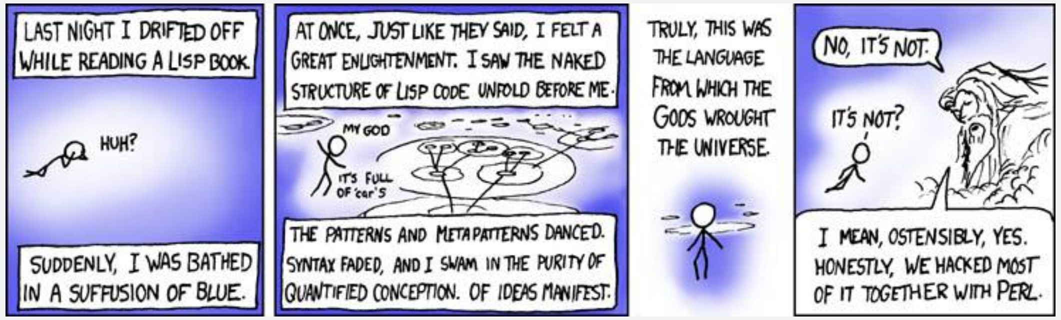
CS450 (section 2)
High Level Languages

UMass Boston Computer Science

Monday, September 18, 2023


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

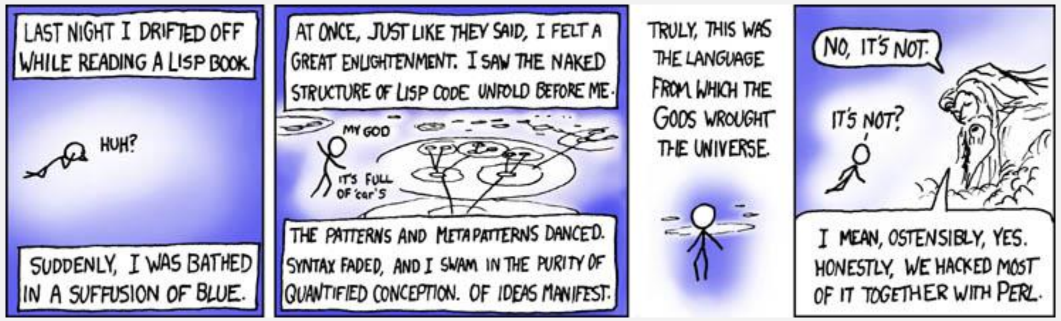
Logistics

- 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 NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

