UMass Boston Computer Science

CS450 High Level Languages

Programming with Compound Data

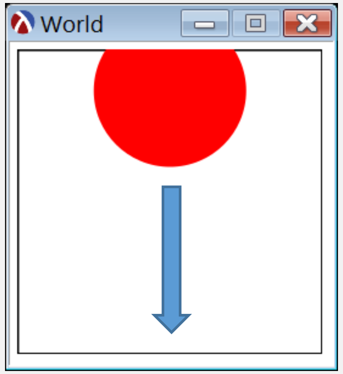
Thursday, February 20, 2025

Logistics

- HW 3 out
  - due: Tuesday 2/25, 11am EST

Last Time

Falling "Ball" Example



What if the **ball can also move side-to-side**?

`WorldState` would need two **pieces of data**:
the **x** *and* **y coordinates**

Last Time

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center coordinate
;; - y is vertical center coordinate
(struct world [x y] #:transparent)
```

```
(check-equal?
  (next-WorldState
    (mk-WorldState 0 0))
  (mk-WorldState X-VEL Y-VEL))
```

```
;; next-WorldState : WorldState -> WorldState
;; Computes the ball position after 1 tick
```

```
;; TEMPLATE for WorldState-fn: WorldState -> ???
(define/contract (WorldState-fn w)
  (-> WorldState? ??? )
  .... (world-x w) ....
  .... (world-y w) .... )
```

Template?

Template for compound data extracts the pieces ...

Last Time

```
(check-equal?
  (next-WorldState
    (mk-WorldState 0 0))
  (mk-WorldState X-VEL Y-VEL))
```

```
;; don’t need Signature, if redundant with contract
~~;; next-WorldState : WorldState -> WorldState~~
;; Computes the ball position after 1 tick
```

```
(define/contract (next-WorldState w)
  (-> WorldState? WorldState?)
  .... (world-x w) ....
  .... (world-y w) .... )
```

Last Time

```
(check-equal?
  (next-WorldState
    (mk-WorldState 0 0))
  (mk-WorldState X-VEL Y-VEL))
```

```
;; Computes the ball position after 1 tick
```

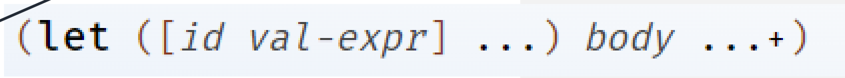
```
(define/contract (next-WorldState w)
  (-> WorldState? WorldState?)
  (mk-WorldState
    (+ (world-x w) X-VEL)
    (+ (world-y w) Y-VEL)))
```

Last Time

Extract Compound Pieces – **`let`**

Extract all compound data pieces first, before doing “arithmetic”

```
(define/contract (next-WorldState w)
  ; ...
  (let ([x (world-x w)]
        [y (world-y w)])
    (mk-WorldState (+ x X-VEL) (+ y Y-VEL))))
```



```
(let ([id val-expr] ...) body ...+)
```

Defines new local variables

Local variables **shadow** previously defined vars

in scope only in the body

Last Time

Extract Compound Pieces – (internal) **define**

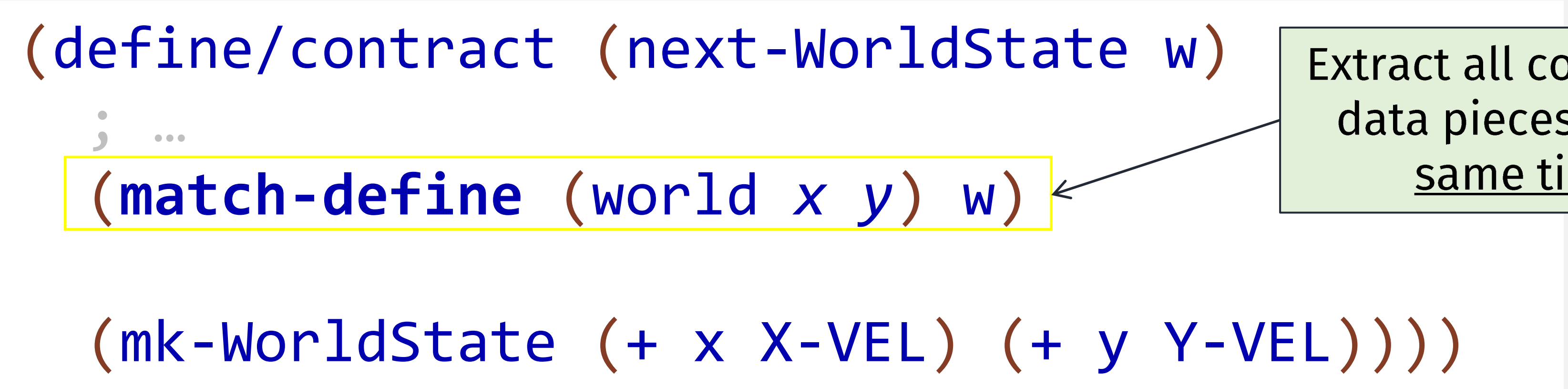
```
(define/contract (next-WorldState w)
  ; …
  (define x (world-x w))
  (define y (world-y w))
  (mk-WorldState (+ x X-VEL) (+ y Y-VE
```

Extract all compound data pieces first, before doing “arithmetic”

(is there an easier way to do this?)

Last Time

Extract Compound Pieces – Pattern Match!



