Design Strategies 1: Combine Simpler Functions

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 1.7

Learning Objectives

- At the end of this lesson, the student should be able to define short functions by composing existing functions.

Introduction

- In this lesson, you will learn about Steps 4 and 5 of the design recipe: Design Strategies and Function Definitions.
- We will start with the simplest design strategy: Combine Simpler Functions

Programs are sets of Functions

- We organize our programs as sets of *functions*.
- A function takes an argument (or arguments) and returns a result.
- The contract says what kind of data the argument and result are.
- Purpose statement describes how the result depends on the argument.
- The design strategy is a short description of how to get from the purpose statement to the code.

Typical Program Design Strategies

Design Strategies
1. Combine simpler functions
2. Use template for <data def> on <vble>
3. Divide into cases on <condition>
4. Use HOF <mapfn> on <vble>
5. Call a more general function

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.

Programs are sets of Functions

The Function Design Recipe

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

Design Strategies

1. Combine simpler functions
2. Use template for <data def> on <vble>
3. Divide into cases on <condition>
4. Use HOF <mapfn> on <vble>
5. Call a more general function

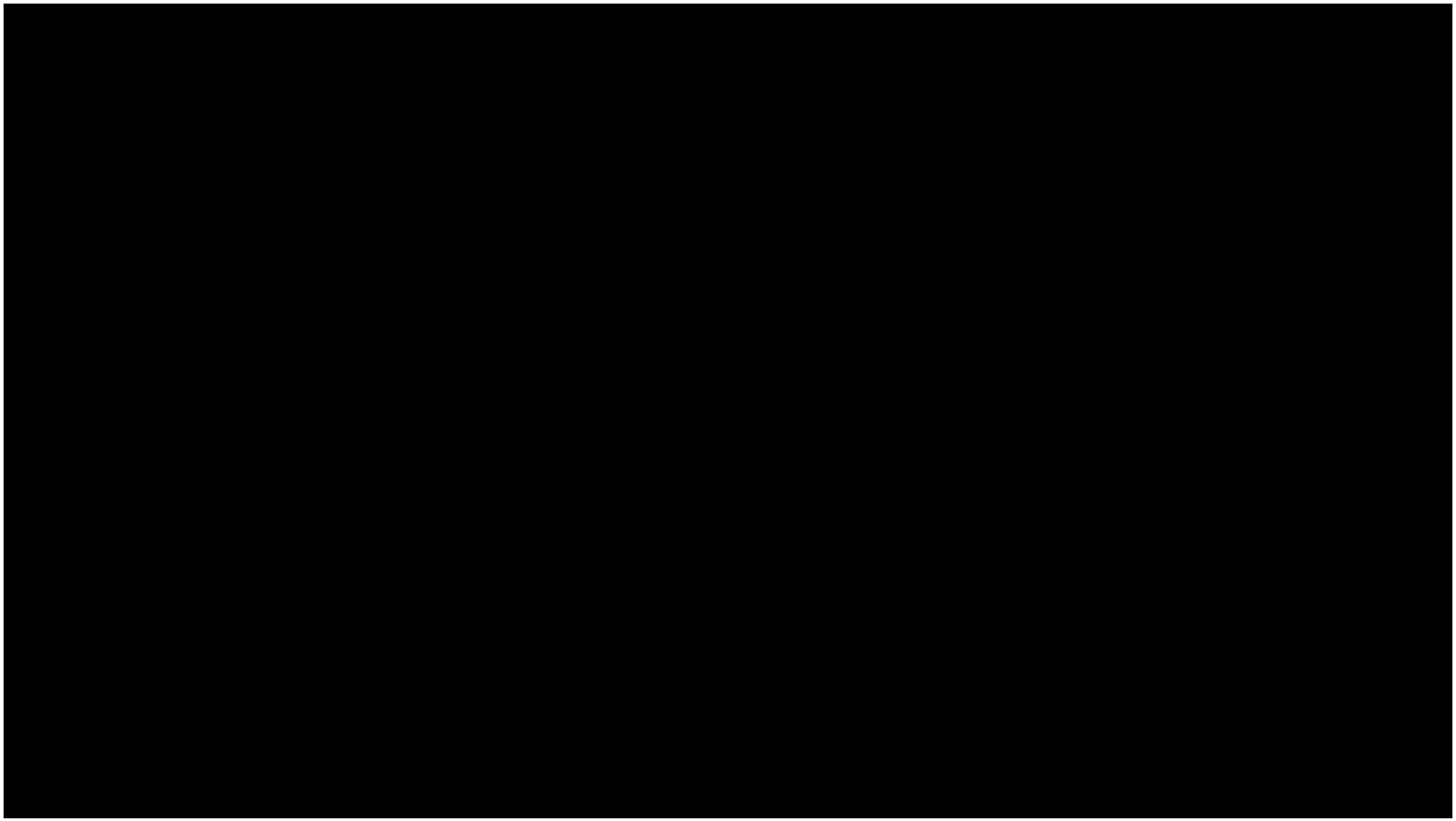
Design Strategy #1: Combine Simpler Functions

