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

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

Here we've written the purpose statement in two parts. The first is the general specification ("produces the world that should follow the given world after the given key event"), and the second is a more specific statement of what that world is.

Wishlist (3)

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

```
;; world-to-scene : World -> Scene
;; RETURNS: a Scene that portrays the given
;; world.
```

Another response described as a function!

Wishlist (4): Running the world

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
```

Here the function has an effect in the real world (like reading or printing). We document this by writing an EFFECT clause in our purpose statement.

For now, functions like **`main`** will be our only functions with real-world effects. All our other functions will be pure: that is, they compute a value that is a mathematical function of their arguments. They will not have side-effects.

Side-effects make it much more difficult to understand what a function does. We will cover these much later in the course.

Next: develop each of the functions

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

;; EXAMPLES:
;; cat falling:
;; (world-after-tick unpaused-world-at-20)
;;  = unpaused-world-at-28
;; cat paused:
;; (world-after-tick paused-world-at-20)
;;  = paused-world-at-20
```

We add some examples. We've included some commentary and used symbolic names so the reader can see what the examples illustrate.

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 we'll make it a separate function

Helper function

```
;; world-after-tick-helper : Integer Boolean -> World
;; GIVEN the position of the cat 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.

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))

(begin-for-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"))
```

How does it respond to key events?

```
;; world-after-key-event :
;;    World KeyEvent -> World
;; GIVEN: a world w
;; RETURNS: 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: cases on kev : KeyEvent
(define (world-after-key-event w kev)
  (cond
    [(key=? kev " ")
     (world-with-paused-toggled w)]
    [else w]))
```

We make a decision based on **kev**, and pass the data on to a help function to do the real work.

Requirements for Helper Function

```
;; world-with-paused-toggled : World -> World
;; RETURNS: a world just like the given one, but with
;;  paused? toggled
```

If this helper function does what it's supposed to, then world-after-key-event will do what *it* is supposed to do.

Tests (1)

```
;; for world-after-key-event, we have 4 equivalence
;; classes: 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")
```

Tests (4)

```
(define (world-after-key-event w kev) ...)
```

Here's how we lay out the tests in our file.

```
(begin-for-test
  (check-equal? ...)
  (check-equal? ...)
  (check-equal? ...)
  (check-equal? ...))
```

```
(define (world-with-paused-toggled? w) ...)
```

Contract, purpose function, etc., for world-with-paused-toggled?

Now we're ready to design our help function

Definition for Helper Function

```
;; world-with-paused-toggled : World -> World
;; RETURNS: a world just like the given one, but with
;;  paused? toggled
;; STRATEGY: structural decomposition on 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 test it.

What else is on our wishlist?

```
;; world-to-scene : World -> Scene
;; RETURNS: a Scene that portrays the given world.
;; EXAMPLE:
;; (world-to-scene (make-world 20 ??))
;;  = (place-image CAT-IMAGE CAT-X-COORD 20 EMPTY-CANVAS)
;; STRATEGY: structural decomposition on w : World
(define (world-to-scene w)
  (place-image CAT-IMAGE CAT-X-COORD
               (world-pos w)
               EMPTY-CANVAS))
```

Here's where we decompose **w**. Note that we don't need **(world-paused? w)**. That's ok– this still follows the template.

Testing world-to-scene

```
;; 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))

;; these tests are only helpful if image-at-20 is the right image.
;; Here I've made the strings into error messages.  This is often
;; works well.

(begin-for-tests
  (check-equal?
    (world->scene unpaused-world-at-20)
    image-at-20
    "unpaused world yields incorrect image")

  (check-equal?
    (world->scene paused-world-at-20)
    image-at-20
    "paused world yields incorrect image"))
```

Last wishlist item

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
```

The real purpose of **main** is not to return a useful value; instead, its purpose is have some visible effect in the real world– in this case, to display some things on the real screen and take input from the real user. We document this in the purpose statement by writing an EFFECT clause.

Template for big-bang

```
;; big-bang
;; EFFECT : runs a world with the specified event handlers.
;; RETURNS: the final state of the world
(big-bang
  initial-world
  (on-tick tick-handler rate)
  (on-key  key-handler)
  (on-draw render-fcn)))
```

frame rate in secs/tick

names of events

functions for responses

There are other events that big-bang recognizes, see the Help Desk for details

Putting the pieces together

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
;; STRATEGY: function composition
(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-to-scene)))
```

main just calls **big-bang**, so the strategy is function composition.

Let's walk through falling-cat.rkt

- Note: this video differs from our current technology in a couple of ways:
  - it talks about test suites; these are replaced by **begin-for-test**.
  - it talks about "partition data" and gives a template for FallingCatKeyEvents. We've simplified the presentation-- now we just have KeyEvents, which are scalars (no template needed), and we take them apart using the "Cases" strategy.

falling-cat.rkt readthrough

YouTube link

The System Design Recipe

- In building this system, we were actually following a recipe.
- This recipe is so widely usable that we give it a name: the ***System Design Recipe***.
- Here it is– you can see that it matches what we did.

The System Design Recipe

The System Design Recipe
1. Write a purpose statement for your system.
2. Design data to represent the relevant information in the world.
3. Make a wishlist of main functions. Write down their contracts and purpose statements.
4. Design the individual functions. Maintain a wishlist (or wishtree) of functions you will need to write.

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.
- We built a system, using the *system design recipe*.

Next Steps

- Study the files in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 3.1
- Go on to the next lesson