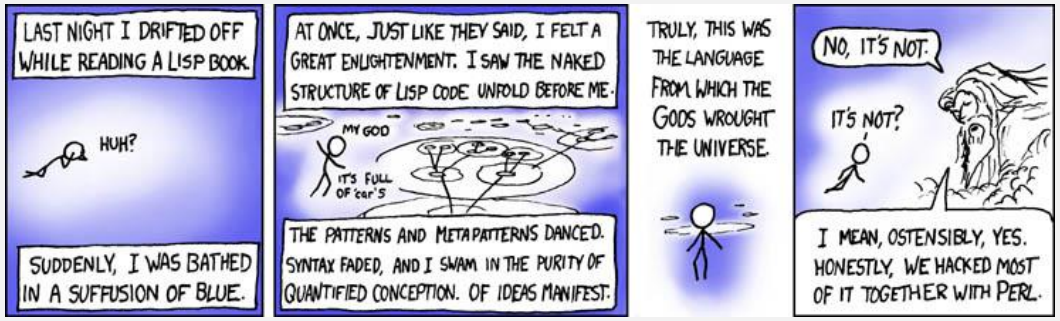
CS450 (section 2)

High Level Languages

UMass Boston Computer Science

Monday, September 18, 2023

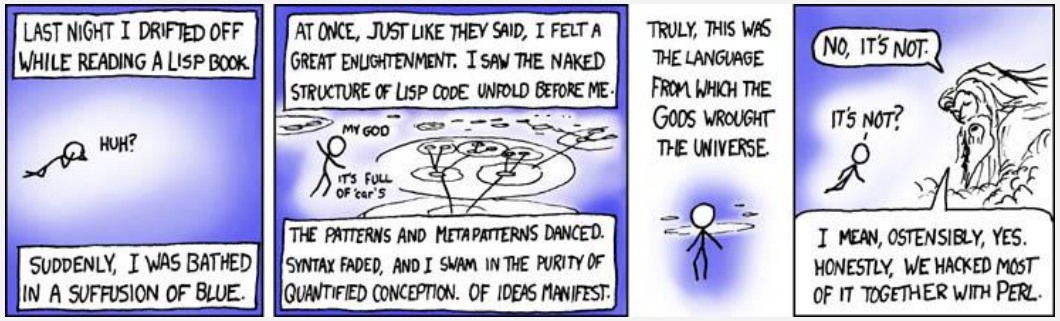


Logistics

- HW 0 in
  - ~~due: Sun 9/17 11:59 pm EST~~
- HW 1 out
  - due: Sun 9/24 11:59 pm EST

- Do not send hw questions by email! (I wont see it)
  - Post to piazza (use private or anonymous if unsure) (I may change)
  - Makes it easier for me 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



- Course web site:
  - Added Design Recipe section
  - Lecture code (see lecture03.rkt) may occasionally be posted

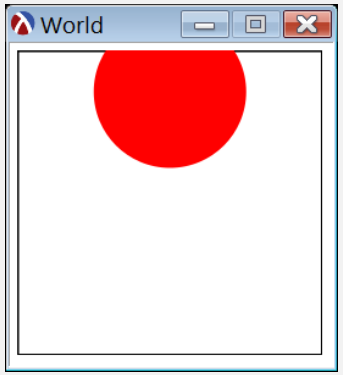
Last Time

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.
```



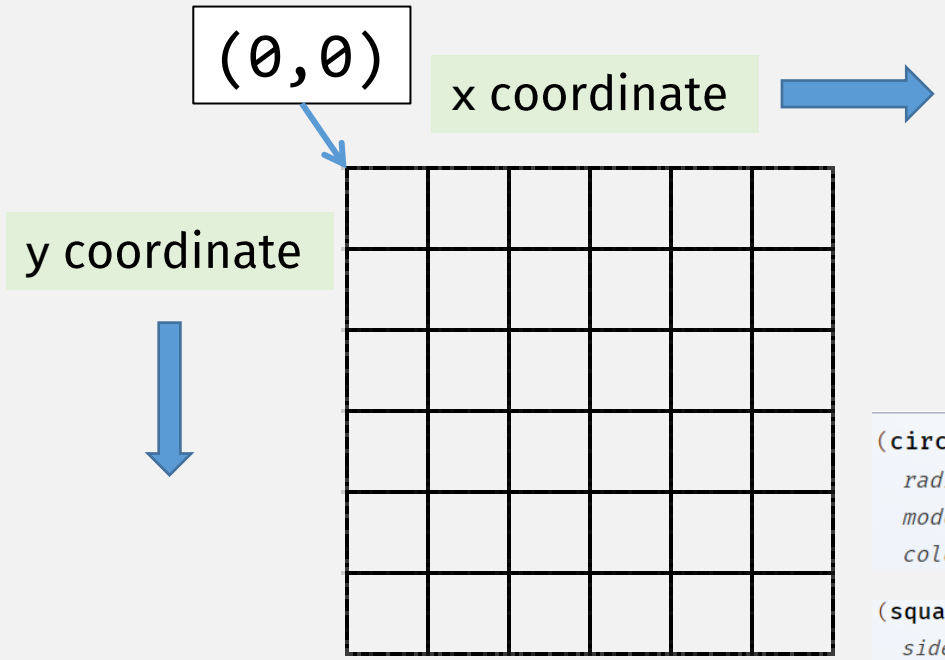
Design Recipe, Step 1: Data Design

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))
```