```
(define/contract (next-WorldState w)
  ; …
  (match-define (world x y) w)

  (mk-WorldState (+ x X-VEL) (+ y Y-VEL))))
```

Extract all compound data pieces, at the same time!

Extract Compound Pieces – Pattern Match!

Do we need separate “coordinate processing” functions?

MAYBE!

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int])
;; Represents coordinate in big-bang animation where:
;; - x is ball (red solid circle) horizontal center
;; - y is ball vertical center
```

WAIT

1 function does
1 task which processes
1 kind of data

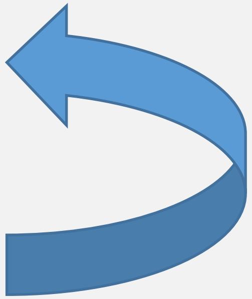
```
(define/contract (next-WorldState w)
  ; …
  (match-define (world x y) w)

  (mk-WorldState (+ x X-VEL) (+ y Y-VEL))))
```

Is this function doing too much?

Program Design Recipe

... is iterative!

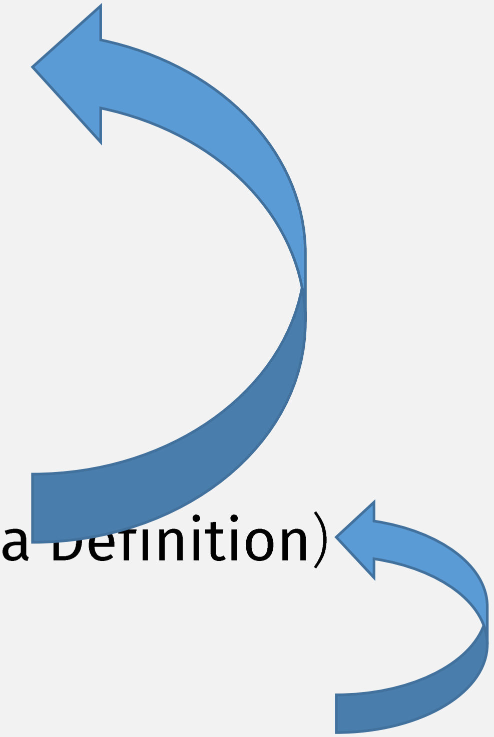
1. **Data Design**
2. **Function Design**



Function Design Recipe

... is iterative!

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



Bigger Compound Data

What if the “velocity” is not constant?

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center
;; - y is vertical center
```

Bigger Compound Data

What if the “velocity” is not constant?

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int] [xv : Int] [yv : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center
;; - y is vertical center
;; - xv is horizonal velocity
;; - yv is vertical velocity

(struct world [x y xv yv] #:transparent)
```

```
;; TEMPLATE for WorldState-fn: WorldState -> ???
(define (WorldState-fn w)
  .... (world-x w) ....
  .... (world-y w) ....
  .... (world-xv w) ....
  .... (world-yv w) .... )
```

Template?

Bigger Compound Data

What if the “velocity” is not constant?

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int] [xv : Int] [yv : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center
;; - y is vertical center
;; - xv is horizonal velocity
;; - yv is vertical velocity

(struct world [x y xv yv] #:transparent)
```

```
;; TEMPLATE for WorldState-fn: WorldState -> ???
(define (WorldState-fn w)
  (match-define (world x y xv yv) w)
  ....

