Designing Data

Lecture 4

16 September 2019
Lab 2

Late submissions
Solutions

Coaching hours

Office hours

Coding style

Assignment 1, out on Wednesday
Practice designing functions
Example: area
Problem

Design a function called `area` that consumes the length of one side of a square and produces the area of the square.
We're representing a square by the length of a side, as a number.

area : Number -> Number
Given a number produce a number

What’s wrong with this purpose?

It just repeats the signature. It’s not saying what the point of the function is.
;; We're representing a square by the length of a side, as a number.

;; area : Number -> Number
;; Given length of one side of a square, produce the area.
(define (area s) 0)

(check-expect (area 3) 3) ;;; 😃
(check-expect (area 3.2) (* 3.2 3.2))
;; We're representing a square by the length of a side, as a number.

;; area : Number -> Number
;; Given length of one side of a square, produce the area.

; Stub: (define (area s) 0)

(define (area s)
  (... s))

(check-expect (area 3) 3)
(check-expect (area 3.2) (* 3.2 3.2))
;; We're representing a square by the length of a side, as a number.

;; area : Number -> Number
;; Given length of one side of a square, produce the area.

;; Stub: (define (area s) 0)

;; Template:
;; (define (area s)
;;   (... s))

(define (area s)
  (* s s))

(check-expect (area 3) 3)
(check-expect (area 3.2) (* 3.2 3.2))
Run the tests:

Ran 2 tests.
1 of the 2 tests failed.

No signature violations.

Check failures:
   Actual value 9 differs from 3, the expected value.
   in area.rkt, line 12, column 0
We're representing a square by the length of a side, as a number.

area : Number -> Number
; Given length of one side of a square, produce the area.

Stub: (define (area s) 0)

Template:
; (define (area s)
;   (... s))

(define (area s)
  (* s s))

(check-expect (area 3) 9)
(check-expect (area 3.2) (* 3.2 3.2))
Example: tall
Problem

Design a function that consumes an image and determines whether the image is tall.
Since we’re asking a yes-or-no question, we know we want to return a Boolean.
When designing functions that produce a Boolean, the purpose should specify how to interpret the output. What does returning true mean?
The problem statement doesn’t say what to call the function, but we’re asking if an image is tall, and the convention for predicates – functions that return true or false – is to end with a question mark.

This is our stub (function header).
(require 2htdp/image)

;; tall? : Image → Boolean
;; Produce true if the image is tall

(define (tall? img) #false)

(check-expect (tall? (rectangle 2 3 "solid" "red")) #true)

How many tests does this function need? Is this enough?
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall

; Stub: (define (tall? img) #false)

(define (tall? img)
  (... img))

(check-expect (tall? (rectangle 2 3 "solid" "red"))
  #true)

We add our template, saying “we’ll need to do something with the input image”.
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall

; Stub: (define (tall? img) #false)

; Template:
; (define (tall? img)
;  (... img))

(define (tall? img)
  (if (... img)
      (... img)
      (... img)))

(check-expect (tall? (rectangle 2 3 "solid" "red"))
  #true)

Boolean functions have two cases – sometimes the image is tall, sometimes it’s not. So, we use if.
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall

; Stub: (define (tall? img) #false)

; Template:
; (define (tall? img)
;   (... img))

(define (tall? img)
  (if (> (image-height img) (image-width img))
    #true
    #false))

(check-expect (tall? (rectangle 2 3 "solid" "red")) #true)
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall

; Stub: (define (tall? img) #false)

; Template:
; (define (tall? img)
;   (... img))

(define (tall? img)
  (if (> (image-height img) (image-width img))
      #true
      #false))

(check-expect (tall? (rectangle 2 3 "solid" "red"))
              #true)
(check-expect (tall? (rectangle 3 2 "solid" "red"))
              #false)

Now we’re testing the #false case!
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image height is greater than the width

; Stub: (define (tall? img) #false)

; Template:
; (define (tall? img)
;    (... img))

(define (tall? img)
  (if (> (image-height img) (image-width img))
      #true
      #false))

(check-expect (tall? (rectangle 2 3 "solid" "red")) #true)
(check-expect (tall? (rectangle 3 2 "solid" "red")) #false)
(check-expect (tall? (rectangle 3 3 "solid" "red")) #false)

Let’s revise our purpose to be more precise.