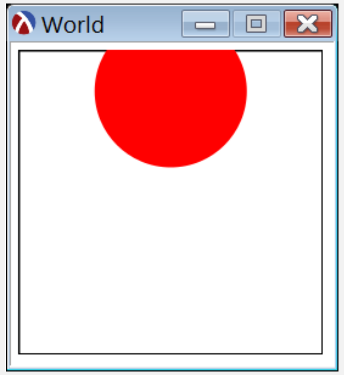
Last Time

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

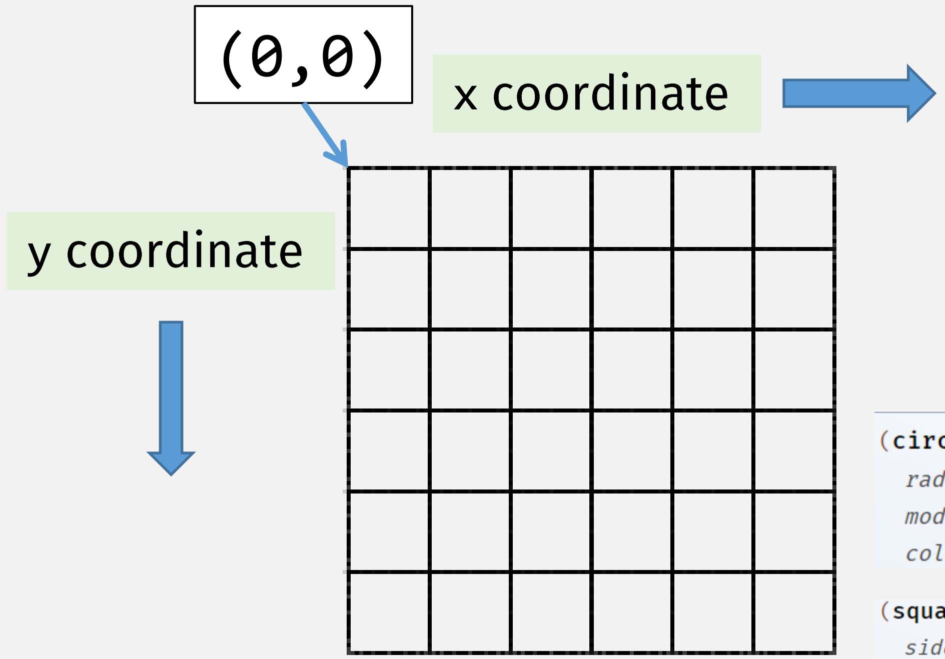
Create **Data Definitions**

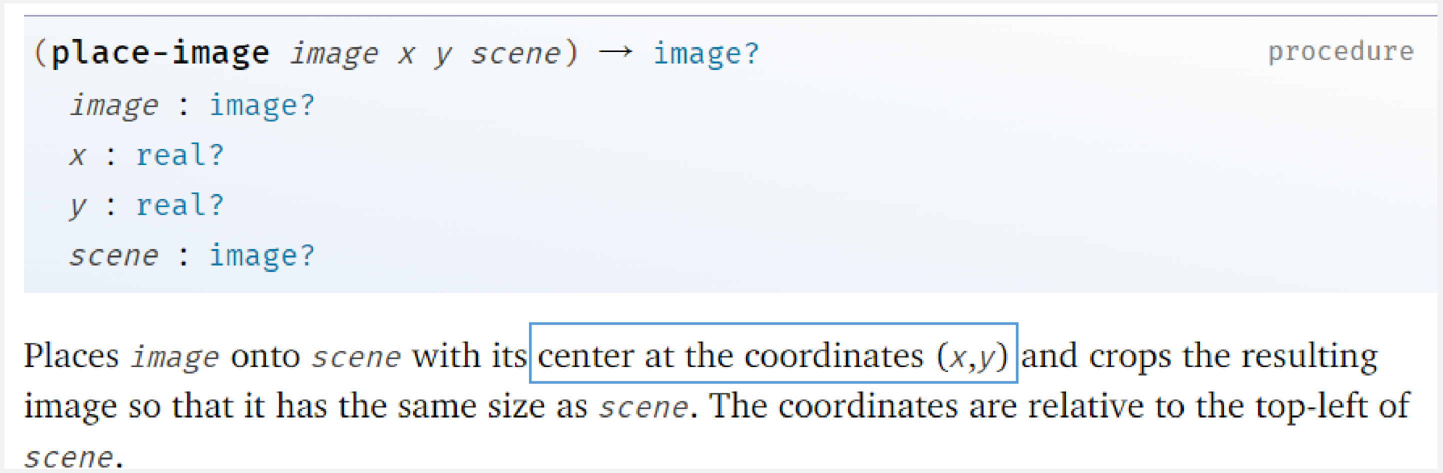
- Describes the types of data that the program operates on
- Has ~~3~~ **4** parts:
  1. A defined **Name**
  2. Description of **all possible values** of the data
  3. An **Interpretation** explains the real world concepts the data represents
  4. A **predicate** returns `true` if a given value is in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

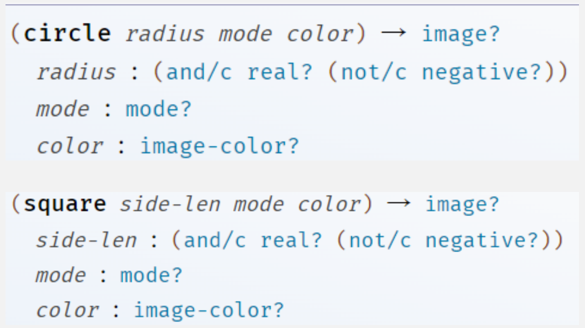
Interlude: htdp universe coordinates





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

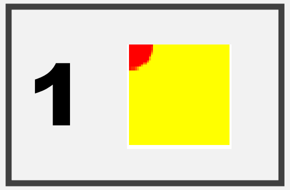
Places *image* onto *scene* with its center at the coordinates (*x*,*y*) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

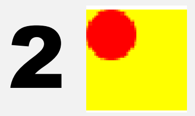


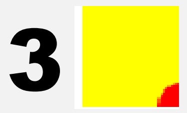
```
(circle radius mode color) → image?
  radius : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?

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

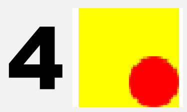
```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???









