UMass Boston Computer Science

CS450 High Level Languages

Kinds of Data Definitions

Thursday, February 12, 2026

Logistics

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

- Course web site:
  - See The Design Recipe section

Last Time

Design Recipe, Step 1: Data Design

Create **Data Definitions**

- Describes the **types of data** that the program operates on
- Has **4** parts:
  1. **Name**
  2. Description of **all possible values** of the **data**
  3. **Interpretation** explaining the **real world concepts** the **data represents**
  4. **Predicate** returning **`false`** for (some) **values not** in the **Data Definition**

Kinds of Data Definitions

- Basic data
- Intervals
- Enumerations
- Itemizations

```
template<typename T>
class Array {
    //...
    int size;
    T* array;
    T &operator[](int index) {
        if(index >= size || index < 0)
            throw OUT_OF_RANGE; //#define OUT_OF_RANGE 0x0A
        return array[index];
    }
}
```

Interval Data Definitions

Is this what we want?

It depends (on our application)**!**
(there is no "correct" data def!)

Yet, **Data Definitions** are <u>crucial</u> because they **determine what the rest of the program looks like!**

```
;; An AngleD is a number in [0, 360)
;; Represents: An angle in degrees
(define (AngleD? deg)
  (and (>= deg 0) (< deg 360)))
```

```
;; An AngleR is a number in [0 2π)
;; Represents: An angle in radians
(define (AngleR? r)
  (and (>= r 0) (< r (* 2 pi))))
```

```
;; deg->rad: AngleD -> AngleR
;; Converts the given angle in degrees to radians
```

<u>Function Recipe Steps 1-3:</u>
name, signature, description

```
(check-equal? (deg->rad 0) 0)
(check-equal? (deg->rad 90) (/ pi 2))
(check-equal? (deg->rad 180) pi)
```

<u>Step 4:</u> **Examples**

Not allowed by data def!
… but should be ok?

```
(check-equal? (deg->rad 360) 0) ; ???
```

```
(check-equal? (deg->rad 360) (* 2 pi)) ; ???
```

```
(define/contract (deg->rad deg)
  (-> AngleD? AngleR?)
  (* deg (/ pi 180)))
```

<u>Step 5:</u> **Code**

<u>Step 6:</u> **Tests**

This is not the only possibility!

Rule of thumb:
1 **function** does
1 **task** which processes
1 **kind of data**

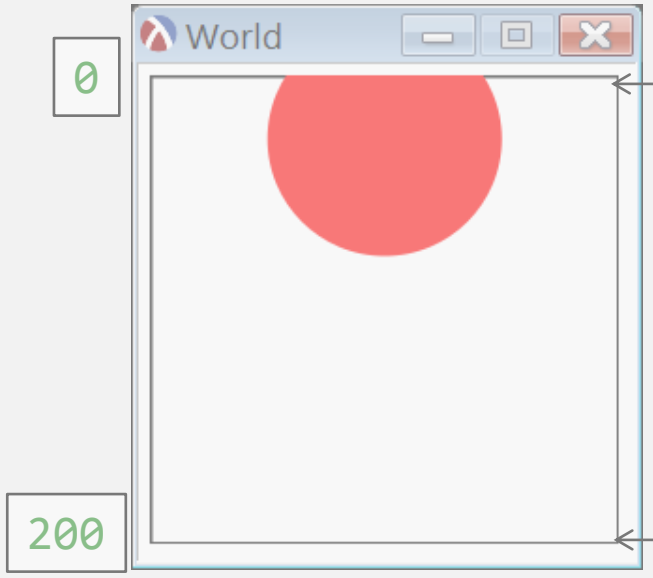
```
Non-neg Int in [SCENE-TOP, SCENE-BOT+CIRC-WIDTH]
~~Non-neg Int in [SCENE-TOP, SCENE-BOT]~~
~~Non-neg Int in [0,200]~~
;; A WorldState is a
;; Represents: bottom y-coordinate of
visible circle, in big-bang animation
```

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

Chosen Data Def affects code!