                              )
```

Template?

Bigger Compound Data

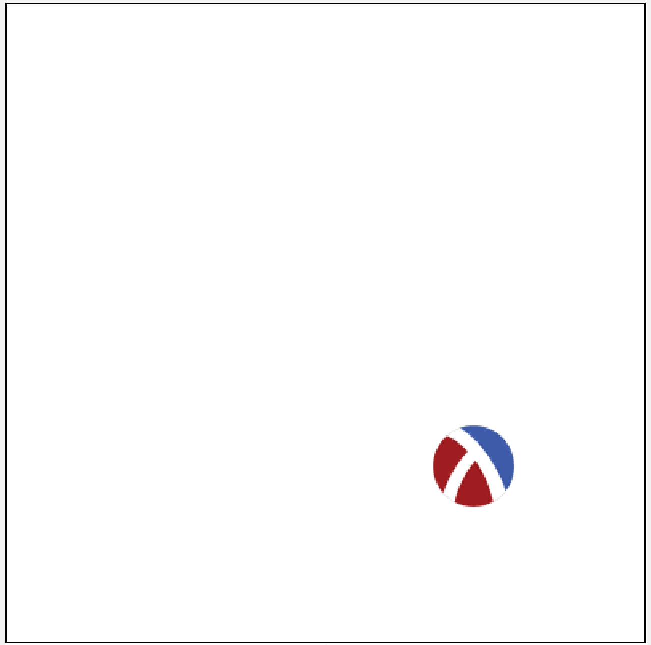
What if the “velocity” is not constant?

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int] [xv : Int] [yv : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center
;; - y is vertical center
;; - xv is horizonal velocity
;; - yv is vertical velocity

(struct world [x y xv yv] #:transparent)
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)

  (mk-WorldState (+ x xv) (+ y yv) xv yv))
```

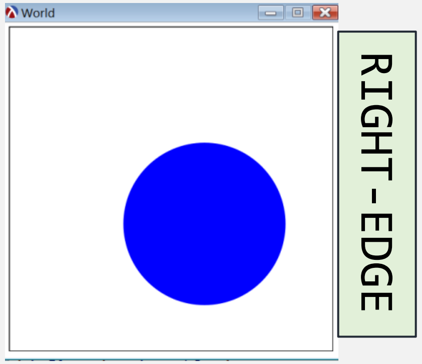
What if velocity can change?

Bouncing Ball



Velocity “reverses” when edge is hit

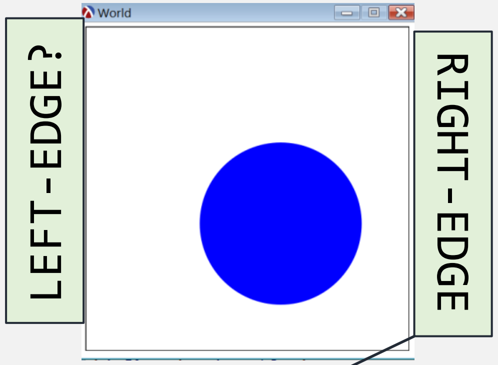
Make it bounce?



```
;; next-WorldState : WorldState -> WorldState
;; Computes the next ball pos
```

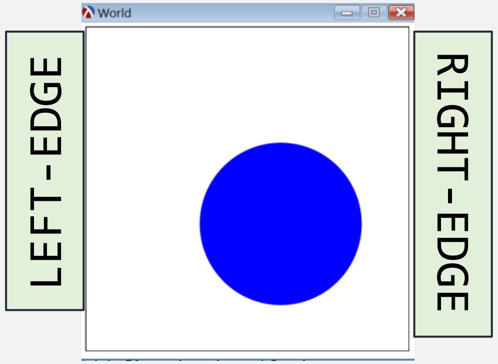
```
(define (next-WorldState w)
  (match-define (world x y xv yv) w)




  (mk-WorldState (+ x xv) (+ y yv) xv yv))
```

Make it bounce?



```
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (define new-xv
    (if (>= x RIGHT-EDGE) (- xvel) xvel))

  (mk-WorldState (+ x xv) (+ y yv) new-xv yv))
```

Make it bounce?



```
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (define new-xv
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (mk-WorldState (+ x xv) (+ y yv) new-xv yv))
```

Make it bounce?

```
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (define new-xv
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (mk-WorldState (+ x new-xv) (+ y yv) new-xv yv))
```

Should this be **xv** or **new-xv**???

Make it bounce?

(no tests!)

```
(define (next-WorldState
  (match-define (world x
  (define new-xv
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (
  (define new-yv???
    (if (or (>= y BOTTOM-EDGE)
```

Keep hacking and hope that it works???

DON'T PROGRAM LIKE THIS!!!

If you're **no longer following the template,** then the **Data Definitions need updating!**

This is undisciplined programming
It is slower and error-prone. Think first!

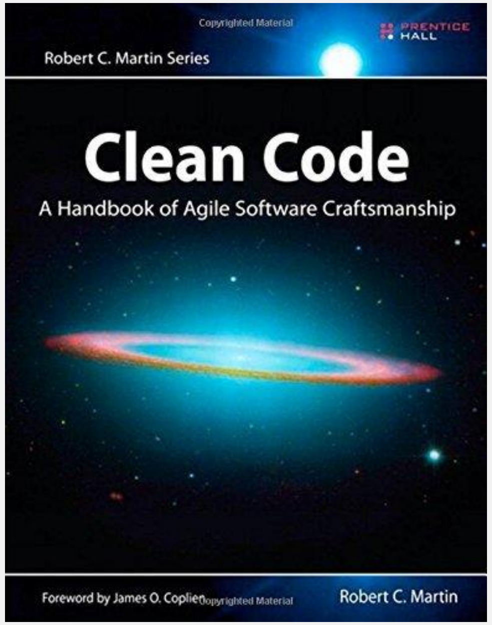
HW Advice

“Perhaps you thought that “**getting it working**” was the first order of business for a professional developer.

