The Design Recipe

15 September 2020
Review: Lab 2 solutions
Following a *design recipe* systematizes the design of a function.

It helps ensure we end up with a high-quality function that does what we need and has been well-tested.
Design recipes will make hard problems much easier – but they also make easy problems a little slower to design.

We’ll use the recipe on easy problems just to learn the recipe.

Hang in there!
How to design data and functions
Data

Data definition

Functions
Data

Data definition

Interpretation

Functions
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td></td>
</tr>
<tr>
<td>Interpretation</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>Template</td>
<td></td>
</tr>
</tbody>
</table>
Data definition
Interpretation
Examples
Template

Functions
; A SeasonNumber is a whole number
; in [1,8]
Data definition

Interpretation

Examples

Template
A SeasonNumber is a whole number in [1,8]

Interpretation: A season number in the Game of Thrones series.
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td></td>
</tr>
<tr>
<td>Interpretation</td>
<td></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td></td>
</tr>
<tr>
<td>Template</td>
<td></td>
</tr>
</tbody>
</table>
A SeasonNumber is a whole number in [1,8]

Interpretation: A season number in the Game of Thrones series.

Examples: 1, 7, 8
A SeasonNumber is a whole number in [1,8]

Interpretation: A season number in the Game of Thrones series.

Examples: 1, 7, 8
(define SEASON-ONE 1)
(define SEASON-SEVEN 7)
(define SEASON-EIGHT 8)
Data definition
Interpretation
Examples
Template

Functions
A SeasonNumber is a whole number in [1,8]

Interpretation: A season number in the Game of Thrones series.

Examples: 1, 7, 8
(define SEASON-ONE 1)
(define SEASON-SEVEN 7)
(define SEASON-EIGHT 8)

(define (fun-for-temp sn) (... sn))
Data

Data definition
Interpretation
Examples
Template

Functions
Data

- Data definition
- Interpretation
- Examples
- Template

Functions

- Signature
- Purpose
Data

Data definition
Interpretation
Examples
Template

Functions

Signature
Purpose
Tests
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
<tr>
<td>Data</td>
<td>Functions</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
;; is-finale?: Number, Number → Boolean
;; is-finale? : Number, Number → Boolean
;; is-finale? : SeasonNumber, EpNumber
;; → Boolean
is-finale? : SeasonNumber, EpNumber
-> Boolean
Do the given season/episode number constitute a GoT season finale?
;; is–finale? : SeasonNumber, EpNumber
;;   -> Boolean
;; Do the given season/episode number
;; constitute a GoT season finale?
(check-expect (is–finale? 6 10) #true)
(check-expect (is–finale? 6 9) #false)
(check-expect (is–finale? 7 8) #false)
(check-expect (is–finale? 7 7) #true)
(check-expect (is–finale? 8 6) #true)
;; is-finales? : SeasonNumber, EpNumber
;;      -> Boolean
;; Do the given season/episode number
;; constitute a GoT season finale?
(check-expect (is-finales? 6 10) #true)
(check-expect (is-finales? 6 9) #false)
(check-expect (is-finales? 7 8) #false)
(check-expect (is-finales? 7 7) #true)
(check-expect (is-finales? 8 6) #true)
;; is–finale? : SeasonNumber, EpNumber -> Boolean
;; Do the given season/episode number constitute
;; a GoT season finale?
(check-expect (is-finale? SEASON–ONE EP–ONE) #false)
(check-expect (is-finale? SEASON–ONE EP–TEN) #true)
(check-expect (is-finale? SEASON–SEVEN EP–ONE) #false)
(check-expect (is-finale? SEASON–SEVEN EP–SEVEN) #true)
(check-expect (is-finale? SEASON–EIGHT EP–SIX) #true)
Data

- Data definition
- Interpretation
- Examples
- Template

Functions

- Signature
- Purpose
- Tests
- Code
;; is-finale? : SeasonNumber, EpNumber -> Boolean
;; Do the given season/episode number constitute
;; a GoT season finale?
(check-expect (is-finale? SEASON-ONE EP-ONE) #false)
(check-expect (is-finale? SEASON-ONE EP-TEN) #true)
(check-expect (is-finale? SEASON-SEVEN EP-ONE) #false)
(check-expect (is-finale? SEASON-SEVEN EP-SEVEN) #true)
(check-expect (is-finale? SEASON-EIGHT EP-SIX) #true)
(define (is-finale? season ep)
  (or (and (< season SEASON-SEVEN)
            (= ep EP-TEN))
       (and (= season SEASON-SEVEN)
            (= ep EP-SEVEN))
       (and (= season SEASON-EIGHT)
            (= ep EP-SIX)))))
<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition</td>
<td>Signature</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Purpose</td>
</tr>
<tr>
<td>Examples</td>
<td>Tests</td>
</tr>
<tr>
<td>Template</td>
<td>Code</td>
</tr>
</tbody>
</table>
Design recipe details
Step 1: Data

Choose a representation suitable for the function’s input data.

- Fahrenheit degrees? **Number**
- Grocery items? **String**
- Faces? **Image**
- Wages? **Number**

In the case of **double**, we’re representing numbers as ... numbers.
Step 2: Signature

Describes the input(s) and output data.