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

Be careful to use correct Data Definitions!




```
(define SCENE-TOP 0)
(define SCENE-BOT 200)
```

Readability: Write **helper functions** when converting between data types

```
(place-image
  some-img
  0 ~~(- y 100)~~
    (bot->center y)
  some-scene)
```

Kinds of Data Definitions

- Basic data
- Intervals
- Enumerations
- Itemizations

```
enum season { spring, summer, autumn, winter };
```

```
enum Colours {
    RED = 'RED',
    YELLOW = 'YELLOW',
    GREEN = 'GREEN'
}
```

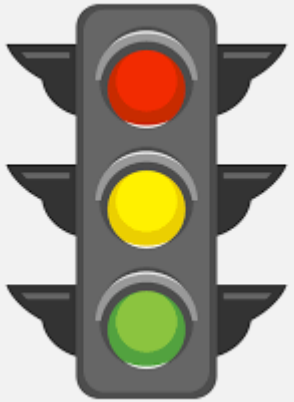
TypeScript

Enumeration Data Definitions

```
;; A TrafficLight is one of:
;; - RED-LIGHT
;; - GREEN-LIGHT
;; - YELLOW-LIGHT
;;   Represents:   possible colors of a traffic light
(define RED-LIGHT "RED")
(define GREEN-LIGHT "GREEN")
(define YELLOW-LIGHT "YELLOW")
```

constants

NOTE: this **is not the only possible data definition.** Can you think of a better one?



```
(define (RED-LIGHT? x) (equal? x RED-LIGHT))
(define (GREEN-LIGHT? x) (equal? x GREEN-LIGHT))
(define (YELLOW-LIGHT? x) (equal? x YELLOW-LIGHT))
```

```
(define (TrafficLight? x)
 (or (RED-LIGHT? x)
     (GREEN-LIGHT? x)
     (YELLOW-LIGHT? x)))
```

Need to **add an extra step** to **Data Design Recipe**

Design Recipe, Step 1: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has **4** parts:
  1. **Name**
  2. Description of **all possible values** of the data
  3. **Interpretation** explaining the real world concepts the data represents
  4. **Predicate** returning `false` for (some) **values not** in the **Data Definition**

- (If needed) **define extra predicates** for **each enumeration** or **itemization** value (some languages do this implicitly for you, Racket does not)

Enumeration Data Definitions

`cond` is only allowed in functions that process enumeration (or itemization) data!

```
;; A TrafficLight is one of:
;; - RED-LIGHT
;; - GREEN-LIGHT
;; - YELLOW-LIGHT
;;   Represents:  possible colors of a traffic light
(define RED-LIGHT "RED")
(define GREEN-LIGHT "GREEN")
(define YELLOW-LIGHT "YELLOW")
```

The **data** and **function** have the **same structure**!

(keep order the same)

```
;; next-light: TrafficLight -> TrafficLight
;; Computes the next light after the given one
(check-equal? (next-light RED-LIGHT) GREEN-LIGHT)
(check-equal? (next-light GREEN-LIGHT) YELLOW-LIGHT)
(check-equal? (next-light YELLOW-LIGHT) RED-LIGHT)

(define (next-light light)
  (cond
    [(RED-LIGHT? light) GREEN-LIGHT]
    [(GREEN-LIGHT? light) YELLOW-LIGHT]
    [(YELLOW-LIGHT? light) RED-LIGHT]))
```

Function Recipe Steps 1-3: name, signature, description

Step 4: Examples

cond is multi-arm `if` (expression)

Step 5: Code

Designing data first makes **writing function** (code) **easier**!

Last Time

Function Design Recipe

1. **Name**

2. **Signature** – types of the **function input(s)** and **output**

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

4. **Examples** – show the **function behavior**

5. **Code** – implement the **rest of the function** (arithmetic)

6. **Tests** – check (using `#lang racket450/testing`) the **function behavior**

Function Design Recipe

1. **Name**
2. **Signature** – types of the **function input(s)** and **output**
3. **Description** – explain (in **English prose**) the **function behavior**
4. **Examples** – show the **function behavior**
5. **Template** – sketch out the **function structure** (using input's **Data Definition**)
6. **Code** – implement the **rest of the function** (arithmetic)
7. **Tests** – check (using `#lang racket450/testing`) the **function behavior**