- Many times the desired function can be described as a combination of simpler functions.
- This is what we did for **f2c**, where the simpler computations were just arithmetic.

Demo: velocity.rkt

- On the next slide, you'll see a video of me defining a function using the strategy "Combine Simpler Functions".
- Observe how I followed the recipe: the contract, purpose statement, examples and tests were written *before* the function definition.
- Oops:
  - The contract should have said **Real**, not **Number**.
  - The strategy should be "combine simpler functions" (we used to call this "function composition" but we decided to change it to a less fancy name. ☺)
- The file is 01-4-velocity.rkt .

Demo: velocity.rkt



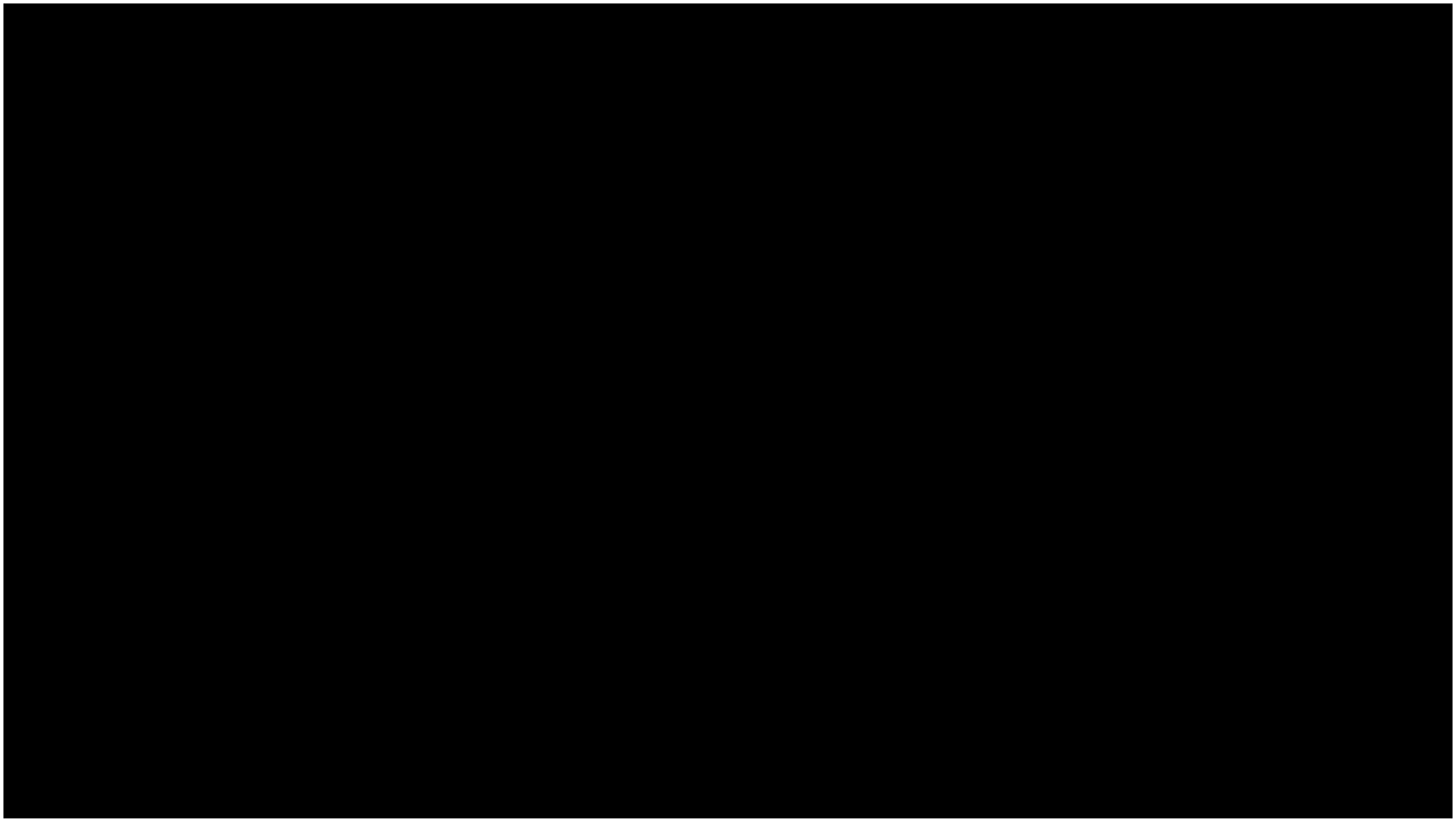
YouTube link

Note: you should never use Number when you mean Integer, NonNegInt, or Real. Here I should have used Real.

Another example: area-of-ring

- Sometimes the simpler functions may include ones you write yourself.
- Here's an example: area-of-ring, which calls area-of-circle.
- Both of these are defined by combining simpler functions.

Video: area-of-ring



YouTube link

I should have used Real (or NonNegReal) here, too.

What can you write in a combination of simpler functions?

- Remember that the goal is to write beautiful programs.
- You want your reader to understand what you're doing immediately.
- So just keep it simple.
- We won't have formal rules about this, but:
- If the TA needs you to explain it, it's not simple enough.
- Anything with an **if** is probably not simple enough.
  - If you need an **if**, that's a sign that you're using a fancier design strategy. We'll talk about these very soon.

Keep it short!

- “Combining simpler functions” is for very short definitions only.
- If you’re writing something complicated, that means one of two things:
  - You’re really using some more powerful design strategy (to be discussed)
  - Your function needs to be split into simpler parts.
    - If you have complicated stuff in your function you must have put it there for a reason. Turn it into a separate function so you can explain and test it.

When do you need to introduce new functions?

- If a function has pieces that can be given meaningful contracts and purpose statements, then break it up and use function composition.
- Then apply the design recipe to design the pieces.

Bad Example

```
;; ball-after-tick : Ball -> Ball
;; strategy: use template for Ball
(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: combine simpler functions
(define (ball-after-tick b)
  (if
    (ball-would-hit-wall? b)
    (ball-after-bounce b)
    (ball-after-straight-travel b)))
```

Here's a pair of examples.
Which do you think is clearer?
Which looks easier to debug?
Which would you like to have to defend in front of a TA?

Summary

- In this lesson, you've learned
  - How to use Function Composition to write a function definition.
  - When a function definition needs to be simplified by using help functions.
  - How to use Cases to partition a scalar data type.

Next Steps

- Study the files
- If you have questions or comments about this lesson, post them on the discussion board.