We write the signature in the form

\[
\langle \text{Function name} \rangle : \langle \text{Type} \rangle \ldots \rightarrow \langle \text{Type} \rangle
\]
Step 2: Signature

Hand-in artifact: A comment like this:

```plaintext
;; double : Number -> Number
or
;; cat? : Image -> Boolean
```
Step 3: Purpose

Describes in English what the function will do, e.g.,

- Converts Fahrenheit degrees $f$ to Celsius-degrees, or
- Checks whether $i$ is an image of a cat

If it’s clearer, use the parameters names you give in the header rather than write “the given whatever”.

Step 3: Purpose

Hand-in artifact: A comment after the signature like this:

```
;; double : Number -> Number
;; Produce 2 times the given number
```
Step 4: Tests

Show examples of what will happen when the function is done.

Wrapping the examples in `check-expect` statements means they will also serve as unit tests for the completed function, e.g.,

```
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
```
Step 4: Tests

Hand-in artifact: One or more check-expect statements after the header:

;; double : Number -> Number
;; Produce 2 times the given number
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
Step 5: Code

How to draw an owl

1. Draw some circles
2. Draw the rest of the fucking owl
Step 5: Code

We start with the template. We’ll see more interesting ones later; for now it’s just the parameters, e.g.,

```
(define (double n)
  (... n))
```
Step 5: Code

Use everything written before to know how to complete the function body.

Sometimes it helps to elaborate the examples to show how the expected value could have been produced.
Step 5: Code

Hand-in artifact: A complete function:

`; double : Number -> Number
`; Produce 2 times the given number
(check-expect (double 3) 6)
(check-expect (double 4.2) 8.4)
(define (double n)
  (* 2 n))
(Step 6: Debugging!)

Click *Run*.

Did the tests work? 🎉

If not, there are three possibilities to consider:

1. You gave the wrong expected answer in one of the test cases.
2. The function definition computes the wrong result – it’s a 🐞!
3. Both the examples *and* the function definition are wrong.
As programs become more complex in the next few weeks, the design recipe will prove more helpful.

If you don’t learn to use the recipe now, you’ll be stuck having to learn both the recipe and other concepts later on.
Design recipe examples
Example: yell
Problem

Design a function called **yell** that takes as input strings like "Hello" and adds "!" to produce strings like "Hello!".
;; We're working directly with strings; no data definition!

;; yell : String -> String
;; Add "!" to the end of s
(check-expect (yell "Hello") "Hello!")
(check-expect (yell "Bye!") "Bye!!")
(define (yell s)
  (string-append s "!"))
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.
;; A Square is a natural number
;; Interpretation: the length of the side of a square in
;; pixels
(define SQUARE1 10)
(define SQUARE1 153)

;; area : Square -> Natural
;; Given a square, produce a number
;; A Square is a natural number
;; Interpretation: the length of the side of a square in pixels
(defun SQUARE1 10)
(defun SQUARE1 153)

;; area : Square -> Natural
;; Given a square, produce a number

What's wrong with this purpose?
;; A Square is a natural number
;; Interpretation: the length of the side of a square in pixels
(define SQUARE1 10)
(define SQUARE1 153)

;; area : Square -> Natural
;; Given a square, 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.
;; A Square is a natural number
;; Interpretation: the length of the side of a square in
;; pixels
(define SQUARE1 10)
(define SQUARE1 153)

;; area : Square \to Natural
;; Given the length of one side of a square, produce
;; the area.
(check-expect (area 3) 3) ;; 😊
(check-expect (area 10) (* 10 10))
;; A Square is a natural number
;; Interpretation: the length of the side of a square in pixels
(define SQUARE1 10)
(define SQUARE1 153)

;; area : Square -> Natural
;; Given the length of one side of a square, produce the area.
(check-expect (area 3) 3) ;; 😊
(check-expect (area 10) (* 10 10))
(define (area s)
  (... s))
A Square is a natural number
Interpretation: the length of the side of a square in pixels
(define SQUARE1 10)
(define SQUARE1 153)

area : Square -> Natural
Given the length of one side of a square, produce the area.
(check-expect (area 3) 3) ; ; 😄
(check-expect (area 10) (* 10 10))
(define (area s)
  (* s s))
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

 😫
;; A Square is a natural number
;; Interpretation: the length of the side of a square in pixels
(define SQUARE1 10)
(define SQUARE1 153)

;; area : Square \rightarrow Natural
;; Given the length of one side of a square, produce the area.
(check-expect (area 3) (* 3 3))
(check-expect (area 10) (* 10 10))
(define (area s)
  (* s s))
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?
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall
(check-expect (tall? (rectangle 2 3 "solid" "red"))
  #true)

How many tests does this function need? Is this enough?
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.
;; tall? : Image -> Boolean
;; Produce true if the image is tall
(check-expect (tall? (rectangle 2 3 "solid" "red"))
  #true)

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

This case is never executed!
(require 2htdp/image)

;; tall? : Image -> Boolean
;; Produce true if the image is tall
(check-expect (tall? (rectangle 2 3 "solid" "red")) #true)
(check-expect (tall? (rectangle 3 2 "solid" "red")) #false)
(define (tall? img)
  (if (> (image-height img) (image-width img)) #true #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
;; than the width
(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)

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

Let's revise our purpose to be more precise.

We decide that square images aren't tall.
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith
Laney Strange