We decide that square images aren’t tall.
Whenever we have an `if` statement that looks like this:

```
(if ⟨question⟩
    #true
    #false)
```

We should simplify it to

```
⟨question⟩
```

That is, the `if` statement adds nothing to the question; it already returns true if it’s true and returns false if it’s false!
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image height is greater than the width

; Stub: (define (tall? img) #false)

; Template:
; (define (tall? img)
;   (... img))

(define (tall? img)
  (> (image-height img) (image-width img)))

(check-expect (tall? (rectangle 2 3 "solid" "red")) #true)
(check-expect (tall? (rectangle 3 2 "solid" "red")) #false)
(check-expect (tall? (rectangle 3 3 "solid" "red")) #false)
Interlude: cond
Let’s define three rectangles:

```
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 20 20 "solid" "red"))
(define I3 (rectangle 20 10 "solid" "red"))
```

What if, instead of producing a Boolean to say these images are tall or not, we classify them as “tall”, “square”, or “wide”?
(require 2htdp/image)

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 20 20 "solid" "red"))
(define I3 (rectangle 20 10 "solid" "red"))

;; aspect-ratio : Image -> String
;; Produce shape of image, one of "tall", "square", or "wide"
(define (aspect-ratio img)
  (if (> (image-height img) (image-width img))
    "tall"
    (if (= (image-height img) (image-width img))
      "square"
      "wide")))

(check-expect (aspect-ratio I1) "tall")
(check-expect (aspect-ratio I2) "square")
(check-expect (aspect-ratio I3) "wide")
(require 2htdp/image)

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 20 20 "solid" "red"))
(define I3 (rectangle 20 10 "solid" "red"))

;; aspect-ratio : Image -> String
;; Produce shape of image, one of "tall", "square", or "wide"
(define (aspect-ratio img)
  (cond [(> (image-height img) (image-width img))
         "tall"]
        [(= (image-height img) (image-width img))
         "square"]
        [else "wide"]))

(check-expect (aspect-ratio I1) "tall")
(check-expect (aspect-ratio I2) "square")
(check-expect (aspect-ratio I3) "wide")
The **cond** expression has this form:

```
(cond  [⟨question 1⟩  ⟨answer 1⟩]
       [⟨question 2⟩  ⟨answer 2⟩]
       ...
       [else  ⟨default answer⟩])
```

All of the question expressions need to produce a Boolean value, true or false. The **else** keyword is just a question that’s always true.

It’s optional to use square brackets, but it makes it easier to see the structure than if you only using parentheses.
Evaluation rule for cond expressions:

1. If there are no question–answer pairs, signal an error.

2. If the first question is not a value, evaluate it and replace it with its value. That is, replace the entire cond with a new cond where the first question has been replaced by its value.

3. If the first question is #true or else, replace the entire cond expression with the first answer.

4. If the first question is false, drop the first question–answer pair; that is, replace the cond with a new cond that does not have the first question–answer pair.

5. Since the first question is not true or false, signal an error.
Trace the evaluation of this simple \texttt{cond} expression, either manually or with the Stepper:

\begin{verbatim}
(cond [(> 1 2) "bigger"]
  [(= 1 2) "equal"]
  [(< 1 2) "smaller"])
\end{verbatim}
Designing with cond

When the problem statement divides the input into several categories:

Write a test for each one.

Start the body with a cond expression with a case for each one.

Formulate a question to recognize each category.
Data definitions
Suppose you’re working on a program that someone else wrote for simulating traffic.

