# Design Strategies

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 2.2

# Learning Objectives

- At the end of this lesson, the student should be able to recognize and use the design strategies:
  - transcribe formula
  - combine simpler functions
  - use template
  - divide into cases

# Review: 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 you got from the purpose statement to the code.

# Examples of Design Strategies

1. Transcribe formula
2. Combine simpler functions
3. Use the template for <data def> on <variable>
4. Divide into cases on <condition>

- A particular piece of code could be described by several different strategies.
- What's important is to write down a strategy that helps the reader understand the code

# Design Strategy: Transcribe formula

- Many times the desired function is just the evaluation of a mathematical formula
- This is what we did for **f2c**
- Another example: 02-2-1-velocity.rkt

# Design Strategy: combine simpler functions

- Sometimes the problem can be solved by composing two or more subproblems.
- Here’s an example: area-of-ring, which calls area-of-circle.
- We say the strategy for area-of-ring is “combine simpler functions”, and the strategy for area-of-circle is “transcribe formula”
- Read 02-2-2-area-of-ring.rkt and the commentary there.

# 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?

Do you think "combine simpler functions" is a good description of how this function works?

# Design Strategy: Use template

- We've already seen examples of using an observer template in Lesson 1.4, so we won't repeat that here.
- If we are returning a struct, sometimes it's more informative to say that we are using a constructor template.

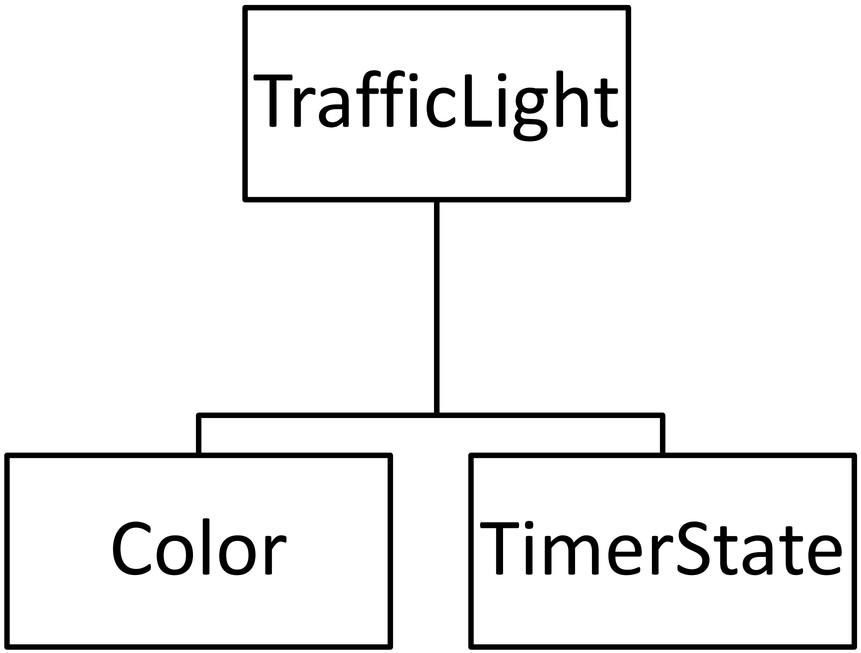
# Example of using a constructor template

```
;; A Traffic Light changes its color every 20 seconds, controlled by a
;; countdown timer.

;; A TrafficLight is represented as a struct
;;  (make-light color time-left)
;; with the fields
;; color : Color  represents the current
;;                color of the traffic light
;; time-left : TimerState   represents the
;;                current state of the timer

;; For the purposes of this example, we leave
;; Color and TimerState undefined.  For a
;; working example, we would have to define
;; these.

;; IMPLEMENTATION
(define-struct list (color time-left))

;; CONSTRUCTOR TEMPLATE
;; (make-light Color TimerState)

;; OBSERVER TEMPLATE (omitted)
```

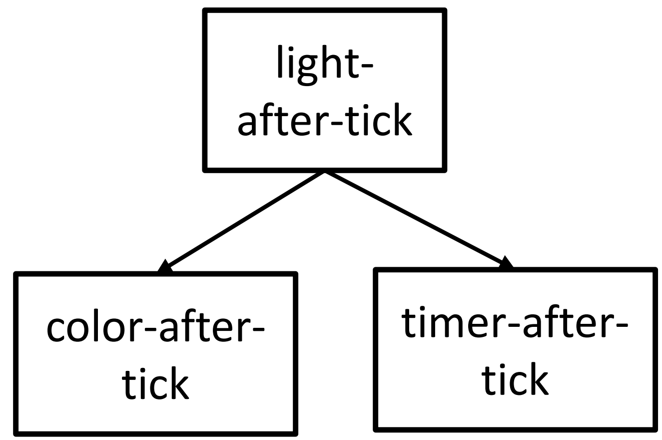
```
;; light-after-tick :
;;    TrafficLight -> TrafficLight
;; GIVEN: the state of a traffic light
;; RETURNS: the state of a traffic
;;    light after 1 second
;; EXAMPLES: (omitted)

;; DESIGN STRATEGY: Use constructor
;;   template for TrafficLight

(define (light-after-tick l)
  (make-light
   (color-after-tick l)
   (timer-after-tick l)))
```

Here we've divided the problem into 2 parts: finding the color after a tick and finding the timer state after a tick.
It would be OK to describe this as "combine simpler functions", but it's more informative to describe it as using the constructor template. This is also a very common pattern in our code.

