

CS450

High Level Languages

UMass Boston Computer Science

Tuesday, February 4, 2025



Logistics

- HW 0 in
 - ~~Due: Tue 2/4 11am EST~~
- HW 1 out
 - Due: Tue 2/11 11am EST
- Course web site:
 - Style: see “Racket Basics and Style”

<https://www.cs.umb.edu/~stchang/cs450/s25>



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))
```

Arithmetic ... More Th

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**

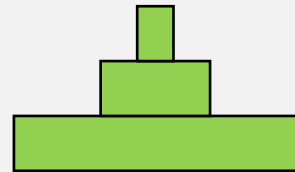
arithmetic expressions

(string-append “hi” “world”)

→ “hi-world”

(above   )

→

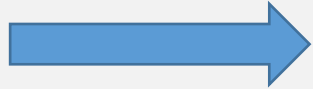


Programs Need Input

e.g., students



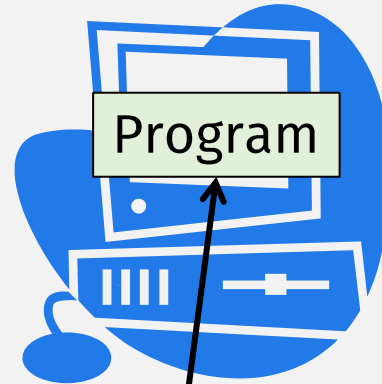
"convert"
into "data"



Input:

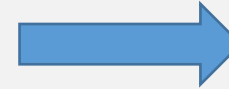
- Keyboard
- Mouse
- Gamepad
- Touchscreen
- Voice
- File

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



Do a "real
world" task

"run"
(evaluate)

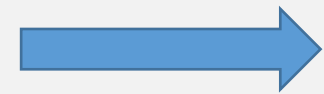


"answer", e.g., 42

Program vs Real World

Real World "things" ...

e.g., Temperature



Input:
- Keyboard
- Mouse

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



Program

When programming,
choosing data representations must be the first task!

(way before writing any code ...
which processes the data)

A Data Definition name

Specify possible values of the data

```
;; A TempC is an Integer  
;; Interpretation: It represents a  
temperature in degrees Celsius
```

Interpretation ... connects
data to a real world concept

```
;; A TempF is an Integer  
;; Interp: It represents a  
temperature in degrees Fahrenheit
```

```
;; A TempK is an non-negative Integer  
;; Interp: It represents a  
temperature in degrees Kelvin
```

Data Design Recipe

(Steps to follow)

A **predicate** for a data definition is a **function** that: evaluates to **true** when the given an **argument** is a value of the data definition

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

A Data Definition name

Specify possible values of the data

```
;; A TempC is an Integer  
;; Interpretation: It represents a  
temperature in degrees Celsius
```

Interpretation ... connects data to a real world concept

- 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**

Parts of a Data Definition

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

Function **Signature** uses Data Definitions
... to specify types of input and output data

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

```
;; A TempC is an Integer  
;; Interpretation: It represents a temperature in degrees Celsius
```

A **predicate** for a data definition is a **function** that:
evaluates to true when the given an **argument** is a **value** of the data definition

A data definition defines a new “type” of data

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

Design Recipe(s)

(Steps to follow when writing a program)

- Data Design

- • Function Design

Designing Functions

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

1. Name

```
;; c2f: TempC -> TempF
```

2. Signature

```
;; Converts a Celsius temperature to Fahrenheit
```

- # of arguments and their data type
- Output type
- May only reference “defined” Data Definition names

3. Description

Designing Functions

1. Name

```
;; c2f: TempC -> TempF
```

2. Signature

```
;; Converts a Celsius temperature to Fahrenheit
```

- # of arguments and their data type
- Output type
- May only reference “defined” Data Definition names

3. Description – shows how fn works, in English

4. Examples – shows how fn works, in code

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

5. Code

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

6. Tests

```
(check-equal? (c2f 1) (+ (/ 9 5) 32))
```

From testing framework (stay tuned!)

Designing Functions

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

1. Name

```
;; c2f: TempC -> TempF
```

2. Signature

```
;; Converts a Celsius temperature to Fahrenheit
```

- # of arguments and their data type
- Output type
- May only reference “defined” Data Definition names

3. Description – shows how fn works, in English

4. Examples – shows how fn works, in code

5. Code

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

6. Tests