Enumeration Data Definitions

data definition tells **you** what the **code** (that processes that type data) **will look like**

```
;; A TrafficLight is one of:
;; - RED-LIGHT
;; - GREEN-LIGHT
;; - YELLOW-LIGHT
;;   Represents:  possible colors of a traffic light
(define (RED-LIGHT? x) (equal? x RED-LIGHT))
(define (GREEN-LIGHT? x) (equal? x GREEN-LIGHT))
(define (YELLOW-LIGHT? x) (equal? x YELLOW-LIGHT))
```

A **function's template** is completely **determined** by the **input's Data Definition**

```
;; next-light: TrafficLight -> TrafficLight
;; Computes the next light after the given one
(define (next-light light)
  (cond
    [(RED-LIGHT? light) ....]
    [(GREEN-LIGHT? light) ....]
    [(YELLOW-LIGHT? light) ....]))
```

(keep order the same)

Step 5: ~~Code~~ **Template**

Step 6: **Code** (fill in the "...." with arithmetic)

Some Pre-defined Enumerations

```
; A KeyEvent is one of:
; – 1String
; – "left"
; – "right"
; – "up"
; – ...
```

```
; A MouseEvt is one of these Strings:
; – "button-down"
; – "button-up"
; – "drag"
; – "move"
; – "enter"
; – "leave"
```

```
; WorldState KeyEvent -> ...
(define (handle-key-events w ke)
  (cond
    [(= (string-length ke) 1) ...]
    [(string=? "left" ke) ...]
    [(string=? "right" ke) ...]
    [(string=? "up" ke) ...]
    [(string=? "down" ke) ...]
    ...))
```

Template

Or even better: `key=?`

```
;; handle-mouse: WorldState Coordinate Coordinate MouseEvt -> WorldState
;; Produces the next WorldState
;; from the given Worldstate, mouse position, and mouse event
(define (handle-mouse w x y evt)
  (cond
    [(string=? evt "button-down") ....]
    [(string=? evt "button-up") ....]
    [else ....]))
```

Or even better: `mouse=?`

Design Recipe allows <u>combining cases</u> if they are **handled the same**

```
; A 1String is a String of length 1,
; including
; – "\\" (the backslash),
; – " " (the space bar),
; – "\t" (tab),
; – "\r" (return), and
; – "\b" (backspace).
; Represents: keys on the keyboard
```

Kinds of Data Definitions

- Basic data
- Intervals
- Enumerations
- Itemizations

(Generalized enumeration)

Itemization Data Definitions (Generalized enumeration)

2026 tax brackets

Tax Rate	For Single Filers	For Married Individuals Filing Joint Returns	For Heads of Households
10%	$0 to $12,400	$0 to $24,800	$0 to $17,700
12%	$12,401 to $50,400	$24,801 to $100,800	$17,701 to $67,450
22%	$50,401 to $105,700	$100,801 to $211,400	$67,451 to $105,700
24%	$105,701 to $201,775	$211,401 to $403,550	$105,701 to $201,775
32%	$201,776 to $256,225	$403,551 to $512,450	$201,776 to
35%	$256,226 to $640,600	$512	
37%	$640,601 or more	$768,701 or more	$640,601 or

Source: IRS

```
;; A Salary is one of:
;; [0, 12400]
;; [12401 50400]
;; [50401 105700]
;; ...
;; Represents: Salary in USD,
;;             split by 2026 tax bracket
(define (10%-bracket? salary)
  (and (>= salary 0) (<= salary 12400))
(define (12%-bracket? salary)
  (and (>= salary 12401) (<= salary 50400))
;; ...
```

The **data** and **function** have the **same structure**!

