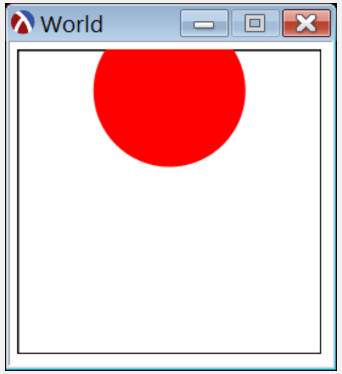
CS450

“Big Bang”, Testing, Contracts

UMass Boston Computer Science

Tuesday, February 10, 2026



Announcements

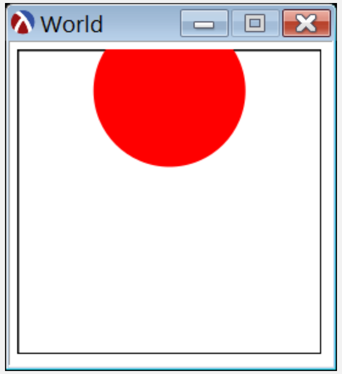
- HW 1 in, HW 0 grades out
  - Questions / complaints: must use gradescope re-grade request use
  - Re-grade requests: must address specific deduction - may result in +/- points
  - Vague / unclear questions, e.g., "I really need a few points back", won't be answered

- HW 2 out
  - due: Tue 2/17 11am EST

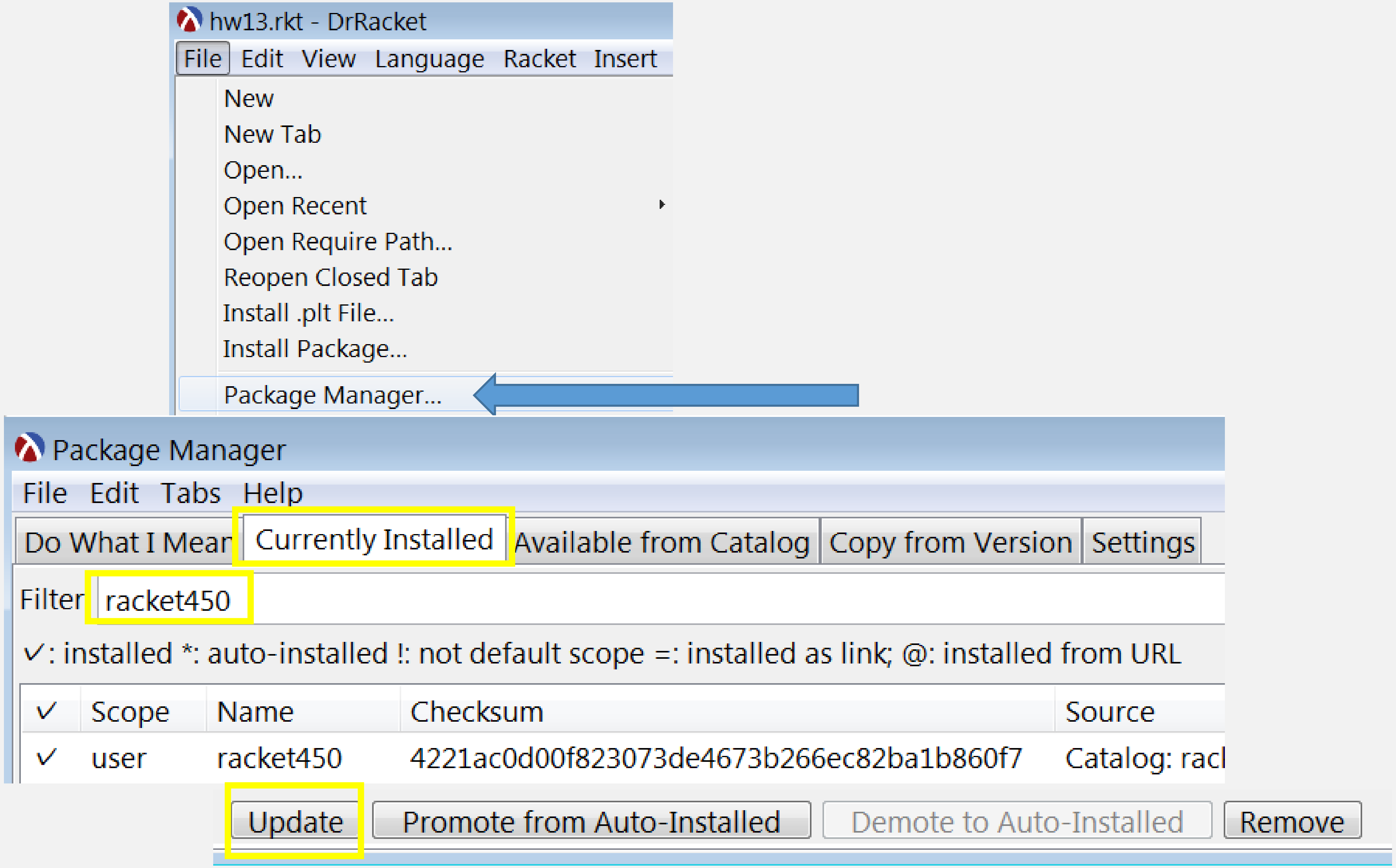
- No HW questions by email! (easy to miss)
  - Post to piazza (use private or anonymous if unsure) (I may change)
  - Make it easier for students/staff to avoid asking/answering duplicate questions

- Reminder: there's no autograder available to students
  - So: no mention of autograder please
  - Also: it may be wrong, incomplete, and subject change without notice
  - If you manage to get some benefit, consider it bonus information
  - Instead: ask questions using small examples! (no code dumps) (See forum rules)

- Course web site:
  - Added Design Recipe section



Update “racket450”



