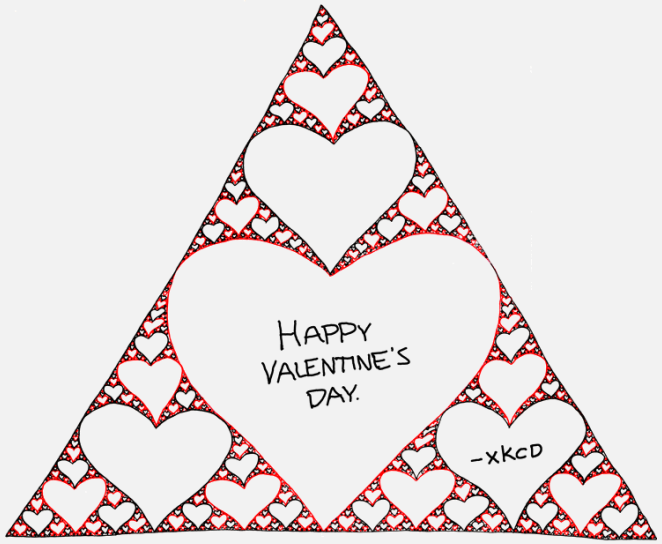
UMass Boston Computer Science

CS450 High Level Languages

Kinds of Data Definitions

Thursday, February 13, 2025



Logistics

- HW 2 out
  - due: Tues 2/18 11am EST

- Course web site:
  - See The Design Recipe section
  - Lecture code (see lecture04.rkt) may occasionally be posted

STYLE notes: 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

- 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")
```

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` if given value is 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)
;; interp: An angle in degrees
(define (AngleD? deg)
  (and (>= deg 0) (< deg 360)))
```

```
;; An AngleR is a number in [0 2π)
;; interp: 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

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

<u>Step 5:</u> Code

```
(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)) ; ???
```

<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**

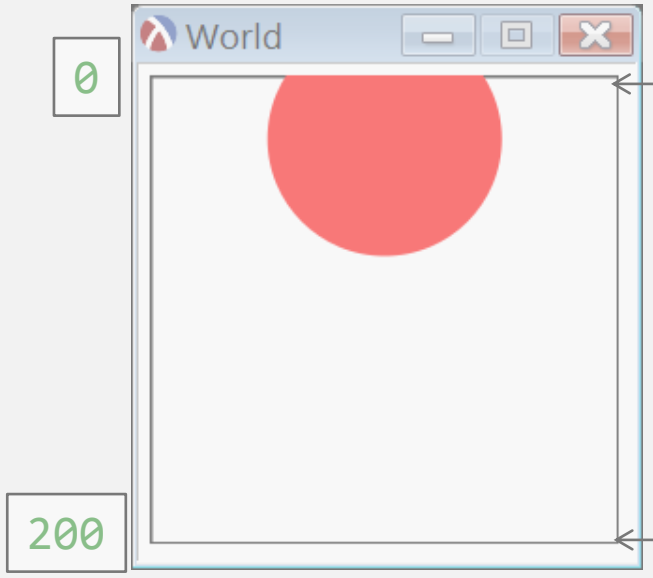
```
Non-neg Int in [SCENE-TOP, SCENE-BOT+CIRC-WIDTH]
~~Non-neg Int in [SCENE-TOP, SCENE-BOT]~~
;; A WorldState is a ~~Non-neg Int in [0,200]~~
;; Represents: bottom y-coordinate of
visible circle, in big-bang animation
```

```
(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)
```

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'
}
```

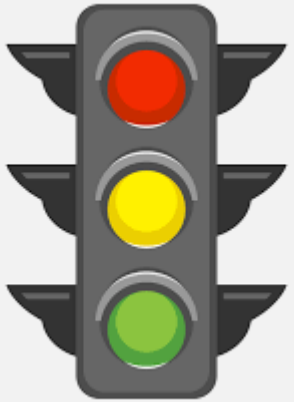
TypeScript

Enumeration Data Definitions

```
;; A TrafficLight is one of:
;; - RED-LIGHT
;; - GREEN-LIGHT
;; - YELLOW-LIGHT
;; Interpretation: 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.** Is there a better one?



```
(define (red-light? x) (string=? x RED-LIGHT))
(define (green-light? x) (string=? x GREEN-LIGHT))
(define (yellow-light? x) (string=? 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` if given value is not in the Data Definition
  - If needed, **define extra predicates for each enumeration** or **itemization** (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
;; Interpretation: 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
(define (next-light light)
  (cond
    [(red-light? light) GREEN-LIGHT]
    [(green-light? light) YELLOW-LIGHT]
    [(yellow-light? light) RED-LIGHT]))
```

cond is multi-arm `if` (expression)

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

Step 5: Code

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

```
(check-equal? (next-light RED-LIGHT) GREEN-LIGHT)
(check-equal? (next-light GREEN-LIGHT) YELLOW-LIGHT)
(check-equal? (next-light YELLOW-LIGHT) RED-LIGHT)
```

Step 4: Examples

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 (using `rackunit`) the **function behavior**

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

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

Enumeration Data Definitions

```
;; A TrafficLight is one of:
(define RED-LIGHT "RED")
(define GREEN-LIGHT "GREEN")
(define YELLOW-LIGHT "YELLOW")
;; Interpretation: Represents possible colors of a traffic light
(define (red-light? x) (string=? x RED-LIGHT))
(define (green-light? x) (string=? x GREEN-LIGHT))
(define (yellow-light? x) (string=? 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"
```

Template

```
; 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) ...]
    ...))
```

```
;; 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 ....]))
```

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).
; interpretation represents keys on the keyboard
```

Kinds of Data Definitions

- Basic data
- Intervals
- Enumerations
- Itemizations

(Generalized enumeration)

Itemization Data Definitions (Generalized enumeration)

2025 tax brackets

Tax rate	Single	Married filing jointly	Married filing separately	Head of household
10%	$0 to $11,925	$0 to $23,850	$0 to $11,925	$0 to $17,000
12%	$11,926 to $48,475	$23,851 to $96,950	$11,926 to $48,475	$17,001 to $64,850
22%	$48,476 to $103,350	$96,951 to $206,700	$48,476 to $103,350	$64,851 to $103,350
24%	$103,351 to $197,300	$206,701 to $394,600	$103,351 to $197,300	$103,351 to $197,300
32%	$197,301 to $250,525	$394,601 to $501,050	$197,301 to	$197,301 to
35%	$250,526 to $626,350	$501,051 $751,600		
37%	$626,351 or more	$751,601 or more	$375,801 or m	$626,351 or

Source: IRS.

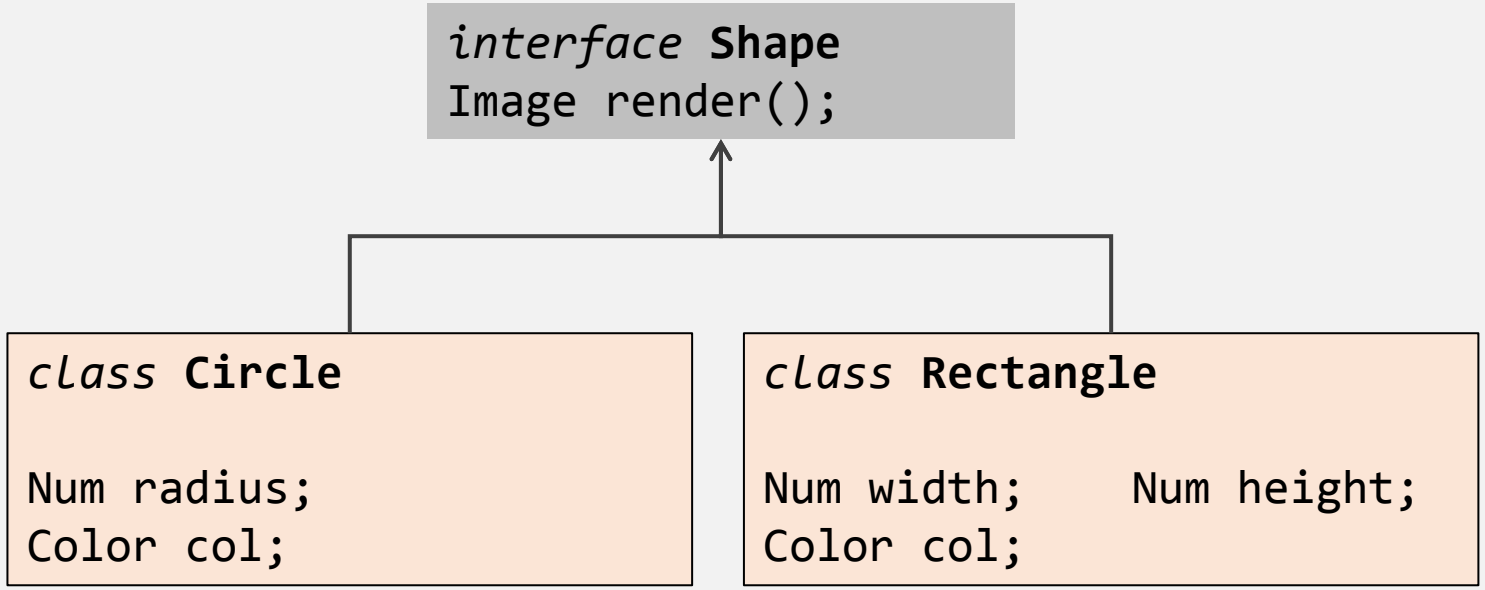
```
;; A Salary is one of:
;; [0, 11925]
;; [11926 48475]
;; [48476 103350]
;; ...
;; Interp: Salary in USD,
;;         split by 2025 tax bracket
(define (10%-bracket? salary)
  (and (>= salary 0) (<= salary 11925))
(define (12%-bracket? salary)
  (and (>= salary 11926) (<= salary 48475))
;; ...
```

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

`else` is fallthrough case

```
;; taxes-owed: Salary -> USD
;; computes federal income tax owed in 2025
(define (taxes-owed salary)
  (cond
   [(10%-bracket? salary) ....]
   [(12%-bracket? salary) ....]
   [else ....]))
```

“Itemization” Data Def in Other Languages



As a Data Definition

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

Itemization Caveats

```
;; A MaybeInt is one of:
(define NaN "Not a Number")
;; or, Integer
;; Interp: represents a number with a 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

mdn web docs

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

- 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

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

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

```
;; 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)))
```

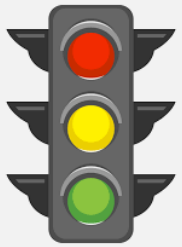
```
; WRONG TEMPLATE for MaybeInt
#;(define (maybeint-fn x)
  (cond
    [(NaN? x) ....]
    [(integer? x) ....]))
```

```
; 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**

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")
;; Interpretation: Represents possible colors of a traffic light
```

TASK 1:

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

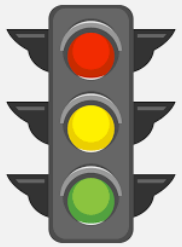
TASK 2:

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

```
;; A TrafficLight2 is one of:
(define GREEN-L 0)
(define YELLOW-L 1)
(define RED-L 2)
;; Interp: 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")
;; Interpretation: Represents possible colors of a traffic light
```

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

Submitting

1. File: **`in-class-02-13-<Lastname>-<Firstname>.rkt`**
2. Join the in-class team: **`cs450s25/teams/in-class`**
3. Commit to repo: **`cs450s25/in-class-02-13`**
   - (May need to **`merge/pull + rebase`** if someone **`push`**es before you)