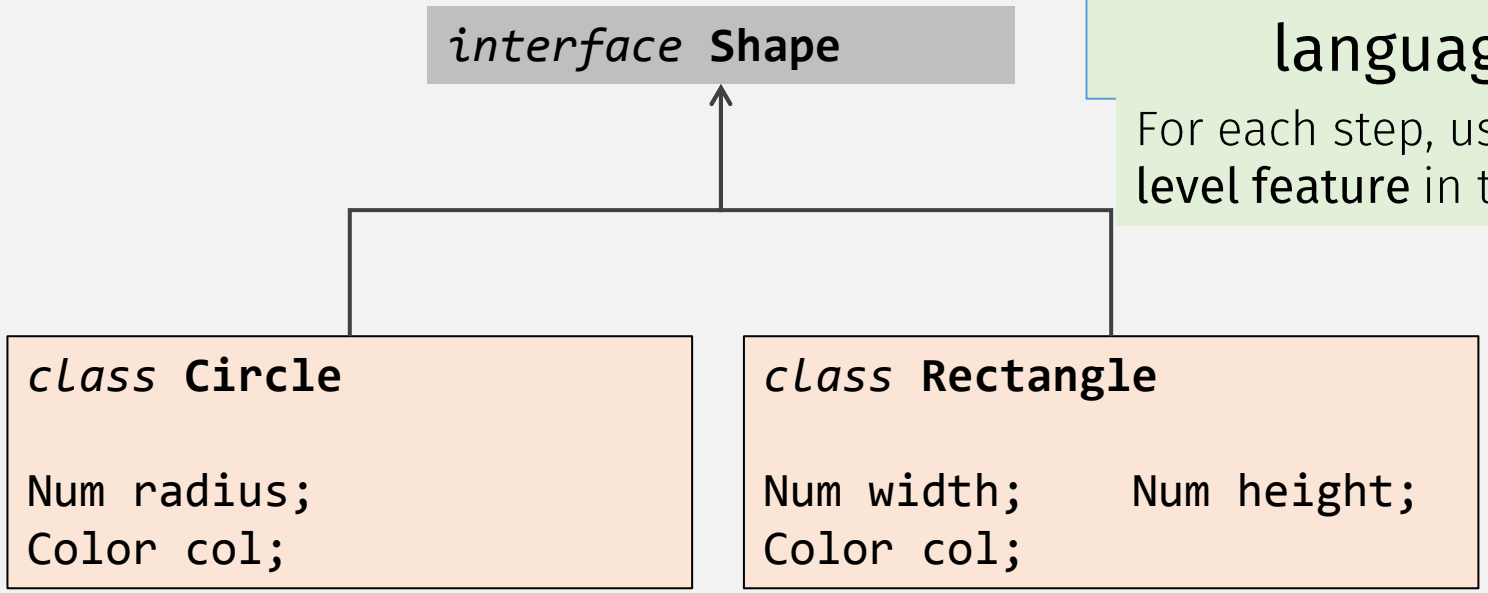
`else` is fallthrough case

```
;; taxes-owed: Salary -> USD
;; computes federal income tax owed in 2026
(define (taxes-owed salary)
  (cond
   [(10%-bracket? salary) ....]
   [(12%-bracket? salary) ....]
   [else ....]))
```

“Itemization” Data Def in Other Languages

The **Design Recipe** is language-agnostic

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




```
;; A Shape is one of:
;; - (rect Num Num Color)
;; - (circ Num Color)
;; Represents: a shape to be drawn on a canvas
;; - rect: fields are width, height, color
;; - circ: fields are radius, color
```

As a **Data Definition** in this course

Itemization Caveats

```
;; A MaybeInt is one of:
(define NaN "Not a Number")
;; or, Integer
;; Represents: a number, but with possible error case
```

`NaN` is a property of the *global object*. In other words, it is a variable in global scope.

In modern browsers, `NaN` is a non-configurable, non-writable property. Even when this is not the case, avoid overriding it.

References › JavaScript › Reference › Standard built-in objects › NaN

There are five different types of operations that return `NaN`: mdn web docs

- Failed number conversion (e.g. explicit ones like `parseInt("blabla")`, `Number(undefined)`, or implicit ones like `Math.abs(undefined)`)
- Math operation where the result is not a real number (e.g. `Math.sqrt(-1)`)
- Indeterminate form (e.g. `0 * Infinity`, `1 ** Infinity`, `Infinity / Infinity`, `Infinity - Infinity`)
- A method or expression whose operand is or gets coerced to `NaN` (e.g. `7 ** NaN`, `7 * "blabla"`) — this means `NaN` is contagious
- Other cases where an invalid value is to be represented as a number (e.g. an invalid Date `new Date("blabla").getTime()`, `"".charCodeAt(1)`)

`NaN` and its behaviors are not invented by JavaScript. Its semantics in floating point arithmetic (including that `NaN !== NaN`) are specified by IEEE 754. `NaN`'s behaviors include:

- If `NaN` is involved in a mathematical operation (but not bitwise operations), the result is also `NaN`. (See counter-example below.)
- When `NaN` is one of the operands of any relational comparison (`>`, `<`, `>=`, `<=`), the result is always `false`.
- `NaN` compares unequal (via `==`, `!=`, `===`, and `!==`) to any other value — including to another `NaN` value.