Filter racket450

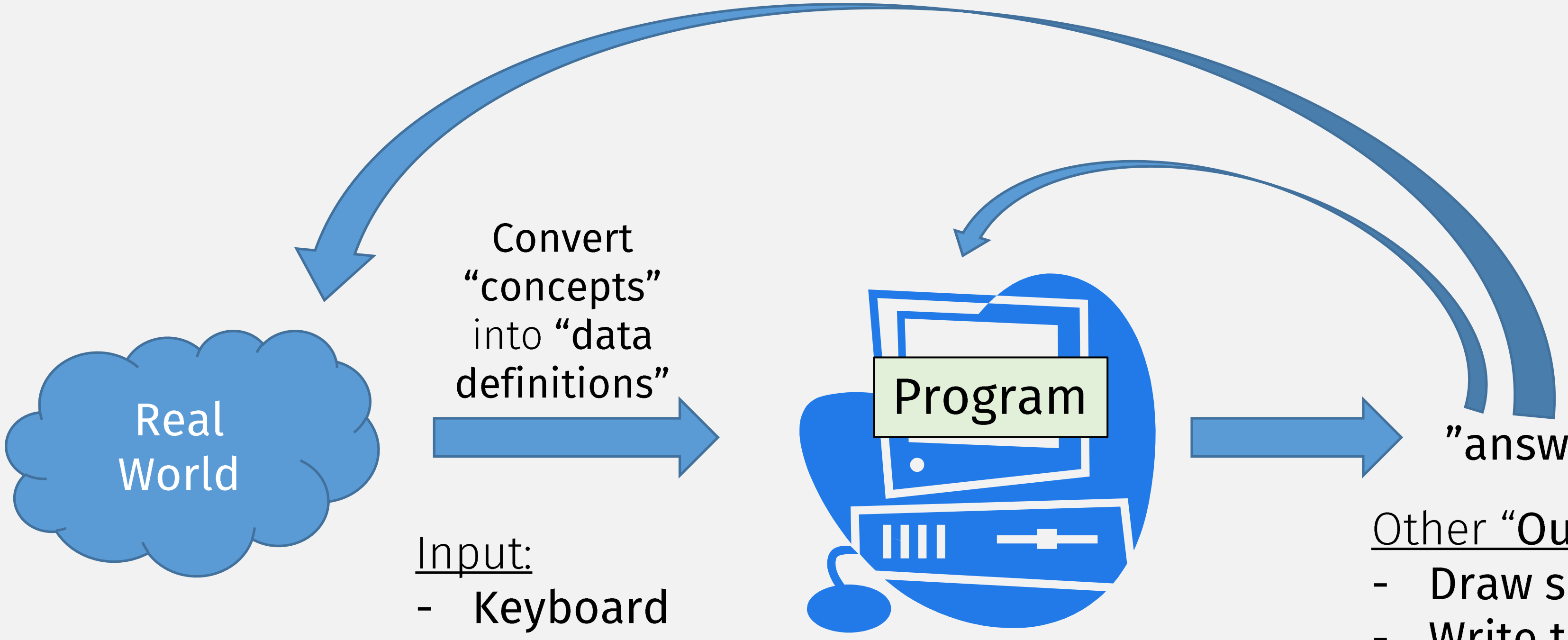
✓: installed *: auto-installed !: not default scope =: installed as link; @: installed from URL

✓	Scope	Name	Checksum	Source
✓	user	racket450	4221ac0d00f823073de4673b266ec82ba1b860f7	Catalog: racl

Last Time

Programs can be Interactive

More fun to write and use!



Convert “concepts” into “data definitions”

”**answer**”, e.g., **42**

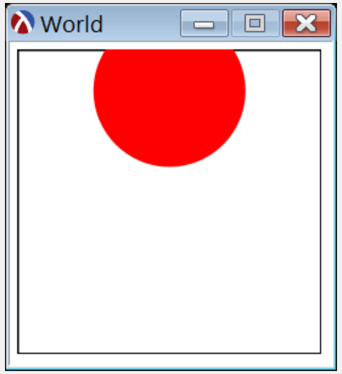
Input:
- Keyboard
- Mouse
- Gamepad
- Touchscreen
- Voice
- File

Other “**Output**” (**side effect**):
- Draw screen
- Write to file
- Play sound
- Change real world (unlock door)
- Changed computer state (game)

Interactive Programs (with **big-bang**)

```
(require 2htdp/universe)
```

- DEMO



```
(require 2htdp/universe)
```

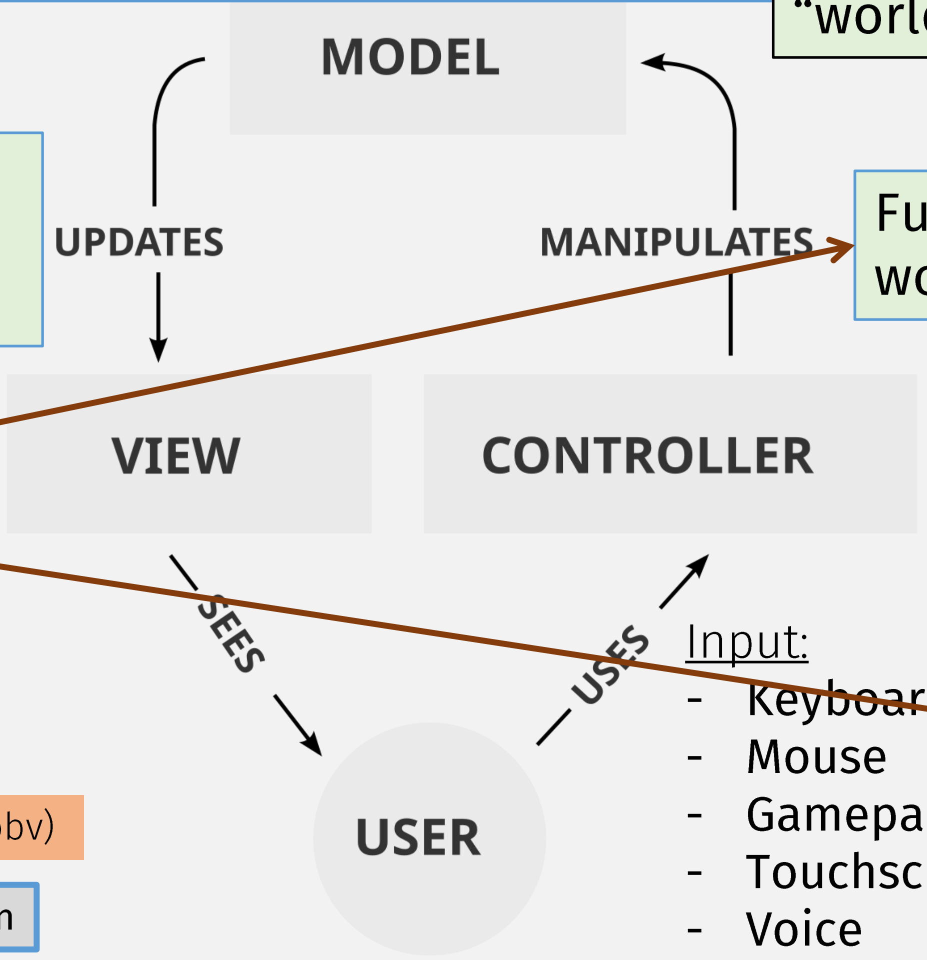
Interactive Programs (with `big-bang`)