```
(check-equal? (c2f 1) (+ (/ 9 5) 32))
```

Something is wrong!

- in Code?
- in Signature?
- in Data Definition?

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, praying it will “work” this time

Instead, **program incrementally**

The Incremental Programming Pledge

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 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

```
;; c2f: TempC -> TempF
```

2. Signature

```
;; Converts a Celsius temperature to Fahrenheit
```

- # of arguments and their data type
- Output type
- May only reference “defined” Data Definition names

3. Description

2. Start with “placeholder” code
(do not submit this!)

4. Examples

5. Code

6. Tests

```
(define (c2f ctemp)
  (case
    [(0) 32]
    [(100) 212]
    [(-40) -40]))
```

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)
```


Incremental Programming, in Action

1. Name

```
;; c2f: TempC -> TempF
```

2. Signature

```
;; Converts a Celsius temperature to Fahrenheit
```

- # of arguments and their data type
- Output type
- May only reference “defined” Data Definition names

3. Description

2. Start with “placeholder” code

1. Make Examples runnable tests

4. Examples

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

5. Code

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

6. Tests

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

Implies: Write small functions!

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

Conventional Wisdom: Write Small Functions

↔ Write Short Functions

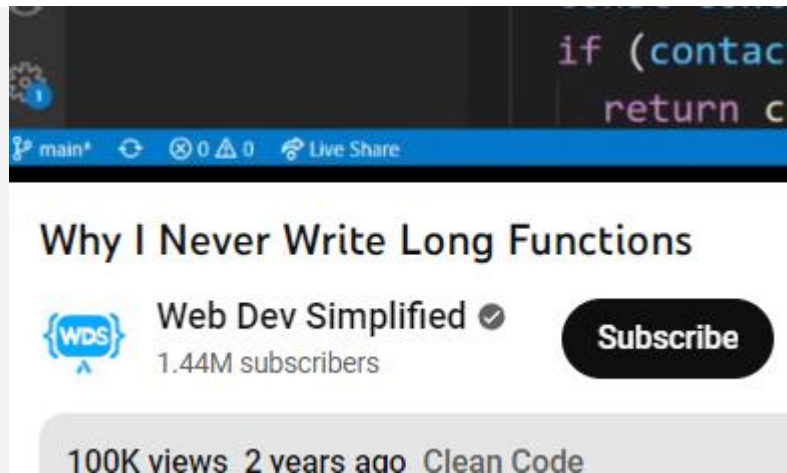
Google C++ Style Guide

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.



The image shows a YouTube video thumbnail. At the top, there's a code editor snippet with the text `if (contact` and `return c`. Below that, the video title is "Why I Never Write Long Functions" by "Web Dev Simplified" (WDS), which has 1.44M subscribers. A "Subscribe" button is visible. At the bottom, it says "100K views 2 years ago Clean Code".