Itemization Caveats

```
;; A MaybeInt is one of:
(define NaN "Not a Number")
;; or, Integer
;; Represents: a number, but with possible error case
```

OR: **modify the data def!**
More **common cases should go first!**

```
;; better predicate for MaybeInt
(define (MaybeInt? x)
  (or (integer? x)
      (and (string? x) (NaN? x))))
```

```
(define (NaN? x)
  (string=? x "Not a Number"))
```

```
;; WRONG predicate for MaybeInt
#;(define (MaybeInt? x)
  (or (NaN? x)
      (integer? x)))
```

```
> (MaybeInt? 1)
string=?: contract violation
  expected: string?
  given: 1
```

```
;; OK predicate for MaybeInt
(define (MaybeInt? x)
  (or (and (string? x) (NaN? x))
      (integer? x)))
```

```
; WRONG TEMPLATE for MaybeInt
#;(define (maybeint-fn x)
  (cond
    [(NaN? x) ....]
    [(integer? x) ....]))
```

```
; OK TEMPLATE for MaybeInt
(define (maybeint-fn x)
  (cond
    [(string? x) ....]
    [(integer? x) ....]))
```

```
;; better TEMPLATE
(define (maybeint-fn x)
  (cond
    [(integer? x) ....]
    [else ....]))
```

Inside the function, **we only need to distinguish between valid input cases**

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 **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: Real-World Example

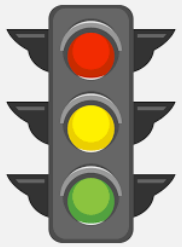
- https://www.youtube.com/watch?v=1SlGgCxJa3w
- "when you do everything at once …
  you're not sure why it's not working!"
- "when you layer it, when you break it down …
  and you hit a spot when it's not working …
  then you can just focus on that spot!"

5. Code
6. Tests

Make small code changes only (something easy to revert)

Test each (small) change (before making another one)

In-class exercise: Template practice



Data Definition choice?

- Pros?
- Cons?

```
;; A TrafficLight is one of:
(define RED-LIGHT "RED")
(define GREEN-LIGHT "GREEN")
(define YELLOW-LIGHT "YELLOW")
;;                Represents:   possible colors of a traffic light
```

TASK 1:

Find **Template** for **`TrafficLight`** Data Def

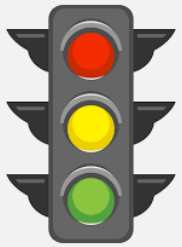
TASK 2:

Write **Template** for **`TrafficLight2`** Data Def

```
;; A TrafficLight2 is one of:
(define GREEN-L 0)
(define YELLOW-L 1)
(define RED-L 2)
;;         Represents:  a traffic light state
```

Submit to Gradescope

In-class exercise 2: **`big-bang`** practice



- Create a **`big-bang`** traffic light **simulator** that changes on a mouse click (“**`button-down`**” event)

```
;; A TrafficLight is one of:
(define RED-LIGHT "RED")
(define GREEN-LIGHT "GREEN")
(define YELLOW-LIGHT "YELLOW")
;;                Represents:   possible colors of a traffic light
```

```
;; A TrafficLight2 is one of:
(define GREEN-L 0)
(define YELLOW-L 1)
(define RED-L 2)
;;         Represents:  a traffic light state
```

Submitting

1. File: **`in-class-02-12-<Lastname>-<Firstname>.rkt`**
2. Join the in-class team: **`cs450s26/teams/in-class`**
3. Commit to repo: **`cs450s26/in-class-02-12`**
   - (May need to **`merge/pull + rebase`** if someone **push**es before you)