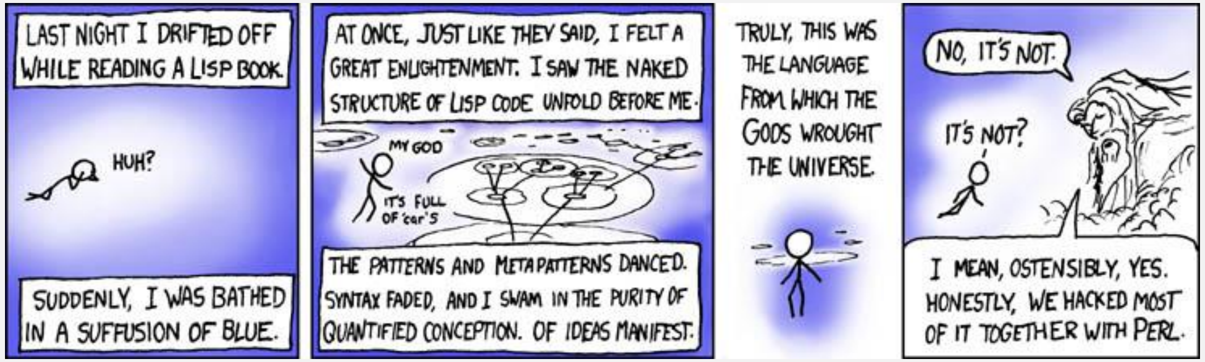
# CS450

# High Level Languages

UMass Boston Computer Science

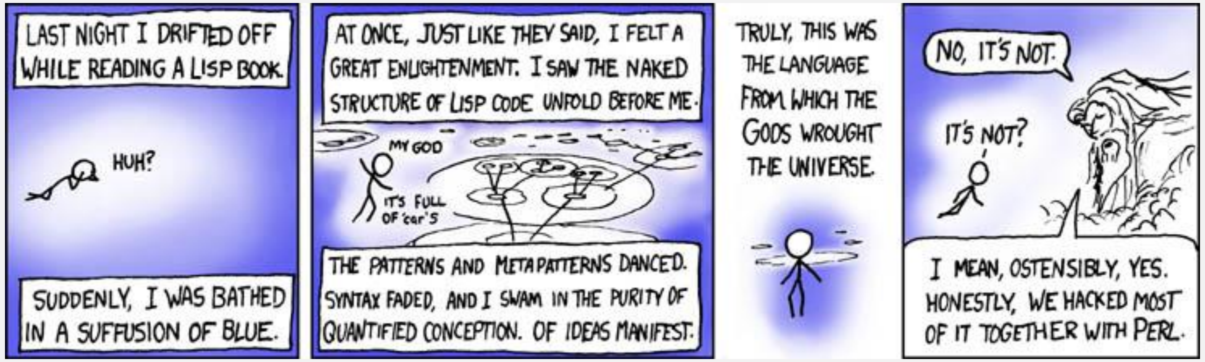
Thursday, February 5, 2026



# *Logistics*

- HW 0 in
  - ~~Due: **Thu 2/5 11am EST**~~
- HW 1 out
  - Due: **Tue 2/10 11am EST**
- Course web site:
  - Style: see "Racket Basics and Style"

`https://www.cs.umb.edu/~stchang/cs450/s26`



*Previously*

# Statements vs Expressions

Most other courses

**Imperative** programs are:
… sequences of (“low level”) **statements** / instructions
(C, Java, Python)

This course

**Declarative** programs are:
… (“high level”) **declarative expressions**, i.e., “arithmetic”
(Racket)

```
int add_one ( int x ) {
  return x + 1;
}
```

```
(define (add-one x)
  (+ x 1))
```

Last Time

# Arithmetic … on More Than Numbers!

**This position must be an** (arithmetic expression that evaluates to a) **function value**

- Function call: **prefix notation** (fn name first)
  - **Easier to write multi-arity functions**

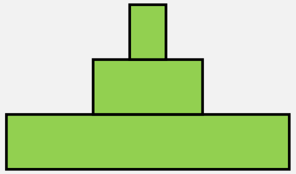
```
(+ 1 2 3 4)
```

- (fundamental) programming model: **arithmetic** expressions
  - But **not just numbers!**
  - When **"run", arithmetic expressions evaluate** to an **answer** or **value**

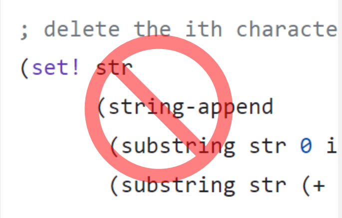
```
(string-append "hi" "world")
```

run "hi-world"

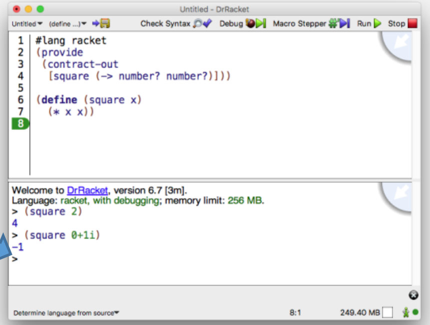
(above  )

run 

- **No statements!**
  - E.g., "**assign**" or "**return**"




- Use the **REPL** ("interactions") for basic testing!



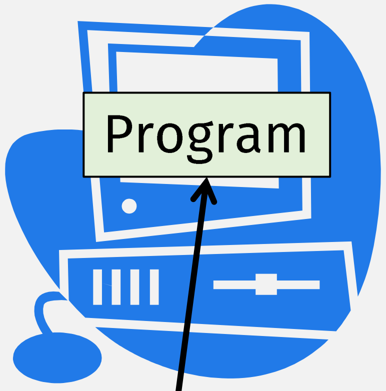
# Programs Need Input

e.g., **students**

Real World

Convert "concept" into "data"

```
class Student {
  int ID;
  int year;
  string address;
  … }
```




Do a "real world" task

"run" (**evaluate**)

"answer", e.g., 42

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

# Program vs Real World

Real World "things" ...

e.g., Temperature

... need a **data representation** in the program

Real World


Input:
- Keyboard
- Mouse

A **Data Definition** name

Specify possible values of the data

;; A **TempC** is an Integer
;; Represents: a temperature in degrees Celsius

**Interpretation ...** connects data to a real world concept

;; A **TempF** is an Integer
;; Represents: a temperature in degrees Fahrenheit

;; A **TempK** is an non-negative Integer
;; Represents: a temperature in degrees Kelvin

When programming,
**choosing data representations must be the first task!**
(way before writing any **code** ...
which **processes the data**)

# Data Design Recipe

(Coding in this course must follow these steps)

*Not for HW1!*

A **Data Definition** name

Specify possible values of the data

```
;; A TempC is an Integer
;; Represents: a temperature in degrees Celsius
```

**Interpretation** … connects data to a real world concept

A data def's **predicate** should reject bad values: i.e., **evaluate to `false`** when **the given argument** is not in the **data definition**

```
(define (TempC? x)
  (integer? x))
```

- A **Data Definition** represents a **real world concept**
- It is what a **program's code computes "on"**
- It has the following components
  1. **Name**
  2. **Set of values specification** (using other data definitions)
  3. **Interpretation** that **explains** the **connection to the real world**
  4. **Predicate - code version** of **Set of Values** (step 2)

     A **predicate** is a **function** that **evaluates to `true`/`false`**

# Data Definitions, in general

Treat data definitions as <u>formal names</u>!

```
;; A TempC is an Integer
;; Represents: a temperature in
degrees Celsius
```

Refers to previously defined
data definition names!
(can be built-in or come from library)

A Function **Signature** will use Data Definitions
… to specify types of input and output data

```
;; Any -> Boolean
(define (TempC? x)
  (integer? x))
```

A data def's **predicate** should reject bad values:
i.e., evaluate to `false` when the given argument is <u>not</u> in the data definition

A **data definition** defines a new "<u>type</u>" of data

- <u>Different languages</u> have different mechanisms to define new types of data:
  - `typedef`
  - `class`
  - `enum`
  - `struct`
- <u>In this course</u>, we use a combination of comments + code

# Design Recipe(s)

Not for HW1!

(Steps to follow when writing a program)

- Data Design
- Function Design

# Designing Functions

```
;; A TempC is an Integer
;; Represents: a temp in degrees Celsius
;; A TempF is an Integer
;; Represents: a temp in degrees Fahrenheit
```

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference “defined” Data Definition names

(user-defined, or built-in)

3. Description

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

# Designing Functions

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference "defined" Data Definition names
3. Description – explains how fn works, in English
4. Examples – shows how fn works, in code
5. Code
6. Tests

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

Yes for HW1!

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

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

```
(check-equal? (c2f 1) (/ 169 5))
```

From `racket450` testing framework (stay tuned!)

# Designing Functions

```
;; A TempC is an Integer
;; Represents: a temp in degrees Celsius
;; A TempF is an Integer Rational
;; Represents: a temp in degrees Fahrenheit
```

1. Name
2. Signature
   - **# of arguments** and **their data type**
   - **Output type**
   - May only **reference "defined" Data Definition names**
3. **Description** – explains how fn works, in English
4. **Examples** – shows how fn works, in code
5. Code
6. **Tests**

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

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

```
(check-equal? (c2f 1) (/ 169 5))
```

Something is wrong!
- in Code?
- in Signature?
- in Data Definition?

*Previously*

# *Interlude:* Software Dev 101

## 1. Write Specifications / Requirements

- Figure out what the **program should do**

## 2. Implement code

- Make the **program**

## 3. Verify correctness (i.e., **testing**):

- Check the **program does what it should do?**

| Effort Allocation | |
|---|---|
| Software Engineer | Most Students |
| ~20% | |
| ~40% | ~~~110%~~ ~~~100%~~ ~150% "Assignment is too long!" |
| ~40% | |

Hint: How to do well in this course …

Previously

# *Interlude:* Software Dev 101

1. Write Specifications / Requirements
   - Figure out what the **program should do**

2. Implement code
   - Make the **program**

3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**

**Effort Allocation**

For Small projects

| Software Engineer | Most Students |
|---|---|
| ~~~20%~~ ~33% | |
| ~~~40%~~ ~33% | ~~~110%~~ ~~~100%~~ ~150% |
| ~~~40%~~ ~33% | |

"Assignment is too long!"

Hint: How to do well in this course …

# Function Design Recipe … is Software Dev!

1. **Name**
2. **Signature**
   - **# of arguments** and **their data type**
   - **Output type**
   - May only **reference “defined” Data Definition names**
3. **Description** – explains how fn works, in English
4. **Examples** – shows how fn works, in code
5. **Code** Implement ~33%
6. **Tests** Verify ~33%

Specify ~33%

# Design Recipe(s)

(Steps to follow when writing a program)

- Data Design
- Function Design

Programming is an **iterative** process!

# Iterative Programming

Other functions ("wish list")

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

Programming is an **iterative** process!

**Danger, Danger**

This is not a license to "hack"

i.e., **arbitrarily changing code and praying "this time it will just work"**

Instead, **program incrementally**

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., **undefined variable use,** `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 **don't do anything else** 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 (c2f ctmp)
  (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 Tips Summary

1. Make Examples runnable tests
2. Start with "placeholder" code
3. Make small changes only

   Implies: **Write small functions!**
4. Test each (small) change, before making another one

In this course, **all conditions** of the **Increment Programming Pledge** must be true at all times!

<u>This means:</u>
A **programmer should never be stuck** with a **large amount of code** where they **"don't know what's wrong"**

# *Conventional Wisdom:* Write Small Functions

## Write Short Functions

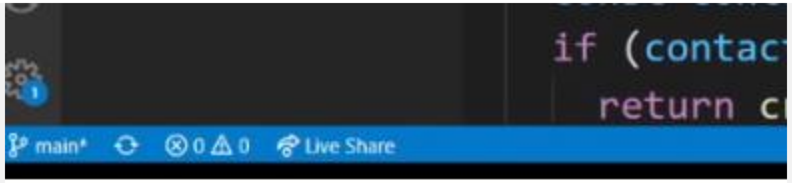
**Google C++ Style Guide**

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. Small functions are also easier to test.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.




**Why I Never Write Long Functions**

Web Dev Simplified
1.44M subscribers
Subscribe

100K views 2 years ago Clean Code

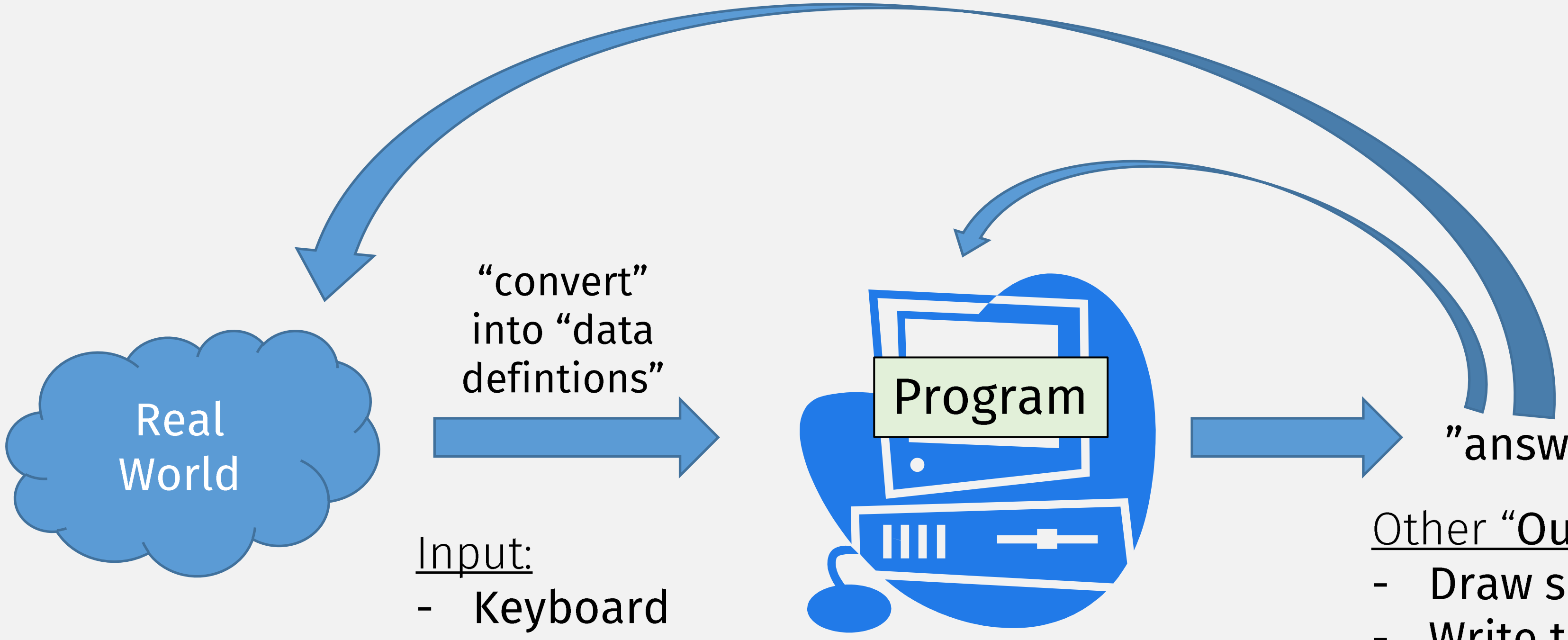
**Small Functions Considered Awesome**

Josh Saint Jacque · Follow
11 min read · Aug 22, 2017

Good <u>rule of thumb</u>:
A **function** should do **one, easily explainable task**

# Programs can be Interactive

More fun to write and use!