Small Functions Considered Awesome



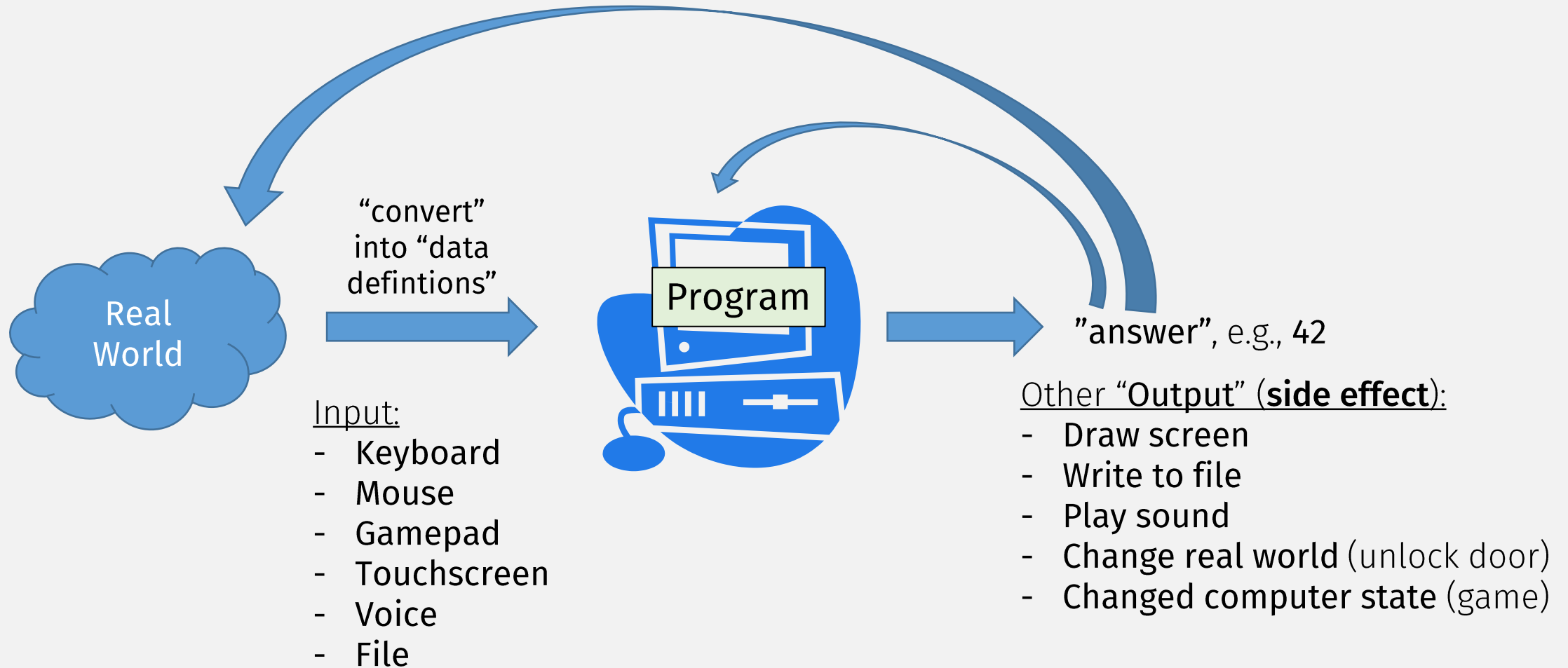
Josh Saint Jacques · Follow

11 min read · Aug 22, 2017

Good rule of thumb:
A function should do one, easily explainable task

Programs can be Interactive

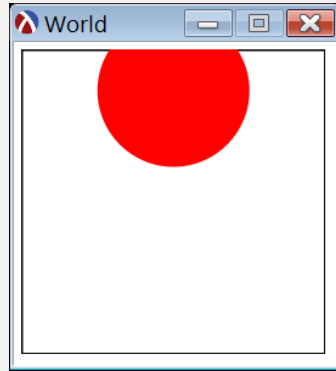
More fun to write and use!



(require 2htdp/universe)

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

- DEMO



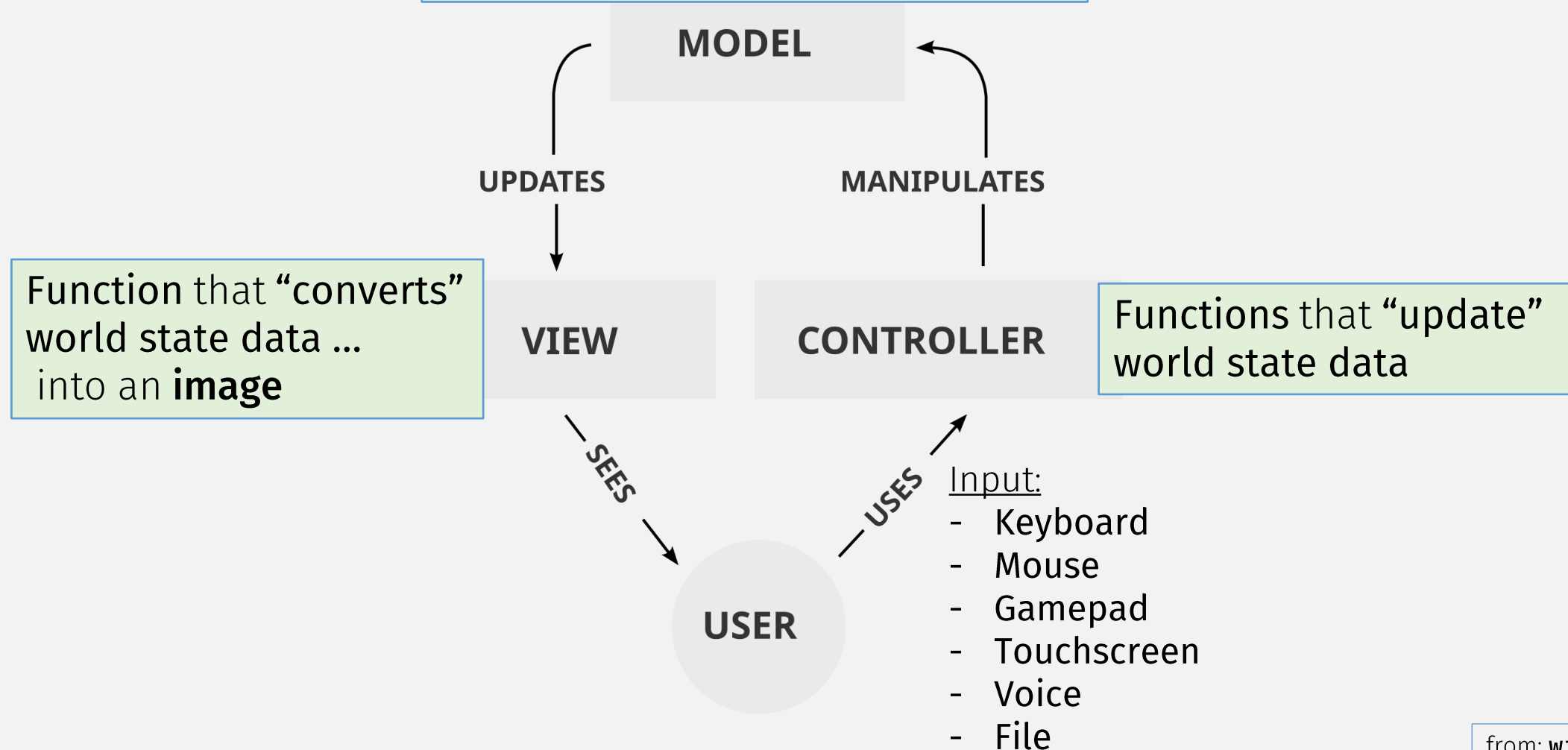
(require 2htdp/universe)