I hope by now, however, that this book has disabused you of that idea.

The functionality that you create today has a good chance of changing in the next release, but the **readability of your code** will have a profound effect on all the changes that will ever be made.”

— Robert C. Martin,
Clean Code: A Handbook of Agile Software Craftsmanship



Process One Kind of Data at a Time

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int] [xv : Int] [yv : Int])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; - x is horizontal center
;; - y is vertical center
;; - xv is horizontal velocity
;; - yv is vertical velocity

(struct world [x y xv yv] #:transparent)
```

1 function does
1 task which processes
1 kind of data

Process One Kind of Data at a Time

```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                    [xv : XVel]  [yv : YVel])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)



    ....

                  )
```

1 function does
1 task which processes
1 kind of data

Template?

Process One Kind of Data at a Time

```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                    [xv : XVel]  [yv : YVel])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (mk-WorldState
    (next-x x)
    (next-y y)
    (next-xv xv)
    (next-yv yv)))
```

1 function does
1 task which processes
1 kind of data

Process One Kind of Data at a Time

```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                    [xv : XVel]  [yv : YVel])
;; Represents a "ball" (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (mk-WorldState
    (next-x x ...)
    (next-y y ...)
    (next-xv xv ...)
    (next-yv yv ...)))
```

Overkill? … Maybe?

(This is what OO programmers have to do though)

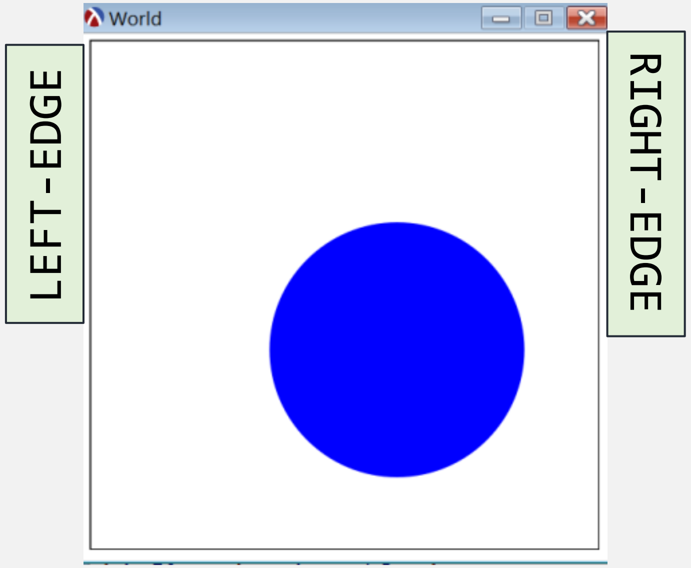
Might need extra args

Can always refactor later!

“X” Data Definition

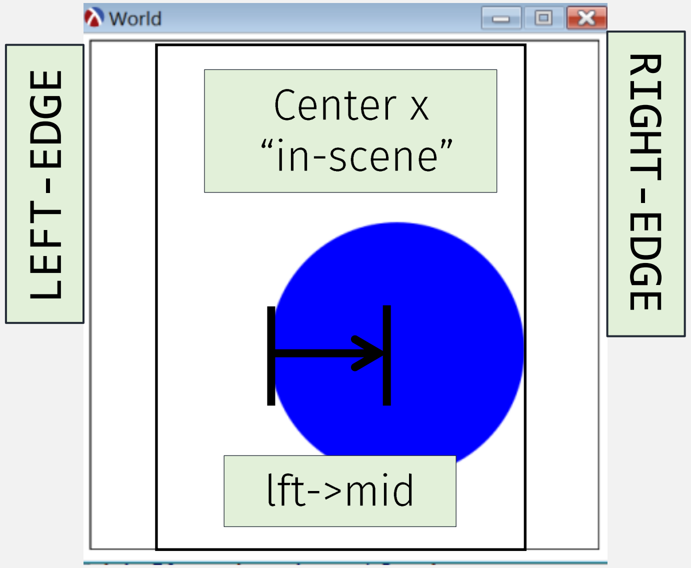
Seems like we want some **intervals**

```
;; An XCoord is one of
;; - < LEFT-EDGE
;; - > RIGHT-EDGE
;; - [LEFT-EDGE, RIGHT-EDGE]
;; Represents: possible x coordinates of ball center
```



“X” Data Definition