You see this function definition:

```
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))
```

It’s short – but not very clear.
If they’d followed the steps of the design recipe, it would be a little clearer:

;; next-color : Natural -> Natural
;; Produce next color of traffic light
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))

(check-expect (next-color 0) 2)
(check-expect (next-color 1) 0)
(check-expect (next-color 2) 1)

What does it mean to get a 2?
What about 3?
Problem domain

A light is red

Program

information

represent

data

interpret

... 0 ...

...
A *data definition* will describe how we are representing information as data.
;; Data definitions:

;; TLColor is one of:
;; 0
;; 1
;; 2
;; interp. color of a traffic light -- 0 is red, 1 yellow, 2 green

The template for a function is determined by the type of data it consumes. Given this data definition, we can write a template for functions that consume a TLColor:

```
(define (fn-for-tlcolor c)
  (cond [(= c 0) (...)]
        [(= c 1) (...)]
        [(= c 2) (...)]))
```
;; Data definitions:

;; TLColor is one of:
;;  - 0
;;  - 1
;;  - 2
;; interp. color of a traffic light -- 0 is red, 1 yellow,
;; 2 green

;; Template for TLColor:
;; (define (fn-for-tlcolor c)
;;  (cond [(= c 0) (...)]
;;     [(= c 1) (...)]
;;     [(= c 2) (...)]))

;; Functions:

;; next-color : TLColor -> TLColor
;; Produce next color of traffic light
(define (next-color c)
  (cond [(= c 0) 2]
        [(= c 1) 0]
        [(= c 2) 1]))

(check-expect (next-color 0) 2)
(check-expect (next-color 1) 0)
(check-expect (next-color 2) 1)
Designing data: examples
Problem

Imagine you’re designing a program that, among other things, has information about the names of cities in its problem domain.

Design a data definition to represent the name of a city.
;; CityName is String
;; interp. the name of a city

(define CN1 "Boston")
(define CN2 "Vancouver")

;; Template for one CityName:
;; (define (fn-for-city-name cn)
;;   (\ldots cn))
Problem

Use the CityName data definition to design a function that produces true if the given city is the largest city in New York State.
;; largest-in-nys? : CityName -> Boolean
;; Produce true if the given city is the largest in
;; New York State
(define (largest-in-nys? cn) #false)
;; largest-in-nys? : CityName → Boolean
;; Produce true if the given city is the largest in
;; New York State
(define (largest-in-nys? cn) #false)

(check-expect (largest-in-nys? CN1) #false)
(check-expect (largest-in-nys? "New York") #true)
;; largest-in-nys? : CityName -> Boolean
;; Produce true if the given city is the largest in
;; New York State

; Stub: (define (largest-in-nys? cn) #false)

; CityName function template:
(define (largest-in-nys? cn)
  (... cn))

(check-expect (largest-in-nys? CN1) #false)
(check-expect (largest-in-nys? "New York") #true)
;;;; largest-in-nys? : CityName -> Boolean
;;;; Produce true if the given city is the largest in
;;;; New York State

;; Stub: (define (largest-in-nys? cn) #false)

;; CityName function template:
;; (define (largest-in-nys? cn)
;;   (... cn))

(define (largest-in-nys? cn)
  (string=? "New York" cn))

(check-expect (largest-in-nys? CN1) #false)
(check-expect (largest-in-nys? "New York") #true)
;; largest-in-nys? : CityName → Boolean
;; Produce true if the given city is the largest in
;; New York State

; Stub: (define (largest-in-nys? cn) #false)

; CityName function template:
; (define (largest-in-nys? cn)
;   (... cn))

(define (largest-in-nys? cn)
  (or (string=? "New York" cn)
      (or (string=? "New York City" cn)
          (string=? "NYC" cn))))

(check-expect (largest-in-nys? CN1) #false)
(check-expect (largest-in-nys? "New York") #true)
(check-expect (largest-in-nys? "New York City") #true)
(check-expect (largest-in-nys? "NYC") #true)
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi