

CS450_(section 2)

High Level Languages

UMass Boston Computer Science

Monday, September 16, 2024

Logistics

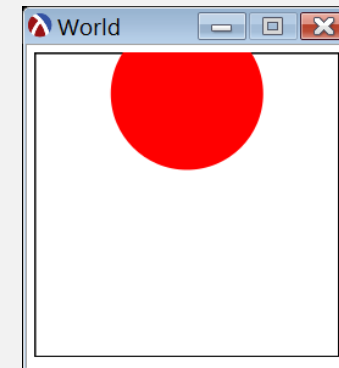
- HW 1 in
 - ~~due: Mon 9/16 12pm (noon) EST~~
- HW 2 out
 - due: Mon 9/23 12pm (noon) EST
- Do not send hw questions by email! (I may not see it)
 - Post to piazza (use private or anonymous if unsure) (I may change)
 - Makes it easier for staff to check one place
- **“Why is the autograder erroring?”**
 - Ask for help before you get to this point!
 - Must test code independently of gradescope
 - Don't submit until HW is complete (autograder error = HW is not complete)
- Course web site:
 - Added Design Recipe section
 - Lecture code (see lecture03.rkt) may occasionally be posted

Design Recipe Intro: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has 3 parts:
 1. A defined **Name**
 2. Description of **all possible values** of the data
 3. An **Interpretation** explains the real world concepts the data represents

```
;; A WorldState is a Non-negative Integer  
;; Interp: Represents the y Coordinate of the center of a  
;;         ball in a `big-bang` animation.
```



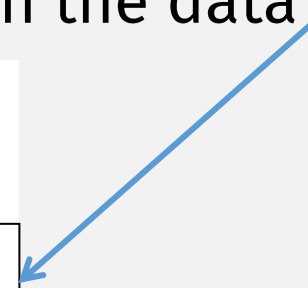
Design Recipe, Step 1: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has **3-4** parts:
 1. A defined **Name**
 2. Description of **all possible values** of the data
 3. An **Interpretation** explains the real world concepts the data represents
 - ➔ 4. A **predicate** returns true if a given value is in the data definition

```
;; A WorldState is a Non-negative Integer  
;; Interp: Represents the y Coordinate of the center of a  
;;          ball in a `big-bang` animation.
```

```
(define (WorldState? x)  
  (exact-nonnegative-integer? x))
```



Design Recipe

- 1. Data Design**
- 2. Function Design**

*Last
Time*


Designing Functions

1. **Name**
2. **Signature**
3. **Description**
4. **Examples**
5. **Code**
6. **Tests**

Designing Functions

1. **Name**
2. **Signature** – types of the function input(s) and output
 - Use Data Definitions (or **create new data defs**, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works

Designing Functions

1. **Name** 

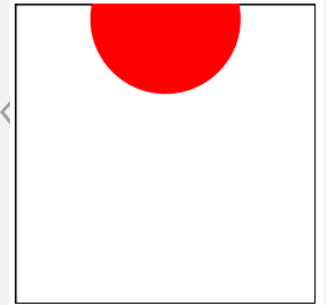
```
;; render: WorldState -> Image  
;; Draws a WorldState as a 2htdp/image Image
```
2. **Signature** – types of the function input(s) and output
 - Use Data Definitions (or **create new data defs**, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works

FAQ: What about “error-checking”?

“Error handling is important, but if it obscures logic, it’s wrong.”
— **Robert C. Martin**, Clean Code: A Handbook of Agile Software Craftsmanship

Designing Functions

1. **Name** `;; render: WorldState -> Image`
`;; Draws a WorldState as a 2htdp/image Image`
2. **Signature** – types of the function input(s) and output
 - Use Data Definitions (or create new data defs, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
 - (put with function definition)
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works



```
(define (render w)
  (place-image
   BALL-IMG
   BALL-X w
   EMPTY-SCENE))
```



```
(check-equal?
 (render INITIAL-WORLDSTATE)
 (place-image
  BALL-IMG
  BALL-X INITIAL-WORLDSTATE
  EMPTY-SCENE))
```

Examples come before (and help to write) Code!

FAQ: What about “error-checking”


This declares that the function cannot be given a non-WorldState argument!

Designing Functions

... but we can make it more robust

1. **Name** `;; render: WorldState -> Image`
`;; Draws a WorldState as a 2htdp/image Image`
2. **Signature** – types of the function input(s) and output
 - Use Data Definitions (or create new data defs, if needed)

The Signature is error-checking

3. **Description** – explain (in English) how the function works
`> (render "bad arg")`
 `place-image: expects a real number as third argument, given "bad arg"`

It's the user's fault if they call the function incorrectly

4. **Examples** – show (using rackunit) how the function works
BUT: This is a bad error message because ...

5. **Code** – implement how the function works
... it reveals internal details that the user doesn't (and shouldn't have to) know about

6. **Tests** – check (using rackunit) that the function works

More Robust Signatures

1. **Name** `;; render: WorldState -> Image`
`;; Draws a WorldState as an Image`

2. **Signature** – types of the function inputs and outputs

- Use Data Definitions (or create new data definitions)
- Use define/contract and predicates!

3. **Description** – explain (in English)

It can be used no matter what language you're programming in

4. `> (render "bad arg")`
`render: contract violation`
`expected: WorldState?`
`given: "bad arg"`
5. `in: the 1st argument of`
`(-> WorldState? image?)`
`contract from: (function render)`

Function contract

Good error message:
precise, and no
internal details!

```
(define/contract (render w)
  (-> WorldState? image?)
  (place-image
   BALL-IMG
   BALL-X w
   EMPTY-SCENE))
```

NOTE:

Different languages have different “signature” or “error handling” mechanisms

- Contracts
- Types
- Asserts
- Try-Catch-Throw

But the **Design Recipe** is
language-agnostic

6.

Designing Functions

1. **Name**
2. **Signature** – types of the function input(s) and output
 - Use Data Definitions (or create new data defs, if needed)
 - Use define/contract and predicates!
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works
 - put in separate test-suite (file)

In-class Office Hours

- Get HW1 working
- Add `tests.rkt` with test-suite named TESTS to HW1
 - 2 per function
 - I might run against other submissions and award bonus pts
- Start HW2