```
;; An XCoord is one of
;; - < LEFT-EDGE
;; - > RIGHT-EDGE
;; - [LEFT-EDGE, RIGHT-EDGE]
;; Represents: possible x coordinates of ball center
```

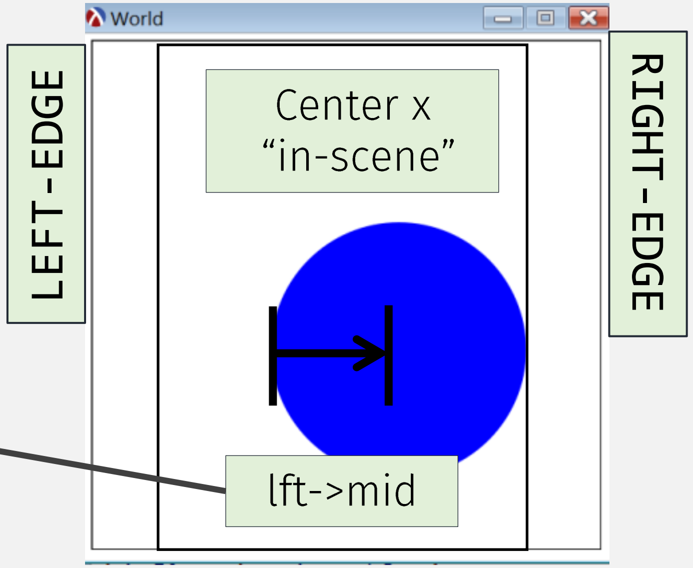


When **converting between data types**, always **define a conversion function**!

Do not inline or try to keep track in your head!

“X” Data Definition

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > RIGHT-EDGE
;; - [LEFT-EDGE, RIGHT-EDGE]
;; Represents: possible x coordinates of ball center
```

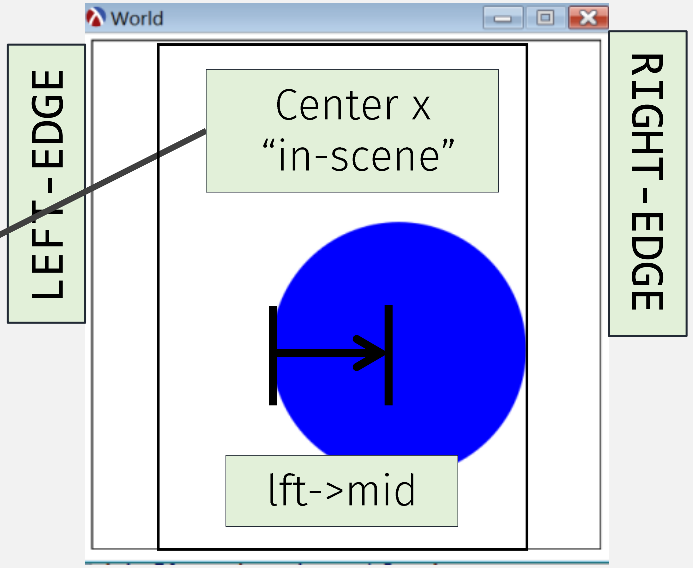


When **converting between data types**, always **define a <u>conversion function</u>**!

Do not inline or try to keep track in your head!

“X” Data Definition

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: possible x coordinates of ball center
```



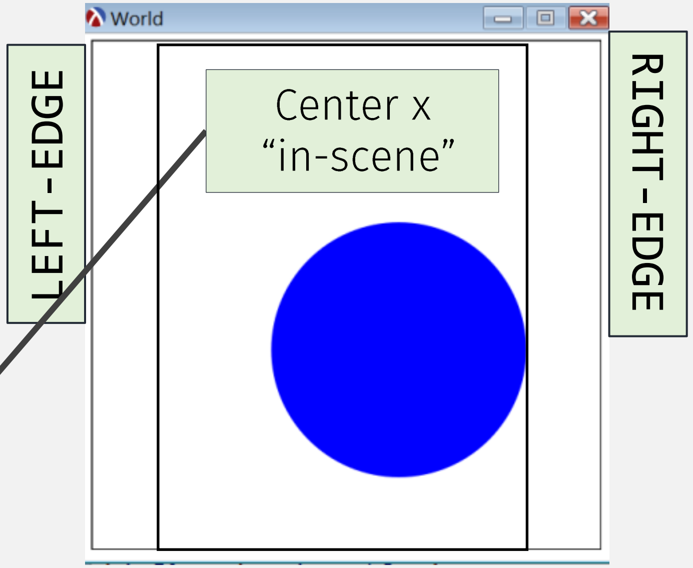
When **converting between data types**, always **define a <u>conversion function</u>**!

Do not inline or try to keep track in your head!

In-scene “X” Data Definition

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```



“Next X”

```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                    [xv : XVel]  [yv : YVel])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (mk-WorldState
    (next-x x ...)
    (next-y y ...)
    (next-xv xv ...)
    (next-yv yv ...)))
```

“Next X”

```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                   [xv : XVel]  [yv : YVel])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (mk-WorldState
    (next-x x xv)
    (next-y y ...)
    (next-xv xv ...)
    (next-yv yv ...)))
```

Need velocity to compute “next x”

“Next X”

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

WANT: X should always be “in-scene”

```
;; next-x : InSceneX Velocity -> InSceneX
;; computes new x position of ball after 1 tick
(define (next-x x xv)
  (+ x xv)))
```

May go out of scene!

Not always an “InSceneX”

“Let’s add some `if`s and **`cond`s**!”

“Next X”

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

When **converting between data types, define a conversion function**!

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

```
;; next-x : InSceneX Velocity -> InSceneX
;; computes new x position of ball after 1 tick
(define (next-x x xv)
 .... (if (in-scene? (+ x xv)) .... ))
```

May go out of scene!

Not always an “InSceneX”

“Let’s add some `ifs` and **conds**!”

(but **only if the data definition allows!**)

Convert “X” to In-scene “X”

When **converting between data types, define a conversion function!**

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

```
;; next-x : InSceneX Velocity -> InSceneX
;; computes new x position of ball after 1 tick
(define (next-x x xv)
  (x->in-scene-x (+ x xv)))
```

“X function” template

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

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

```
;; next-x : InSceneX Veloci
;; compu                 ion
(define (next-x x xv)
```

“Let’s add some **ifs** and **conds**!”

(but **only if the data definition allows!**)

TEMPLATE??

```
;; x-fn: XCoord -> ???

(define (x-fn x)
  (cond
    [(< x (lft->mid LEFT-EDGE)) ....]
    [(> x (rgt->mid RIGHT-EDGE)) ....]
    [(InSceneX? x) ....]))
```

“X function” template

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

```
;; next-x : InSceneX Veloci
;; computes new x position
(define (next-x x xv)
  (x->in-scene-x (+ x xv)))
```

TEMPLATE??

```
;; x-fn: XCoord -> ???

(define (x-fn x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) ....]
   [(InSceneX? x) ....]))
```

X -> In-Scene X

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) ....]
   [(InSceneX? x) ....]))
```

X -> In-Scene X

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinates of ball center
```

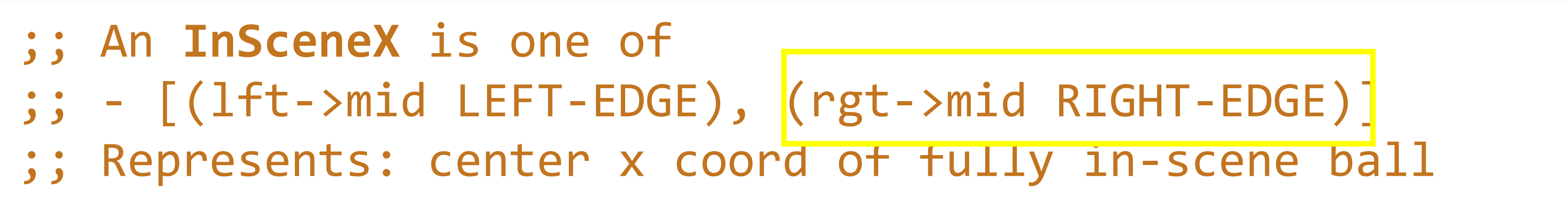
```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) ....]
   [(InSceneX? x) x]))
```

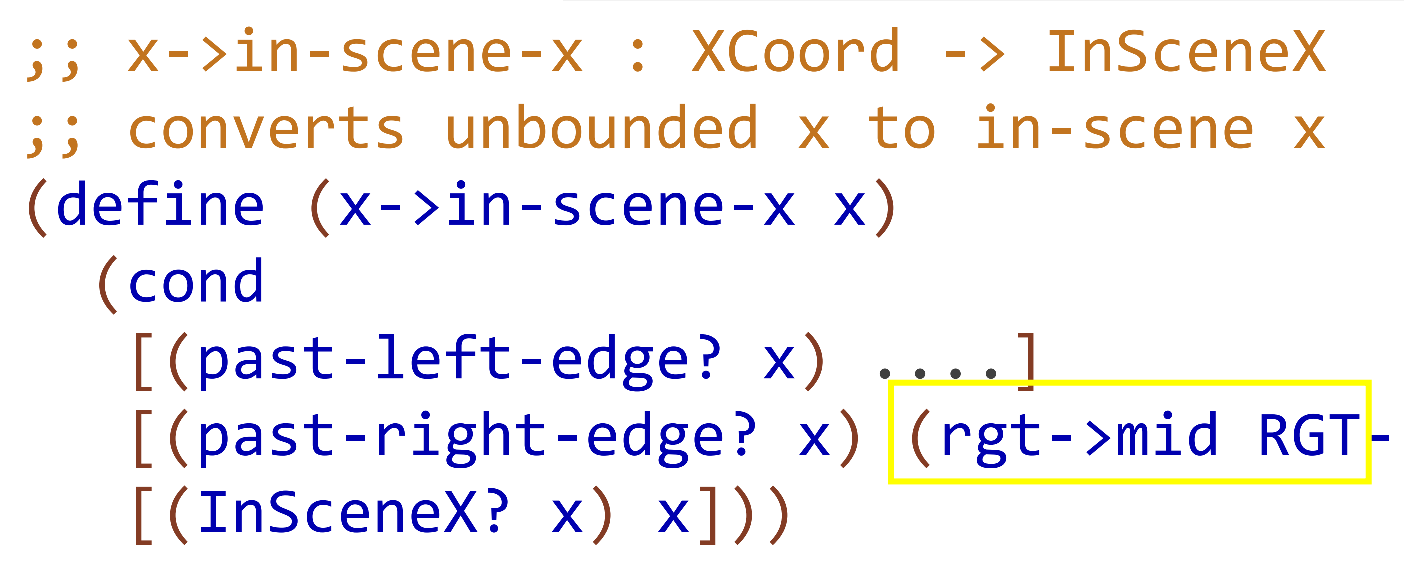
When **converting between data types,**
define/use a conversion function!

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

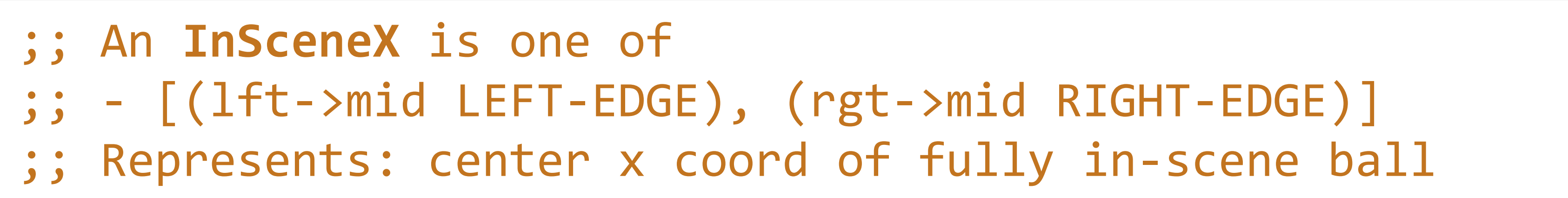
```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) RGT-EDGE .?.]
   [(InSceneX? x) x]))
```



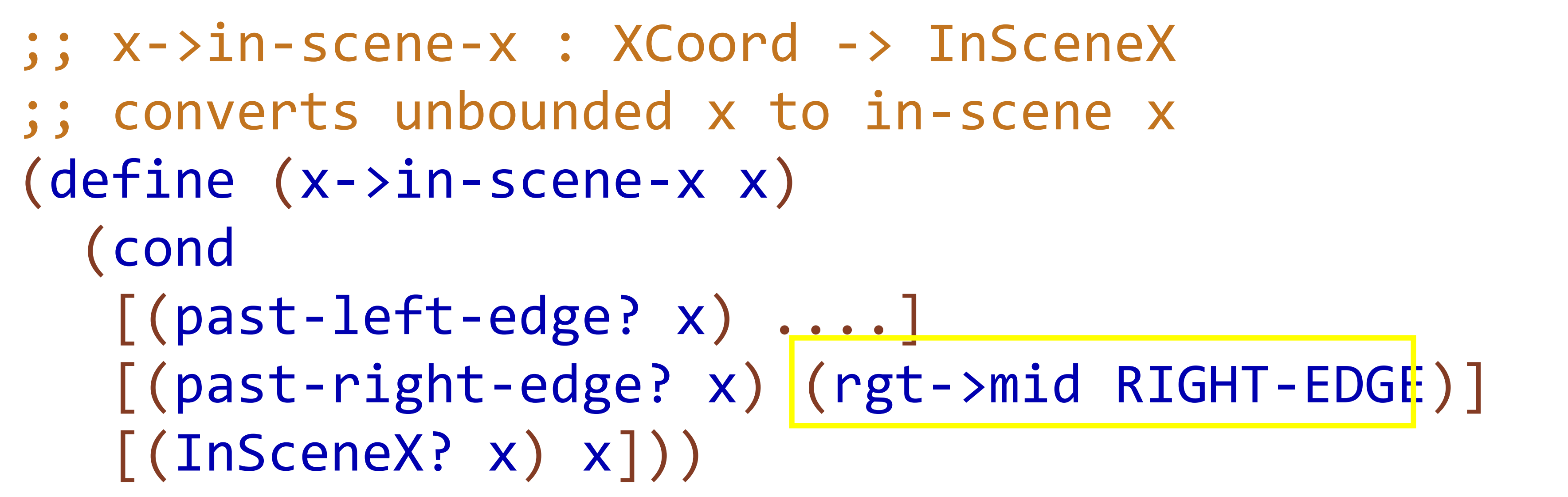
```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```



```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) (rgt->mid RGT-
   [(InSceneX? x) x]))
```



```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```