”answer”, e.g., 42

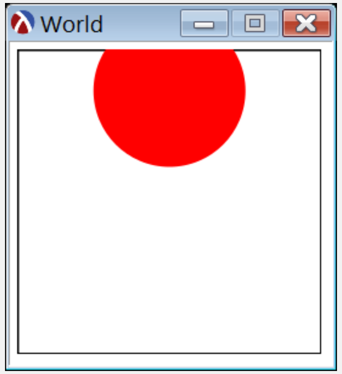
Input:
- Keyboard
- Mouse
- Gamepad
- Touchscreen
- Voice
- File

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



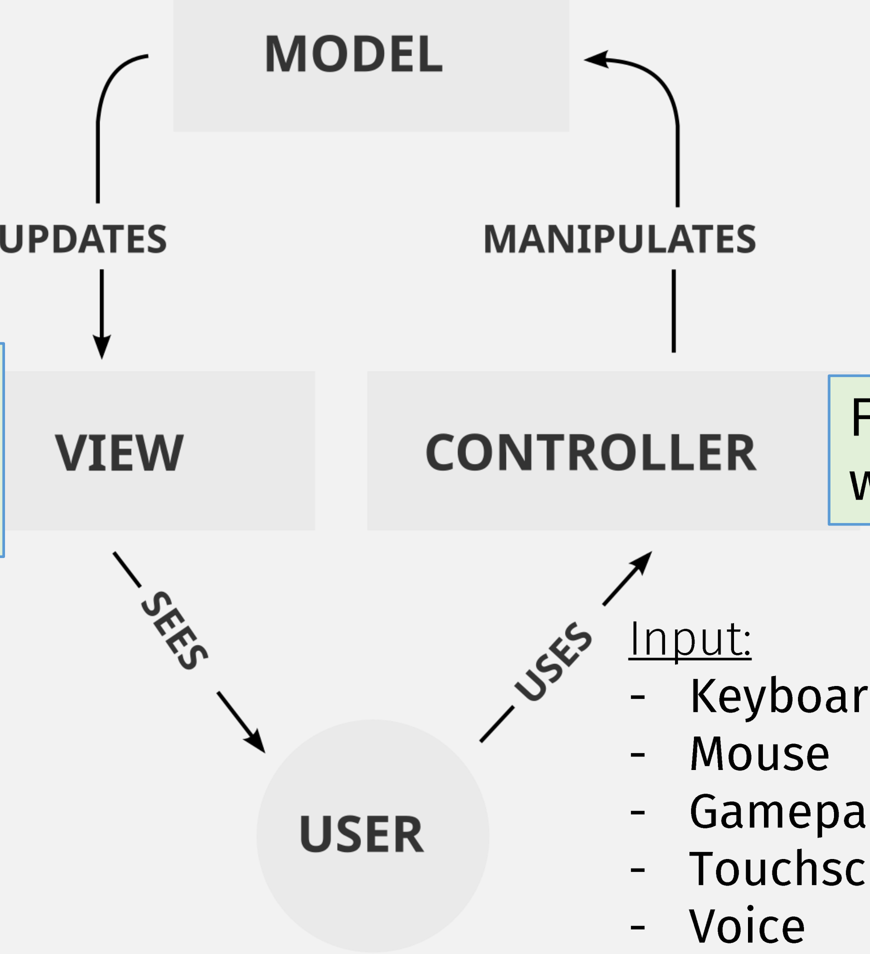
# Interactive Programs (with `big-bang`)

`(require 2htdp/universe)`

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

# Model-View-Controller (MVC) Pattern

“world state” **data definition!**




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

Functions that “update” world state data

Input:

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

from: **wikipedia.org**

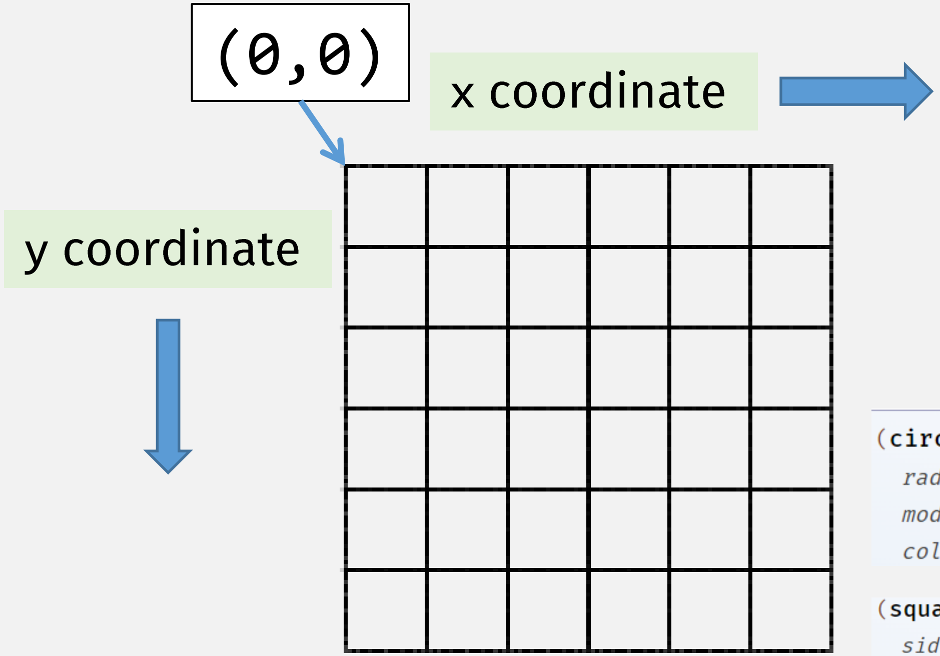
# Interactive Programs (with `big-bang`)

`(require 2htdp/universe)`

- `big-bang` starts an (MVC-like) interactive loop
  - repeatedly updates a "world state"
  - Programmer must 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
```

# *Interlude:* htdp universe coordinates




(**place-image** *image x y scene*) → image?  procedure

*image* : image?
*x* : real?
*y* : real?
*scene* : image?

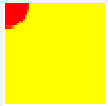
Places *image* onto *scene* with its center at the coordinates (*x*,*y*) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

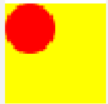
(**circle** *radius mode color*) → image?
*radius* : (and/c real? (not/c negative?))
*mode* : mode?
*color* : image-color?

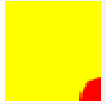
(**square** *side-len mode color*) → image?
*side-len* : (and/c real? (not/c negative?))
*mode* : mode?
*color* : image-color?

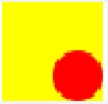
```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???

1 

2 

3 

4 

`(require 2htdp/universe)`

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

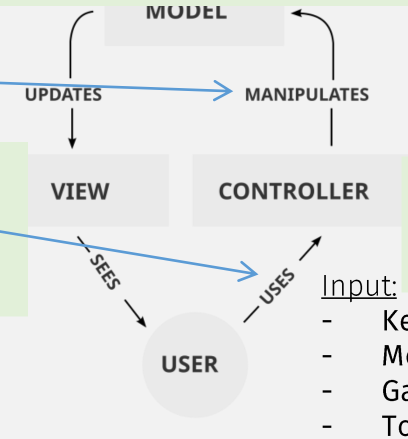
- big-bang starts an (MVC-like) interactive loop
  - repeatedly updates a "world state"
  - Programmer must define what the "World" is …
  - … with a Data Definition!

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

*Next time*

- 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