- **`big-bang` starts an** (MVC-like) **interactive loop**

Model-View-Controller (MVC) Pattern

Requires a **data definition!**

“world” state



Function to “convert” world state data … into a “view” **image**

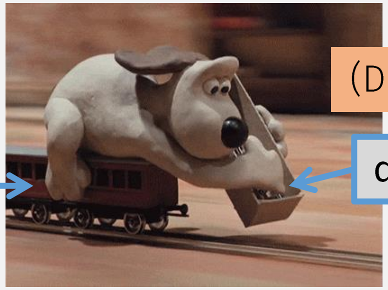
Functions that “update” world state data

Can’t write any code without a Data Definition!

Input:

- Keyboard
- Mouse
- Gamepad
- Touchscreen
- Voice
- File

Input can also “update” world state data



(Don’t do this, obv)

code

data definition

(programmers who skip data design step)

from: **wikipedia.org**

Interactive Programs (with `big-bang`)

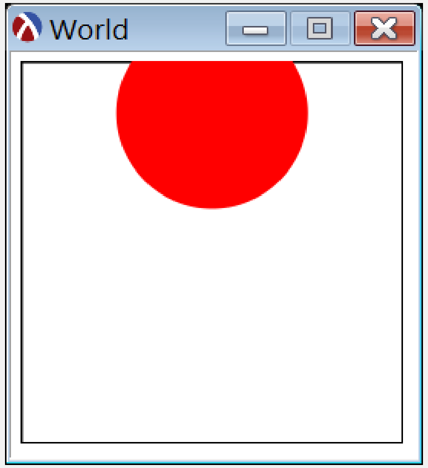
```
(require 2htdp/universe)
```

- **`big-bang`** starts an (MVC-like) interactive loop
  - repeatedly updates a "world state"
  - Programmer must first define what "the World" is ...
  - ... with a Data Definition!

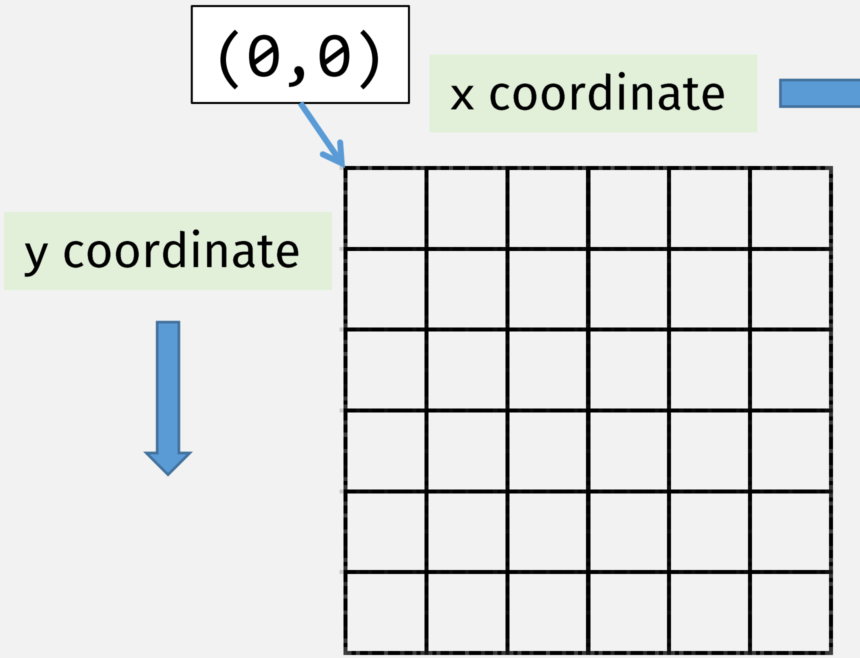
```
;; A WorldState is a Non-negative Integer
;; Represents: y-coordinate of a circle
center, in a big-bang animation
```