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
```

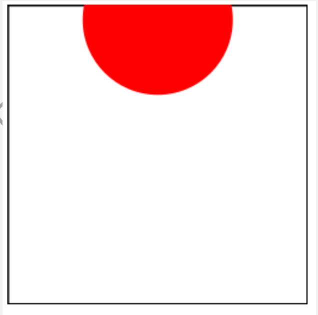
1. **Name**
2. **Signature** – types of the function input(s) and output
   - Use Data Definitions (or **create new data defs**, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how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

FAQ: What about "error-checking"?

6. **Tests** – check (using rac

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

Designing Functions

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

… 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 Engl

```
> (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 rackun

BUT: This is a **bad error message** because …

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

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

FAQ: What about "error-checking"?

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

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

4.

5.

6.

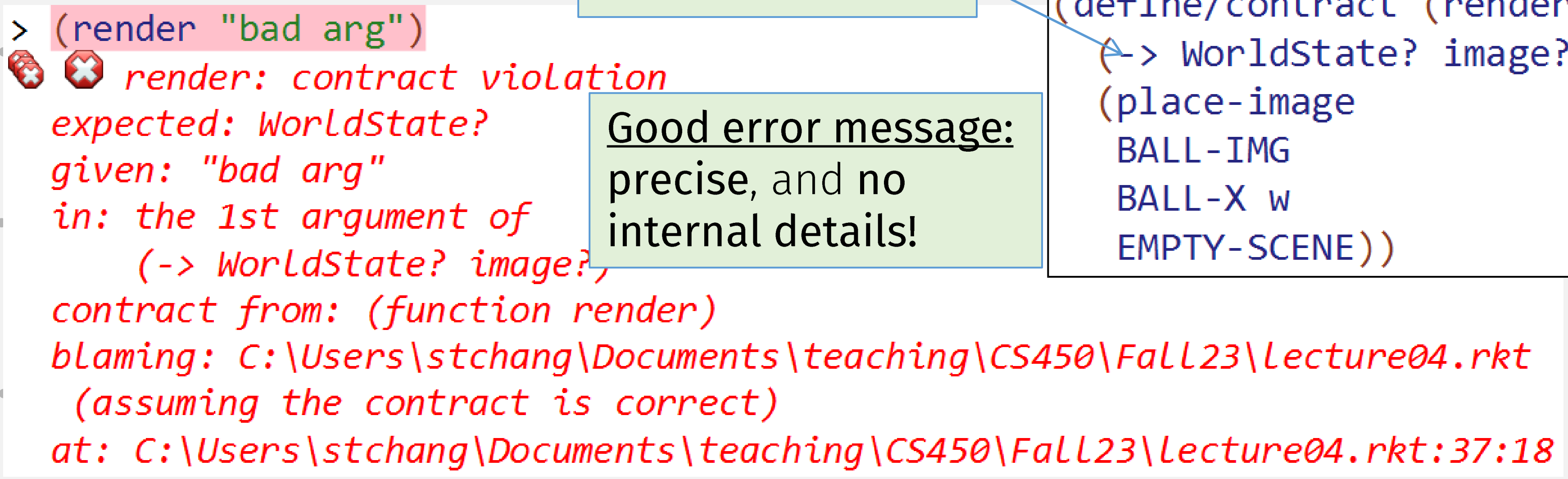
NOTE:
Different languages have different "signature" or "error handling" mechanisms
- Contracts
- Types
- Asserts
- Try-Catch-Throw

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!

