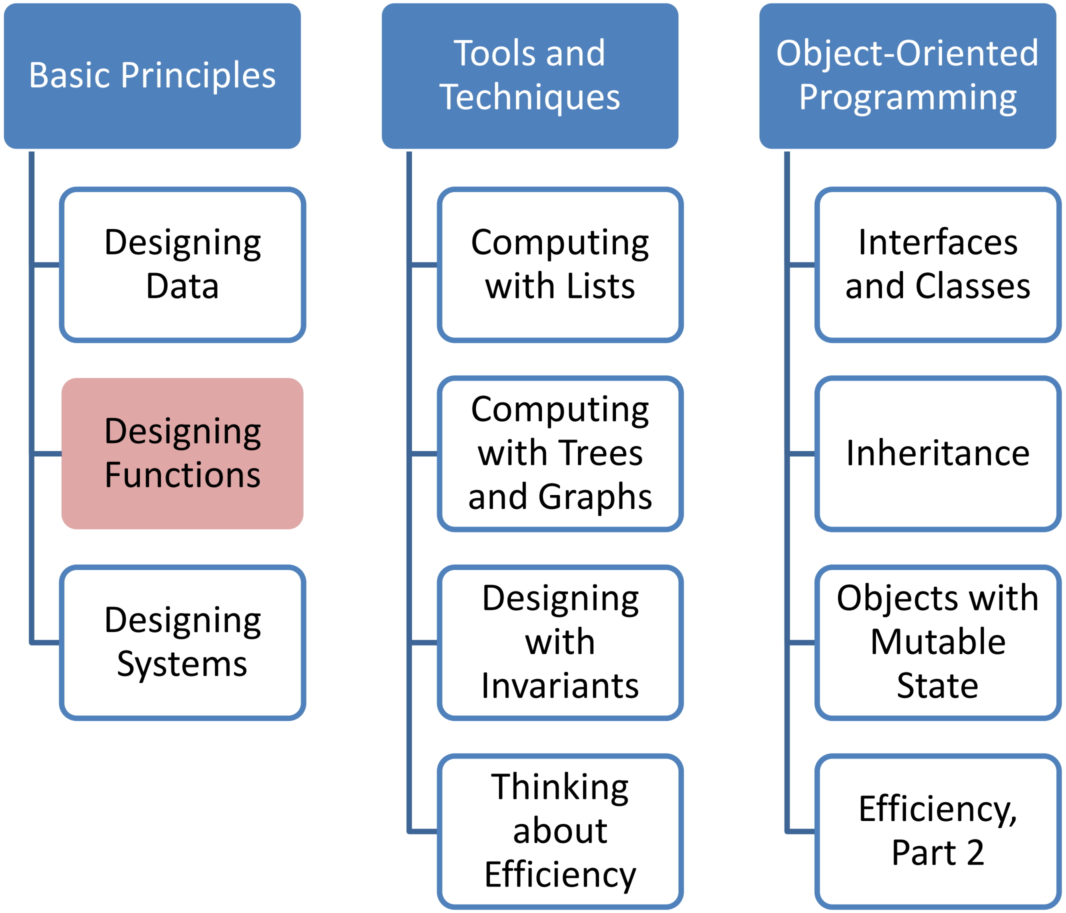
Contracts, Purpose Statements, Examples and Tests

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 2.1

© Mitchell Wand, 2012-2017
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Module 02



Basic Principles

- Designing Data
- Designing Functions
- Designing Systems

Tools and Techniques

- Computing with Lists
- Computing with Trees and Graphs
- Designing with Invariants
- Thinking about Efficiency

Object-Oriented Programming

- Interfaces and Classes
- Inheritance
- Objects with Mutable State
- Efficiency, Part 2

Module Objectives

- Last week, we introduced the Function Design Recipe, and examined the first step, Data Design, in detail.
- This week we will talk in more detail about the rest of the steps in the Function Design Recipe.
- We will also talk about the kinds of bugs you might encounter while running your programs and how to fix them.

Lesson Objectives

At the end of this lesson, students will be able to:

- Write a contract and purpose statements for simple functions.
- Provide examples showing sample arguments and intended results.
- Write down the examples as human readable comments within the program.

Lesson Outline

In this lesson we'll talk about two more steps in the Design Recipe:

- Step 2: Contract and Purpose Statement
- Step 3: Examples and Tests

We'll also talk about a few other things, like how to choose good names for your functions and variables.

The Function Design Recipe

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

FDR Step 2: Contract and Purpose Statement

- *Contract:* specifies the kind of input data and the kind of output data
- *Purpose Statement:* A set of short noun phrases describing *what* the function is supposed to return. These are typically phrased in terms of information, not data.
  - They generally take the form GIVEN/RETURNS, where each of these keywords is followed by a short noun phrase.
  - When possible, they are phrased in terms of information, not data.

Examples of Contract and Purpose Statements

```
;; f2c: FarenTemp -> CelsiusTemp
;; GIVEN: a temperature in Fahrenheit,
;; RETURNS: the equivalent temperature in
;;   Celsius

;; f2mars : FarenTemp -> CelsiusTemp
;; GIVEN: Any temperature in Fahrenheit
;; RETURNS: The mean temperature on the surface
;;   of Mars, in Celsius
```

Examples of Contract and Purpose Statements (2)

```
scene-with-cat : Cat Scene -> Scene
GIVEN: a Cat c and a Scene s
RETURNS: A Scene like s, except that the Cat c
has been painted on it.
```

What makes a good purpose statement?

- It gives more information than just the contract. For example

  ```
  GIVEN: an Integer and a Boolean
  RETURNS: an Integer
  ```

  is not a good purpose statement

- It is *specific*. Ideally, a reader should be able to figure out what a function returns just by reading the purpose statement
  - perhaps along with examples, other documentation, etc.
  - but WITHOUT reading the code!