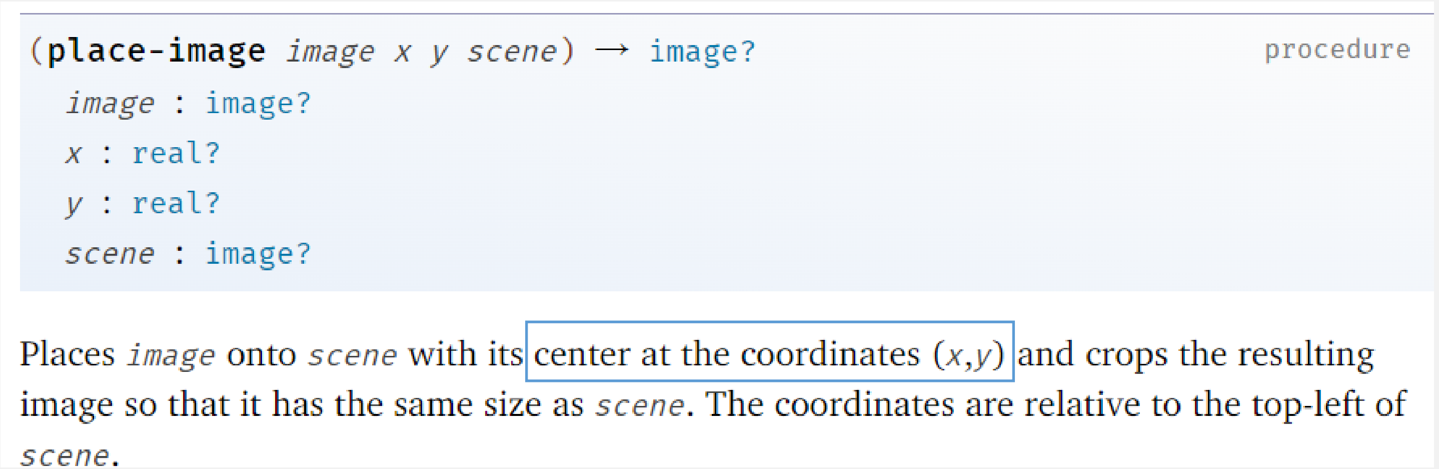
Data Definitions should represent values that change

(Values that don't change should be defined as constants)



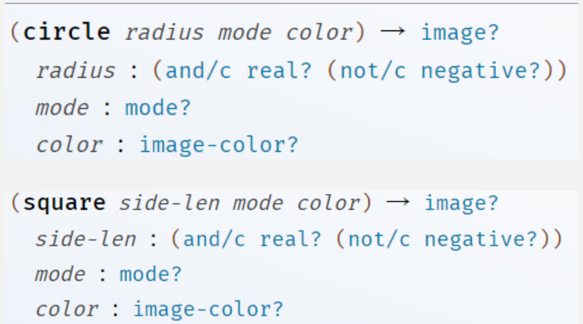
Interlude: htdp universe coordinates



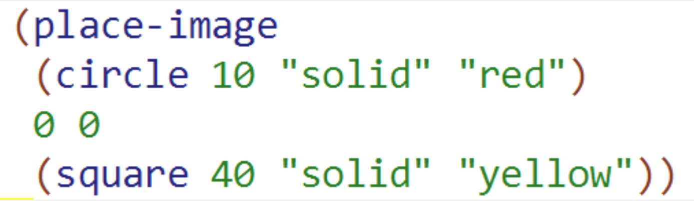


```
(place-image image x y scene) → image?                              procedure
  image : image?
  x : real?
  y : real?
  scene : image?
```

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*.

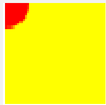


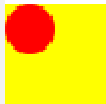
```
(circle radius mode color) → image?
  radius : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?

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

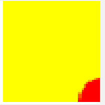


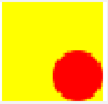
```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???

[✓ 1] 

2 

3 

4 

`(require 2htdp/universe)`

Interactive Programs (with **big-bang**)

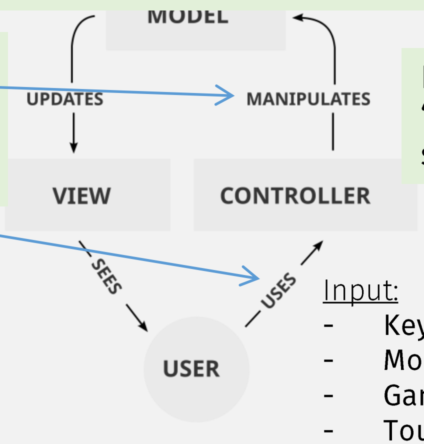
- **big-bang** starts an (MVC-like) interactive loop
  - repeatedly updates a “world state”
  - Programmer must define what “the World” is …
  - … with a Data Definition!

```
;; A WorldState is a Non-negative Integer
;; Represents: y-coordinate of a circle
center, in a big-bang animation
```

- Programmers specify “handler” functions to manipulate “World”
  - Render
  - World update
  - Input handlers

“world state” **data definition!**

Function that “converts” world state data … into an **image**



Functions that “update” world state data

Input:
- Keyboard
- Mouse
- Gamepad
- Touchscreen

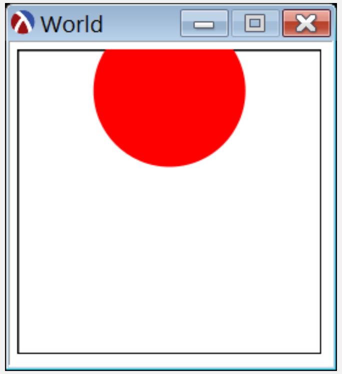
Last Time

Design Recipe Intro: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has 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

```
;; A WorldState is a Non-negative Integer
;; Represents: y-coordinate of a circle
center, in a big-bang animation
```



Last Time

Design Recipe Intro: Data Design

Create **Data Definitions**

Remember: these are **formal names!**
- Cannot reference undefined names
- Name must be exact when using it

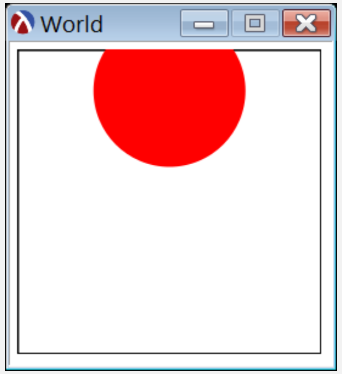
- Describes the types of data that the program operates on
- Has **4** parts:
  1. A defined **Name**
  2. Description of **all possible values** of the data
  3. An **Interpretation** explains the real world concepts the data represents
  4. A **predicate** is code that checks if a value is in the Data Definition
     - returns `false` if a given value is not in the data definition