```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) ....]
   [(past-right-edge? x) (rgt->mid RIGHT-EDGE)]
   [(InSceneX? x) x]))
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

When converting between data types, define/use a conversion function!

```
;; x->in-scene-x : XCoord -> InSceneX
;; converts unbounded x to in-scene x
(define (x->in-scene-x x)
  (cond
   [(past-left-edge? x) (lft->mid LEFT-EDGE)]
   [(past-right-edge? x) (rgt->mid RIGHT-EDGE)]
   [(InSceneX? x) x]))
```

```
;; An XCoord is one of
;; - < (lft->mid LEFT-EDGE)
;; - > (rgt->mid RIGHT-EDGE)
;; - InSceneX
;; Represents: possible x coordinate of ball center
```

```
(define (XCoord? x) (real? x))
```

```
;; An InSceneX is one of
;; - [(lft->mid LEFT-EDGE), (rgt->mid RIGHT-EDGE)]
;; Represents: center x coord of fully in-scene ball
```

Use contracts to verify!

```
(define (InSceneX? x)
  (<= (lft->mid LEFT-EDGE)
      x
      (rgt->mid RIGHT-EDGE)))
```

```
(define/contract (x->in-scene-x x)
  (-> XCoord? InSceneX?)
  (cond
   [(past-left-edge? x) (lft->mid LEFT-EDGE)]
   [(past-right-edge? x) (rgt->mid RIGHT-EDGE)]
   [(InSceneX? x) x]))
```

“Next ?”

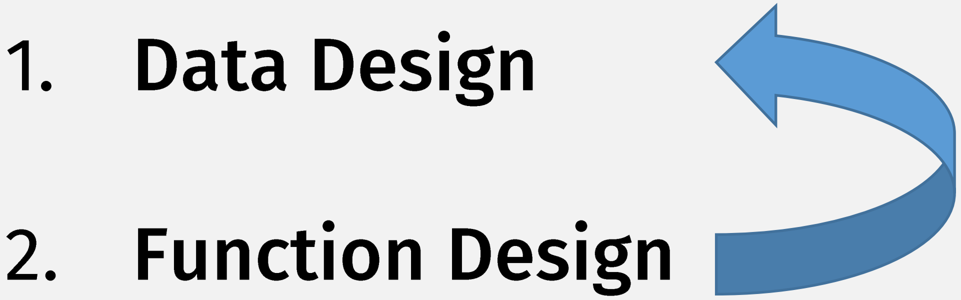
```
;; A WorldState is a (mk-WorldState [x : XCoord] [y : YCoord]
                                    [xv : XVel]  [yv : YVel])
;; Represents a “ball” (solid red circle) in big-bang animation where:
;; …
```

```
;; computes new position and vel of ball after 1 tick
(define (next-WorldState w)
  (match-define (world x y xv yv) w)
  (mk-WorldState
    (next-x x xv)
    (next-y y ...)
    (next-xv xv ...)
    (next-yv yv ...)))
```

Other functions can be defined in a similiar way

Program Design Recipe

... is iterative!



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

Function Design Recipe

... is iterative!

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**

Make it bounce?

```
;; A WorldState is a
(struct world [x y xvel yvel])
;; where:
;; x: Coordinate - represents x coordinate of ball center
;; y: Coordinate - represents y coordinate of ball
;; xvel: Velocity - in x direction
;; yvel: Velocity - in y direction
```

If you're **no longer following the template,** then the **Data Definitions need updating!**

```
;; next-world : WorldState -> WorldState
;; Computes the next ball pos
```

```
(define (next-world w)
  (match-define (world x y xvel yvel) w)
  (define new-xvel
    (if (or (>= x RIGHT-EDGE)
            (<= x LEFT-EDGE)) (- xvel) xvel)
  (define new-yvel???
    (if (or (>= y BOTTOM-EDGE)
```

DON'T PROGRAM LIKE THIS!!!

In-class exercise 2/20

on gradescope