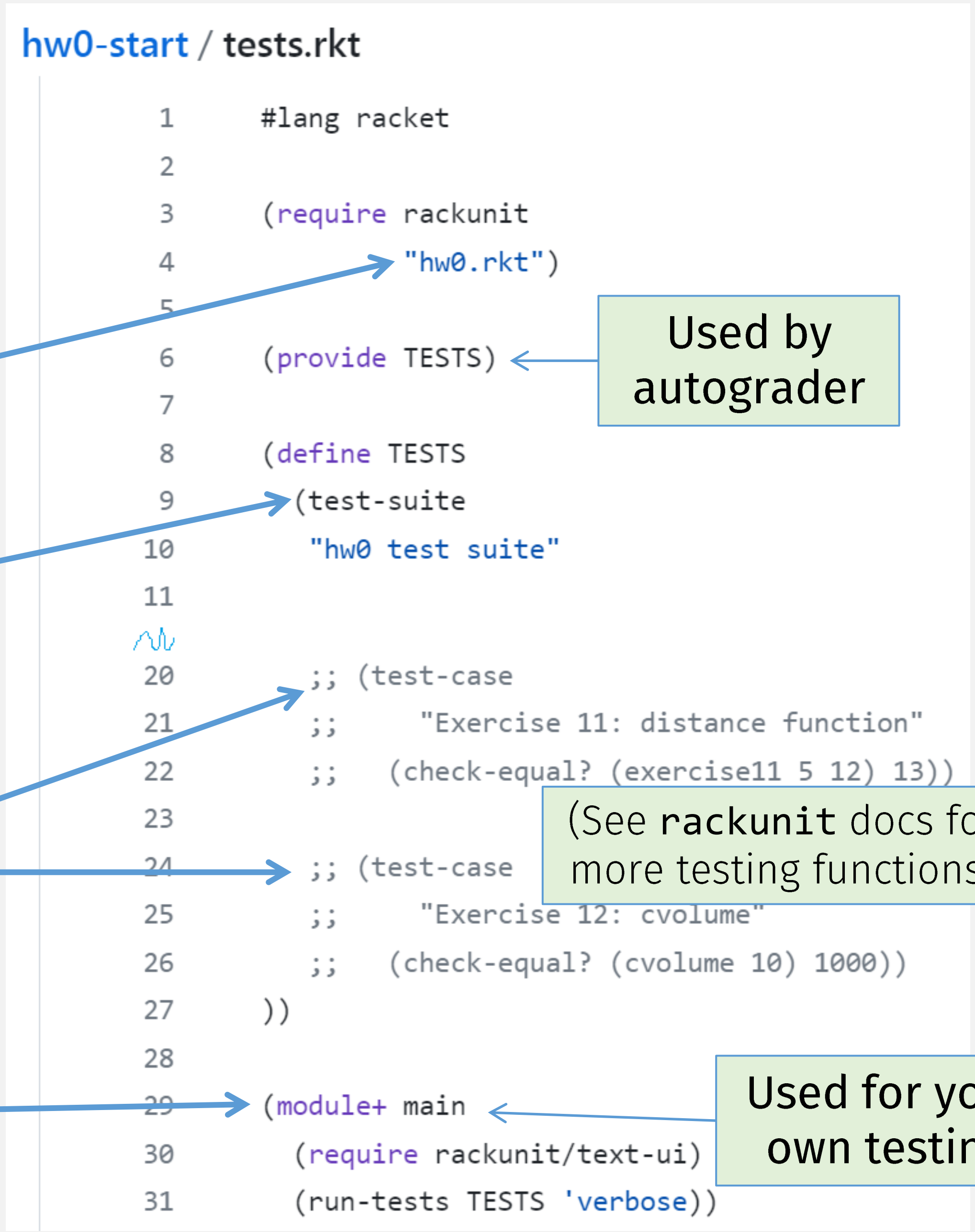
Designing Functions

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

```
#lang racket

(require rackunit
         "hw0.rkt")

(provide TESTS)

(define TESTS
 (test-suite
   "hw0 test suite"

   ;; (test-case
   ;;     "Exercise 11: distance function"
   ;;   (check-equal? (exercise11 5 12) 13))

   ;; (test-case
   ;;     "Exercise 12: cvolume"
   ;;   (check-equal? (cvolume 10) 1000))
))

(module+ main
  (require rackunit/text-ui)
  (run-tests TESTS 'verbose))
```

Used by autograder

(See `rackunit` docs for more testing functions)

Used for your own testing

In-class Programming

- 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)