STYLE: use CapitalizedCamelCase for user-defined data def names

```
;; A WorldState is a Non-negative Integer
;; Represents: y-coordinate of a circle
center, in a big-bang animation
```

STYLE: same as data def name plus "?" suffix

```
(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
   - Refer to Data Definitions (**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

“built-in” data def (from 2htdp/image lib)

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

1. **Name**
2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (**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”?

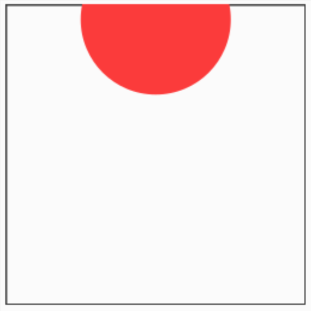
> “Error handling is important,
> but if it obscures logic, it’s wrong.”
> — **Robert C. Martin,** Clean Code: A Handbook of Agile Software Craftsmanship

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
   - Refer to Data Definitions (**create new data defs**, if needed)



3. **Description** – explain (in English prose) how the function works

STYLE: constant names are in ALL-CAPS

4. **Examples** – show (using **check-equal?**) how the function works
   - (put before function definition)

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

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

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

6. **Tests** – check (using ra

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

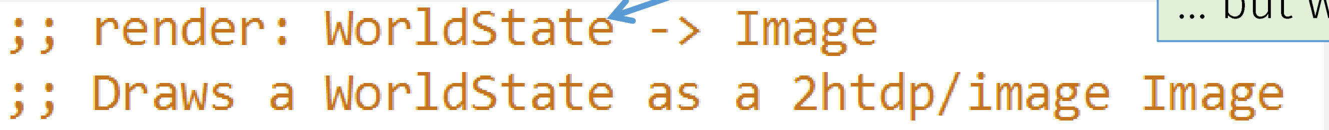
FAQ: What about “error-checking”?

Designing Functions

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

… but we can make it more robust

1. Name

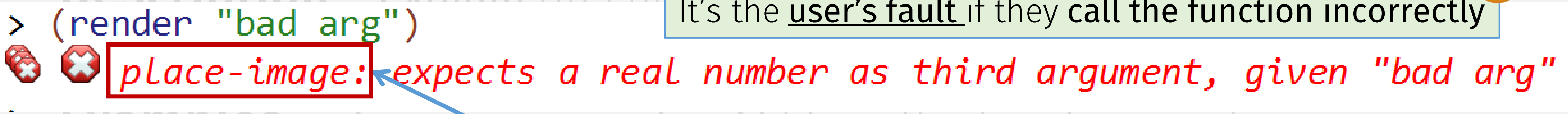


```
;; render: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
```

2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (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 rackunit) how the function works

BUT: This is a bad error message because … ☹

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

5. Code – implement how the functi

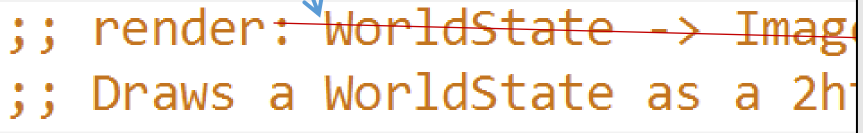
6. Tests – check (using `rackunit`) that the function works

Delete comment Signatures after writing contracts

More Robust Signatures

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

1. Name



```
;; render: ~~WorldState -> Imag~~
;; Draws a WorldState as a 2ht
```

2. **Signature** – types of the function inpu
   - Refer to Data Definitions (create new data
   - Use define/contract with predicates!

The **Design Recipe** is language-agnostic

For each step, use the appropriate **high-level feature** in the **language you're using**

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

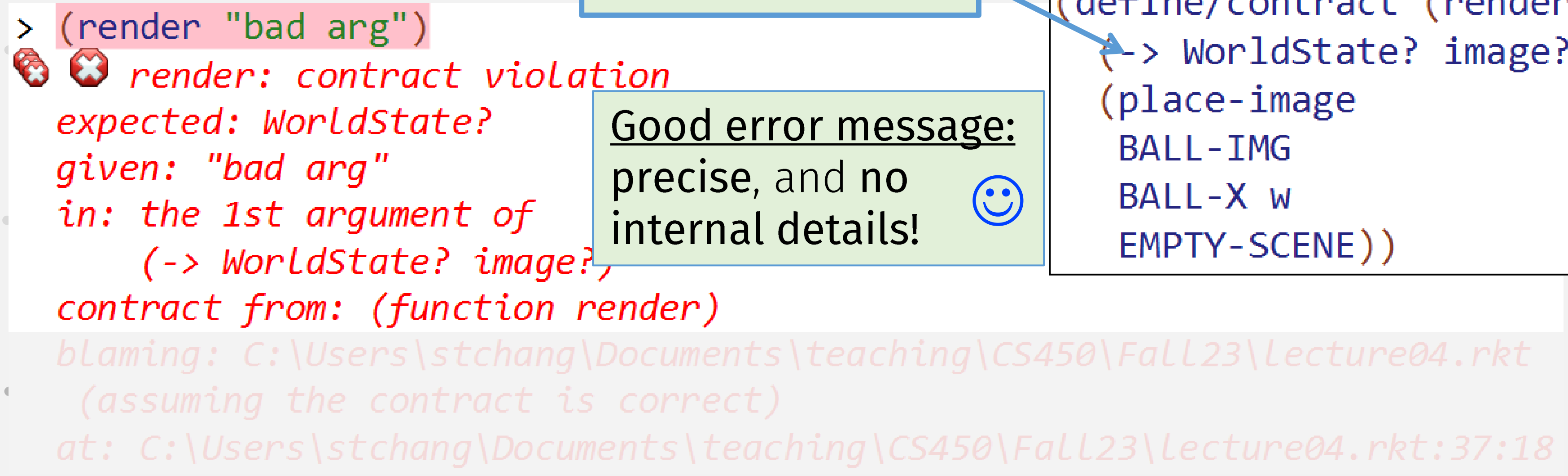
4.

5.

6.

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! ☺

STYLE note: Overcommenting

“The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. **Comments are always failures**.”
– **Robert C. Martin,** Clean Code: A Handbook of Agile Software Craftsmanship

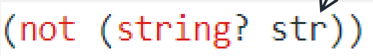
“Redundant comments are just places to collect lies and misinformation.”
– Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

“Don’t Use a Comment When You Can Use a Function or a Variable”
– **Robert C. Martin,** Clean Code: A Handbook of Agile Software Craftsmanship

Design Recipe mostly **tells you what comments to write!**

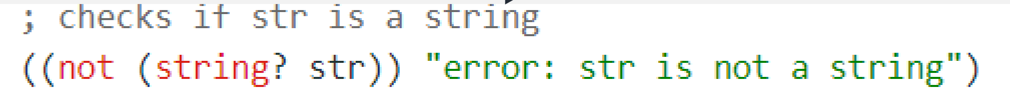
- Use comments to explain code if needed, BUT …
  - … the **best code needs** no comments
- Redundant comments makes code harder to read
  - More comments ≠ “better”
- (Also, don’t submit commented-out code!)

(not a great variable name)