Interactive Programs (with `big-bang`)

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

Model-View-Controller (MVC) Pattern

“world state” data definition!



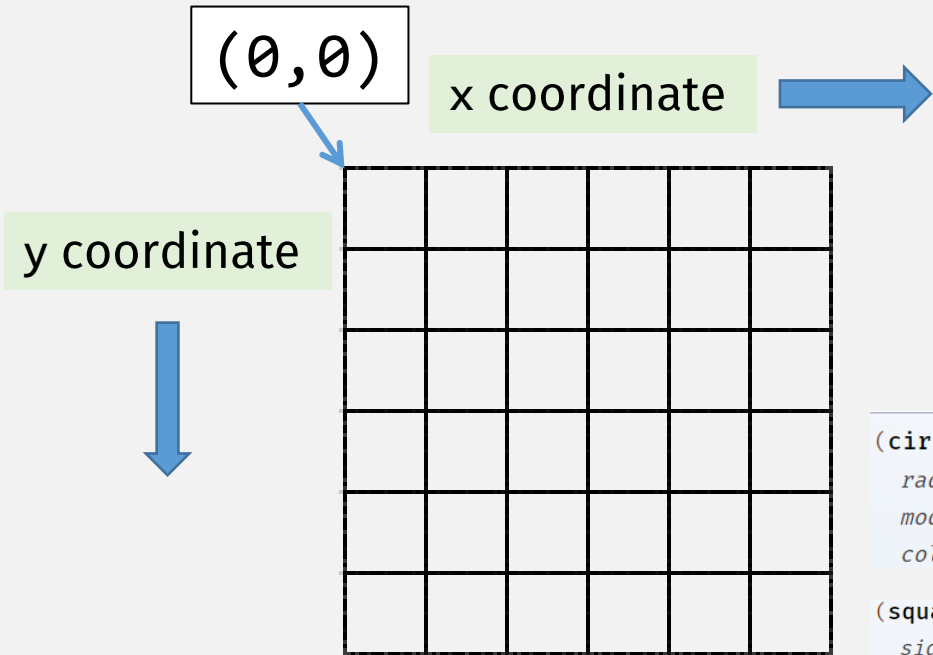
(require 2htdp/universe)

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  
;; Interp: 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?
```

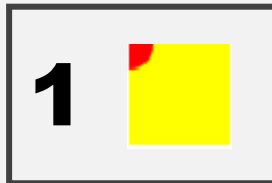
Places *image* onto *scene* with its center at the coordinates (x,y) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

```
(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"))
```

???



(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!

Next time

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

• Programmers specify “handler” functions to manipulate “World”

- Render
- World update
- Input handlers