# Remember: The Shape of the Program Follows the Shape of the Data




Data Hierarchy




Call Tree (the arrow goes from caller to callee)

# Design Strategy: Divide into cases

- Sometimes you need to break up an argument in some way other than by its template.
- We already saw this in Lesson 0.4 in the definition of **abs:**

```
; abs : Real -> Real
; RETURNS: the absolute value of the given real number.
; STRATEGY: divide into cases on sign of x
(define (abs x)
  (if (< x 0)
      (- 0 x)
      x))
```

# Example: income tax

- Imagine we are computing income tax in a system where there are three rates:
  - One on incomes less than \$10,000
  - One on incomes between \$10,000 and \$20,000
  - One on incomes of \$20,000 and over
- The natural thing to do is to partition the income into three cases, corresponding to these three income ranges.

# Write a **cond** or **if** that divides the data into the desired cases

```
;; STRATEGY: Cases on amt
;; f : NonNegReal -> ??
(define (f amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))     ...]
    [(and (<= 10000 amt) (< amt 20000)) ...]
    [(<= 20000 amt) ...]))
```

# Write a **cond** or **if** that divides the data into the desired cases

```
;; tax-on : NonNegInt -> NonNegInt
;; GIVEN: A person's income in USD
;; RETURNS: the tax on the income in USD
;; EXAMPLES: ....
;; STRATEGY: Cases on amt
(define (tax-on amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))     ...]
    [(and (<= 10000 amt) (< amt 20000)) ...]
    [(<= 20000 amt) ...]))
```

The predicates must be exhaustive. Make them mutually exclusive when you can.

# Now fill in the blanks

```
;; tax-on : NonNegReal -> NonNegReal
;; GIVEN: A person's income
;; RETURNS: the tax on the income
;; EXAMPLES: ....
;; STRATEGY: Cases on amt

(define (tax-on amt)
  (cond
    [(and (<= 0 amt) (< amt 10000))
      0]
    [(and (<= 10000 amt) (< amt 20000))
     (* 0.10 (- amt 10000))]
    [(<= 20000 amt)
     (+ 1000 (* 0.20 (- amt 20000)))]))
```

# Another example

```
;; ball-after-tick : Ball -> Ball
;; GIVEN: The state of a ball b
;; RETURNS: the state of the given ball at the next tick
;; STRATEGY: cases on whether ball would hit the wall on
;; the next tick

(define (ball-after-tick b)
  (if (ball-would-hit-wall? b)
    (ball-after-bounce b)
    (ball-after-straight-travel b)))
```

# Where does cases fit in our menu of design strategies?

- If you are inspecting a piece of enumeration or mixed data, you almost always want to use the template for that data type.
- Cases is mostly for when dividing up the data by the template doesn't work.

# Before we go…

- What should the contracts and purpose statements be for **ball-after-bounce** and **ball-after-straight-travel** ?
- It can't be

```
;; GIVEN: The state of a ball b
;; RETURNS: the state of the given ball at the next tick
```

- because then these would have to work for *any* ball.
- When these functions are called, we have additional information, and we need to document that information in these functions' contracts and purpose statements.

# These are better...

```
;; ball-after-bounce : Ball -> Ball
;; GIVEN: The state of a ball b that is going to bounce
;;        on the next tick
;; RETURNS: the state of the given ball at the next tick

;; ball-after-straight-travel : Ball -> Ball
;; GIVEN: The state of a ball b that will not bounce
;;        on the next tick
;; RETURNS: the state of the given ball at the next tick
```

# Summary

- We've now seen four Design Strategies:
  - Transcribe formula
  - Combine Simpler Functions
    - Combine simpler functions in series or pipeline
    - Use with any kind of data
  - Use Template
    - Used for enumeration , compound, or mixed data
    - Template gives sketch of function
    - Our most important tool
  - Cases
    - For when you need to divide data into case
      template doesn't fit.

Remember:
*The shape of the program follows the shape of the data.*

# Next Steps

- Study the example files
  - 02-2-1-velocity.rkt
  - 02-2-2-area-of-ring.rkt
  - 02-2-3-traffic-light-with-timer1.rkt
- If you have questions or comments about this lesson, post them on the discussion board.
- Go on to the next lesson