```
(not (string? str))
```

Terrible comment



```
; checks if str is a string
((not (string? str)) "error: str is not a string")
```

Designing Functions

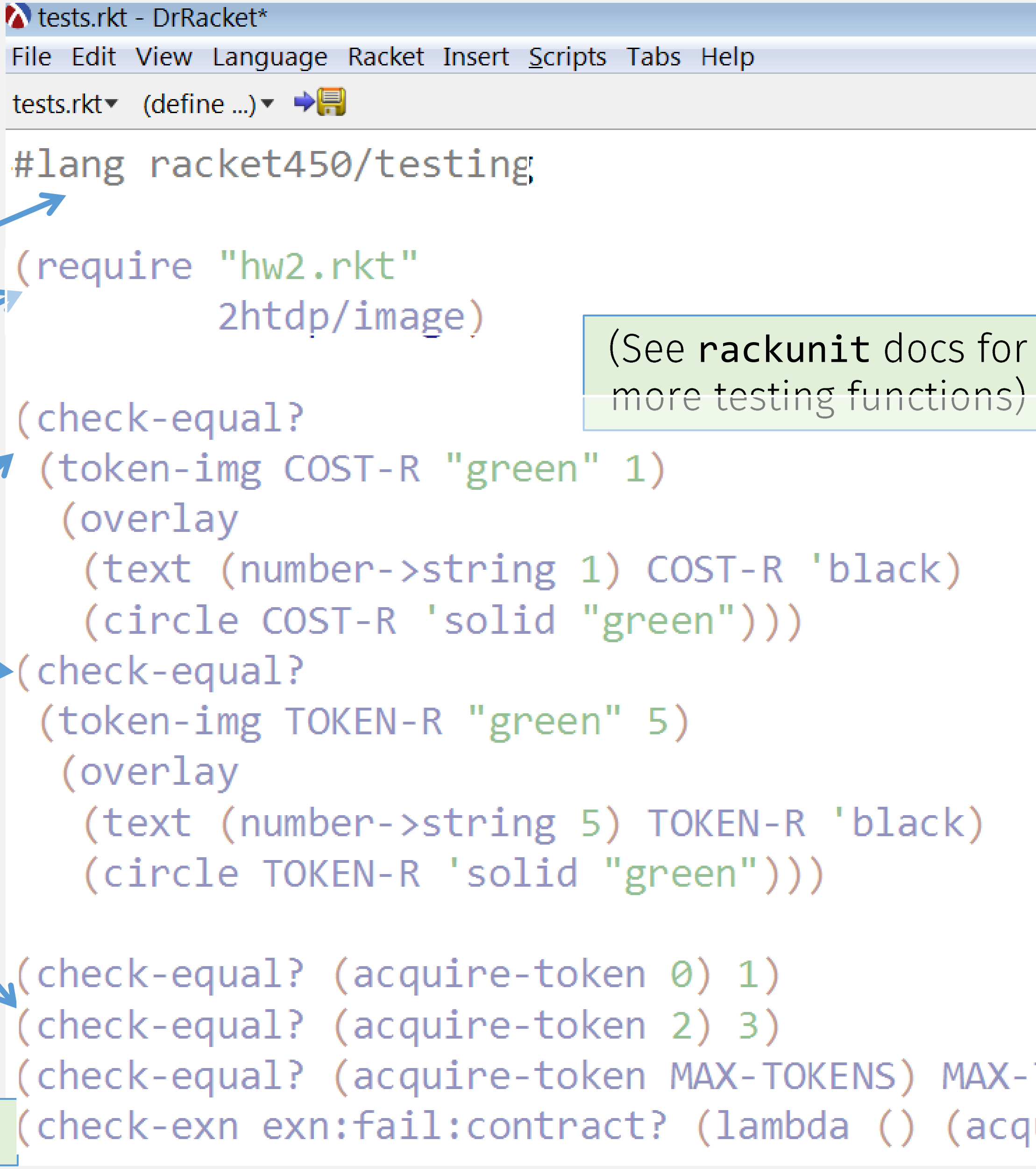
1. **Name**
2. **Signature** – <u>types</u> of the function input(s) and output
   - Refer to Data Definitions (**create new data defs**, if needed)
   - Use `define/contract` **with predicates!**
3. **Description** – <u>explain</u> (in English prose) how the function works
4. **Examples** – <u>show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

Homework Testing

<u>All</u> HW submissions <u>must</u> include **`tests.rkt`**, which:

- uses **`#lang racket450/testing`**
- requires hw code file, e.g., **`hw2.rkt`**
- includes sufficient test cases (from the **Design Recipe**) for every function def
- Must run without error and all tests passing!

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