Good Function Names are Important

- A good choice of function name is important.
- When a function is used in some other piece of code, the reader should be able to tell roughly what a function computes just by looking at its name.
- If further detail is needed, then the reader can refer to the purpose statement of the function.
- If the function name is chosen well and the purpose statement is written well, the reader should rarely, if ever, need to refer to the function definition.

For more discussion, see What's in a Name?

Conventions for Good Function Names

- Function names should almost always be nouns
- Should describe the result of the function
  - e.g. **`area`**, not **`compute-area`**
- Predicates should end in **`?`** : e.g., **`square?`**

  (pronounced "huh?", as in "square-huh?")
- Use first component of the name to distinguish similar functions with different arguments, e.g.:
  - **`circle-area, ring-area`**
  - **`book-price, total-order-price`**

Conventions for Good Names

- In Racket, "-" and "?" are legal characters that may occur in names.
- Use the minus sign to separate components of a name, e.g. **`total-order-price`**
- Use the question mark to name predicates: eg, **`square?`** .
- These are our conventions. Other languages have other conventions; you should follow them.

Argument Names

- We use short names for arguments:
  - **b** for a **Book**
- Or mnemonic names:
  - **`cost`**, **`price`**
- Qualified names:
  - **`mouse-x`**, **`bomb-x`**
- Avoid lame names, like **list1** . Names should refer to the information, not just the data type, whenever possible.
- These are our conventions. Your workplace may have different conventions for argument names.

Numeric Data Types

- In Racket, Number includes Complex numbers, so we'll hardly ever use Number.
- **Integer** vs. **NonNegReal** vs. **PosReal** ?
  - look to the data definition. If your number represents a quantity that is always non-negative (say, a length or an area), then call it a **NonNegInt**.
  - if we're not dealing with physical quantities, then we'll typically use **Integer**.
  - *Your function has to handle any value of the type it says in the contract.*

FDR Step 3: Examples and Tests

- Examples show sample arguments and results, to make clear what is intended.
- This may include showing how the function should be called.
- It should also illustrate the different behaviors of the function.
- How many examples, and what kind, will depend a lot on the function

Examples of Examples (1)

- If the function is a linear function of a single input, two examples are sufficient to uniquely determine the function.
- We saw this for **`f2c`** :

```
;; (f2c 32) = 0
;; (f2c 212) = 100
```

Examples of Examples (2)

- If the function takes an argument that is itemization or mixed data, then choose examples from each subclass of the itemization.
- Example:

```
;; (next-state "red") = "green"
;; (next-state "yellow") = "red"
;; (next-state "green") = "yellow"
```

- If your function uses a cond to divide its inputs into classes, choose examples from each class.

Examples of Examples (3)

- Avoid coincidences in your examples.
- This example is coincidental:

```
(book-profit-margin
 (make-book "Little Lisper" "Friedman" 2.00 4.00))
= 2.00
```

  - Is the answer 2 because we subtracted 2 from 4, or because it is the third field in the book?

- This example is not coincidental:

```
(book-profit-margin
 (make-book "Little Lisper" "Friedman" 2.00 5.00))
= 3.00
```

  - we must have subtracted 2 from 5 to get 3.

Make your examples readable

```
;;; Here’s an example: a rocket simulation.
;; INFORMATION ANALYSIS:

;; An Altitude is represented as a Real, measured in meters

;; A Velocity is represented as Real, measured in meters/sec upward

;; We have a single rocket, which is at some altitude and is
;; travelling vertically at some velocity.

;; REPRESENTATION:
;; A Rocket is represented as a struct (make-rocket altitude velocity)
;; with the following fields:
;; altitude : Altitude  is the rocket's altitude
;; velocity : Velocity  is the rocket's velocity

;; IMPLEMENTATION:
(define-struct rocket (altitude velocity))

;; CONSTRUCTOR TEMPLATE:
;; (make-rocket Real Real)
```

Not-so-readable examples

```
;; EXAMPLE:
;; (rocket-after-dt (make-rocket 100 30) 0)
;;  = (make-rocket 100 30)
;; (rocket-after-dt (make-rocket 100 30) 2)
;;  = (make-rocket 160 30)
```

- What do these examples illustrate? Where did those values come from?
- These are very simple structures, but for more complicated structures you'd have a hard time telling.
  - and so would your grader, or boss!
- And if you change the representation of rockets, you'll have to change all your examples, too!

Better Examples

```
(define rocket-at-100 (make-rocket 100 30))
(define rocket-at-160 (make-rocket 160 30))

;; (rocket-after-dt rocket-at-100 0) = rocket-at-100
;; (rocket-after-dt rocket-at-100 2) = rocket-at-160
```

- Here we’ve introduced mnemonic names for each of the example values. These could serve as examples for the data definitions, too.
- You can inspect those definitions to check whether they represent the rocket they are supposed to represent.
- The example is in terms of information, not data.
- If you decide later to change the representation, you can still use the examples.

Turn your examples into tests

```
(begin-for-test
  (check-equal? (f2c 32) 0)
  (check-equal? (f2c 212) 100))
```

- Tests live in your file, so they are checked every time your file is loaded
- Exact technology for tests may change; see the example files for current technology
- LOTS more to say about testing, but this is enough for now.

Summary

- In this lesson, you have learned how to:
  - Write a contract and purpose statements for simple functions.
  - Provide examples showing sample arguments and intended results.
  - Write down those examples as human readable comments within the program.
  - Turn your examples into executable tests.

Next Steps

- Study the file 02-1-1-rocket-examples.rkt in the Examples folder.
- If you have questions about this lesson, post them on the discussion board.
- Go on to the next lesson.