Interlude: htdp universe coordinates




```
(place-image image x y scene) → image?                    procedure
  image : image?
  x : real?
  y : real?
  scene : image?
```

Places *image* onto *scene* with its center at the coordinates (*x*,*y*) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

```
(circle radius mode color) → image?
  radius : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?

(square side-len mode color) → image?
  side-len : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?
```

```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???

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

```
;; render: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
```

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

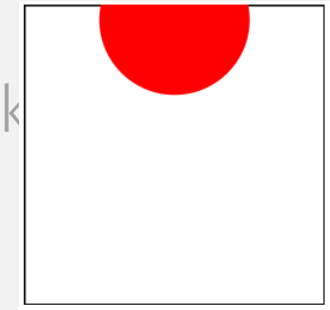
Designing Functions

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 work



4. **Examples** – show (using `rackunit`) how the function works
   - (put with function definition)

```
(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))
```

5. **Code** – implement how the functi

6. **Tests** – check (using rac

FAQ: What about "error-checking"?

Examples come before (and **help to write**) **Code!**

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)

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"
```

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

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

… but we can make it more robust

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

BUT: This is a bad error message because …

… it reveals internal details that the user doesn't (and shouldn't have to) know about

FAQ: What about "error-checking"?

More Robust Signature

1. Name
```
;; render: WorldState -> Im
;; Draws a WorldState as a
```
2. **Signature** – types of the function i
   - Use Data Definitions (or create new data de
   - Use `define/contract` and predicates!
3. **Description** – explain (in E

NOTE:
Different languages have different "signature" or "error handling" mechanisms
- Contracts
- Types
- Asserts
- Try-Catch-Throw

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

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

Function contract

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

```
> (render "bad arg")
render: contract violation
  expected: WorldState?
  given: "bad arg"
  in: the 1st argument of
      (-> WorldState? image?)
  contract from: (function render)
  blaming: C:\Users\stchang\Documents\teaching\CS450\Fall23\lecture04.rkt
   (assuming the contract is correct)
  at: C:\Users\stchang\Documents\teaching\CS450\Fall23\lecture04.rkt:37:18
```

Good error message: precise, and no internal details!

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)

Homework Testing

All HW submissions must include `tests.rkt`, which:

- `requires` the hw code file, e.g., `hw0.rkt`
- `defines` a `rackunit` `test-suite` called `TESTS`
- `provide` `TESTS`
- includes sufficient `test-cases` (from the **Design Recipe**) for every hw function definition
- runs without error!

hw0-start / tests.rkt

```
1   #lang racket
2
3   (require rackunit
4            "hw0.rkt")
5
6   (provide TESTS)
7
8   (define TESTS
9     (test-suite
10      "hw0 test suite"
11
20      ;; (test-case
21      ;;     "Exercise 11: distance function"
22      ;;   (check-equal? (exercise11 5 12) 13))
23
24    ;; (test-case
25      ;;     "Exercise 12: cvolume"
26      ;;   (check-equal? (cvolume 10) 1000))
27
28
29  (module+ main
30    (require rackunit/text-ui)
31    (run-tests TESTS 'verbose))
```

Used by autograder

(See **rackunit** docs for more testing functions)

e.g., **check-exn** for fail test cases!

Used for your own testing

In-class Office Hours

- Get HW0 working
- Add test-suite to HW0
  - 2 per function
  - I might run against other submissions and award bonus pts
- Start HW1

Check-In Quiz 9/18

on gradescope

(due 1 minute before midnight)