```
#lang racket450/testing

(require "hw2.rkt"
         2htdp/image)

(check-equal?
 (token-img COST-R "green" 1)
  (overlay
   (text (number->string 1) COST-R 'black)
   (circle COST-R 'solid "green")))
(check-equal?
 (token-img TOKEN-R "green" 5)
  (overlay
   (text (number->string 5) TOKEN-R 'black)
   (circle TOKEN-R 'solid "green")))

(check-equal? (acquire-token 0) 1)
(check-equal? (acquire-token 2) 3)
(check-equal? (acquire-token MAX-TOKENS) MAX-
(check-exn exn:fail:contract? (lambda () (acq
```

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

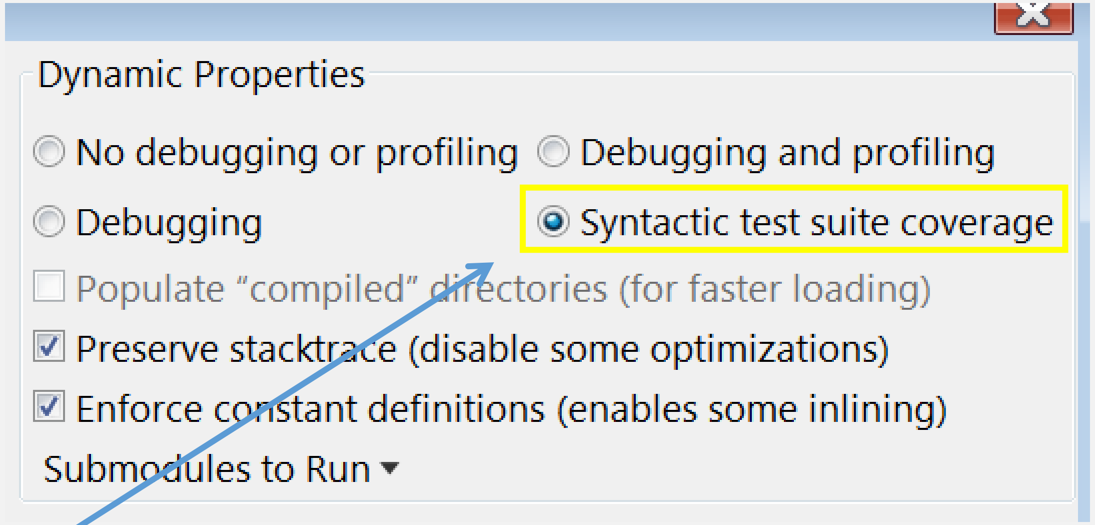
What is a “Sufficient” Number of Tests?

- Wishful: **test every possible input**
  - Usually **impossible: infinite cases**
  - Also **redundant …**

- Realistic: **test all “categories” of inputs**
  - **“category” depends on data defs!**
  - E.g., **“positive” / “negative”, “left” / “right”, valid**
  - Try to think of **corner cases !**

- Minimum: **100% (Test / Example) “Coverage”**
  - **All code is run once by some test**
  - In **“Choose Language” Menu**
  - NOTE: **only works with single files**
  - **Doesn’t guarantee “correctness”!** (why?)

- Ideally: **Until 100% confident in “correctness”**



Dynamic Properties

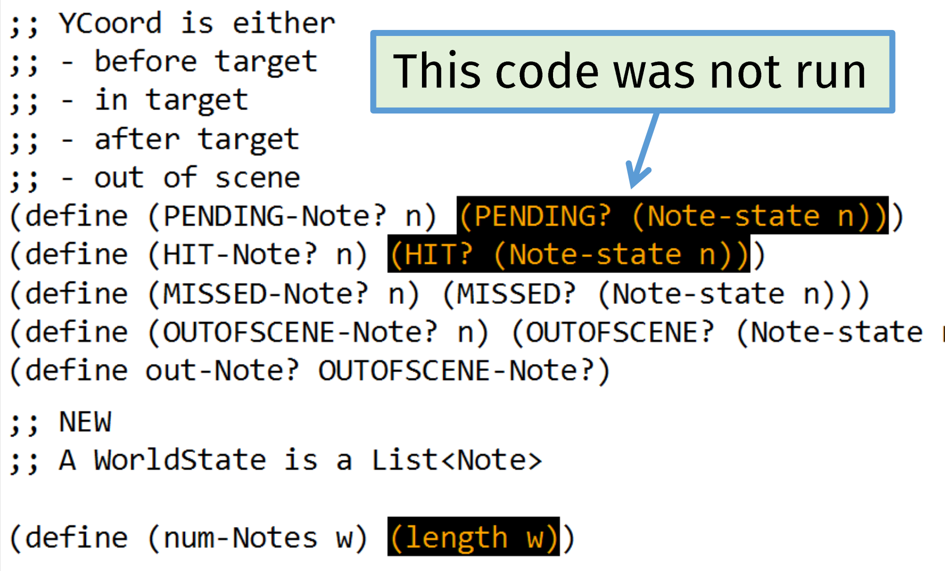
○ No debugging or profiling ○ Debugging and profiling

○ Debugging ● Syntactic test suite coverage

□ Populate “compiled” directories (for faster loading)

☑ Preserve stacktrace (disable some optimizations)

☑ Enforce constant definitions (enables some inlining)



```
;; YCoord is either
;; - before target
;; - in target
;; - after target
;; - out of scene
(define (PENDING-Note? n) (PENDING? (Note-state n)))
(define (HIT-Note? n) (HIT? (Note-state n)))
(define (MISSED-Note? n) (MISSED? (Note-state n)))
(define (OUTOFSCENE-Note? n) (OUTOFSCENE? (Note-state
(define out-Note? OUTOFSCENE-Note?)

;; NEW
;; A WorldState is a List<Note>

(define (num-Notes w) (length w))
```

This code was not run

Last Time

Design Recipe

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

Programming is an **iterative** process!

Each iteration should be **incremental**!

Last Time

The *faster* way to program

The Incremental Programming Pledge

"slow down to speed up"

At all times, all of the following should be **`true`** of your code:

1. Comments (**data defs, signatures**, etc) **match code**
2. **Code has no** syntax errors
    1. E.g., **missing / extra parens**
3. Runs **without runtime errors / exceptions**
    1. E.g., **use undefined variables,** `div by zero`**, call a "non function"**
4. **All** tests pass

When you **make a code edit** that **renders one of the above** **`false`, STOP …**

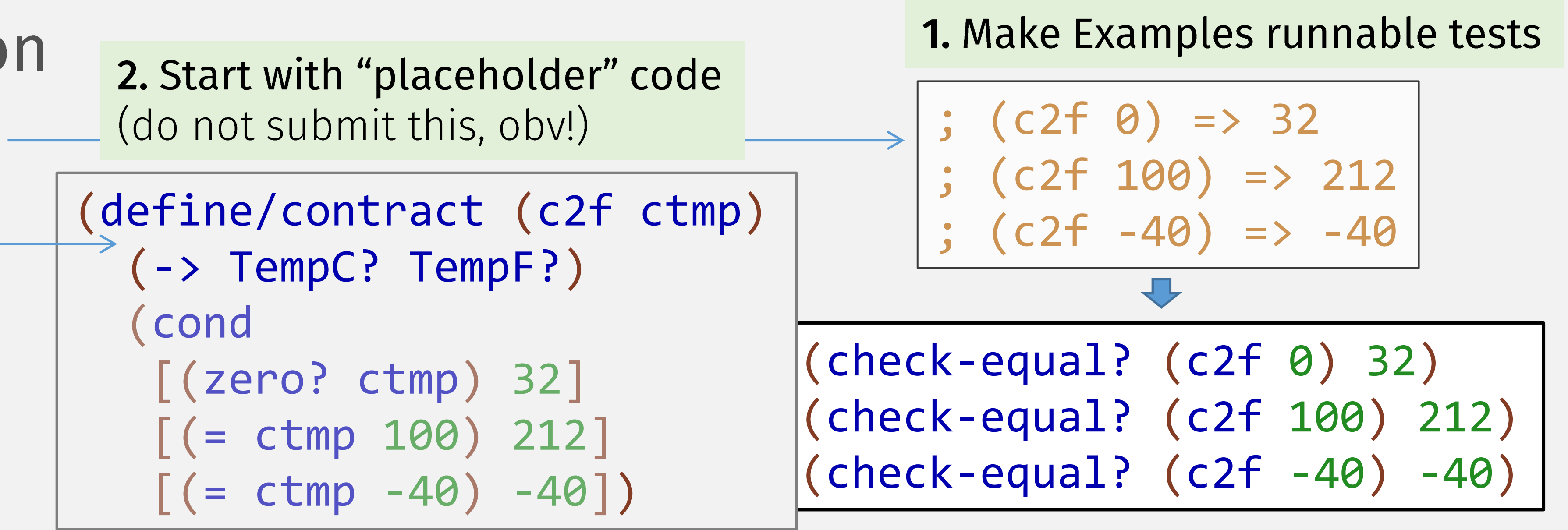
… and **<u>don't do anything else</u>** until **all the statements are true again.**

(this way, it's easy to revert back to a "working" program)

Incremental Programming, in Action

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference "defined" Data Definition names
3. Description
4. Examples
5. Code
6. Tests

```
;; c2f: TempC -> TempF
;; Converts a Celsius temperature to Fahrenheit
```



1. Make Examples runnable tests

```
; (c2f 0) => 32
; (c2f 100) => 212
; (c2f -40) => -40
```

```
(check-equal? (c2f 0) 32)
(check-equal? (c2f 100) 212)
(check-equal? (c2f -40) -40)
```

2. Start with "placeholder" code (do not submit this, obv!)

```
(define/contract (c2f ctmp)
  (-> TempC? TempF?)
  (cond
   [(zero? ctmp) 32]
   [(= ctmp 100) 212]
   [(= ctmp -40) -40]))
```

Incremental Programming, in Action

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference “defined” Data Definition names
3. Description
4. Examples
5. Code
6. Tests

```
;; c2f: TempC -> TempF
;; Converts a Celsius temperature to Fahrenheit
```

1. Make Examples runnable tests

2. Start with “placeholder” code

3. Make small changes only (something easy to revert)

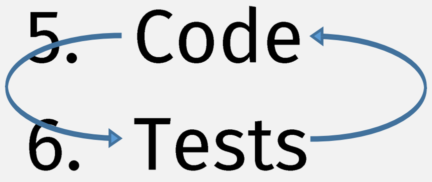
```
(define (c2f ctemp)
  (+ (* ctemp (/ 9 5)) 32))
```

4. Test each (small) **change** (before making another one)

Incremental Programming: Real-World Example

- https://www.youtube.com/watch?v=1SlGgCxJa3w
- > “when you do everything at once …
  > you’re not sure why it’s not working!”
- > “when you layer it, when you break it down …
  > and you hit a spot when it’s not working …
  > then you can just focus on that spot!”

3. Make <u>small changes only</u> (something easy to revert)



5. Code
6. Tests

4. Test each (small) **change** (before making another one)

In-class Office Hours

- Get HW 0 / HW 1 “working”?
- Update `racket450`
- Add `tests.rkt` using `#lang racket450/testing` for HW1
- Start HW 2

Warning: **HW files should not start `` `big-bang` `` loop automatically when run!**