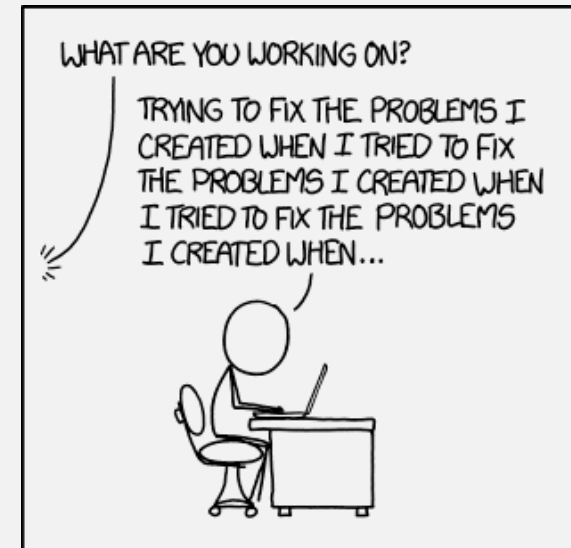


UMass Boston Computer Science
CS450 High Level Languages
Recursive Data Definitions

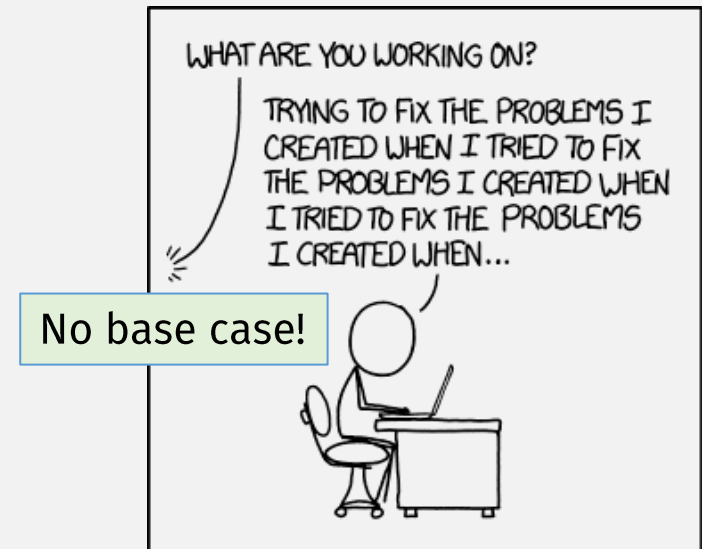
Tuesday, February 25, 2025



Logistics

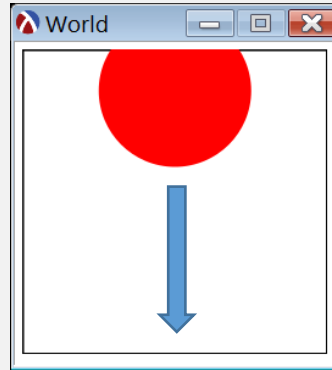
- HW 3 in
 - ~~due: Tue 2/25 11am EST~~
- HW 4 out
 - due: Tue 3/4 11am EST

(What's wrong with this recursion?)



Last
Time

Falling “Ball” Example

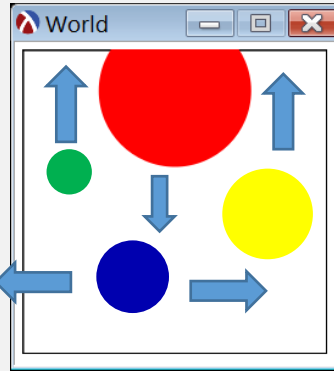


← What if: the ball can also move side-to-side? →

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

Last
Time

Falling “Ball” Example



← What if: the ball can also move side-to-side? →

What if: there are multiple balls?

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

WorldState would need:
multiple (compound) *x* and *y* coordinates

DEMO

Previously

Kinds of Data Definitions

- Basic data
 - E.g., numbers, strings, etc
- Intervals
 - Data that is from a range of values, e.g., $[0, 100)$
- Enumerations
 - Data that is one of a list of possible values, e.g., “green”, “red”, “yellow”
- Itemizations
 - Data value that can be from a list of possible other data definitions
 - E.g., either a string or number (Generalizes enumerations)

Previously

Kinds of Data Definitions

- Basic data
 - E.g., numbers, strings, etc
- Intervals
 - Data that is from a range of values, e.g., [0, 100)
- Enumerations
 - Data that is one of a list of possible values, e.g., “green”, “red”, “yellow”
- Itemizations
 - Data value that can be from a list of possible other data definitions
 - E.g., either a string or number (Generalizes enumerations)
- • Compound Data **???**
 - Data that is a combination of values from other data definitions

Last
time

Random

Ball Animation

Design a **big-bang** animation that:

- Start: a single ball, moving with random x and y velocity

Randomness

[bracketed args] = optional

`(random k [rand-gen]) → exact-nonnegative-integer?`

`k : (integer-in 1 4294967087)`

`rand-gen : pseudo-random-generator?`

`= (current-pseudo-random-generator)`

When called with an integer argument *k*, returns a random exact integer in the range 0 to *k*-1.

Optional arg Default value

`(random min max [rand-gen]) → exact-integer?`

`min : exact-integer?`

`max : (integer-in (+ 1 min) (+ 4294967087 min))`

`rand-gen : pseudo-random-generator?`

`= (current-pseudo-random-generator)`

When called with two integer arguments *min* and *max*, returns a random exact integer in the range *min* to *max*-1.

“random” is not random???

Not secure!
e.g., for generating
passwords

A pseudorandom number generator (PRNG), also known as a **deterministic random bit generator (DRBG)**,^[1] is an **algorithm** for generating a sequence of numbers whose properties approximate the properties of sequences of **random numbers**. The PRNG-generated sequence is **not truly random**, because it is completely determined by an initial value, called the PRNG's **seed**.

VS

A **cryptographically secure** pseudorandom number generator (CSPRNG) or **cryptographic pseudorandom number generator (CPRNG)** is a **pseudorandom number generator** (PRNG) with properties that make it suitable for use in **cryptology**.

Random Functions: Same Recipe (almost)!

```
;; A Velocity is a non-negative integer
;; Interp: reresents pixels/tick change in a ball coordinate
(define MAX-VELOCITY 10)
```

```
;; random-velocity : -> Velocity
;; returns a random velocity between 0 and MAX-VELOCITY
(define (random-velocity)
  (random MAX-VELOCITY))
```

Functions can
have zero args

Random functions have
no examples

```
(check-true (< (random-velocity) MAX-VELOCITY))
(check-true (>= (random-velocity) 0))
(check-true (integer? (random-velocity)))
(check-pred (λ (v) (and (integer? v)
                        (< v MAX-VELOCITY)
                        (>= v 0))))
(random-velocity))
```

Can still **test!**
Just less precise

```
;; random-x      : -> ???
;; random-y      : -> ???
;; random-ball   : -> ???
```

Multi-ball Animation

Design a **big-bang** animation that:

- Start: a single ball, moving with random x and y velocity
- On click: add a ball at a random location, with random velocity
- If any ball “hits” an edge:
 - if it's a vertical edge, the x velocity should flip direction
 - If it's a horizontal edge, the y velocity should flip direction

`;; A WorldState is ... an unknown number of balls!`

Kinds of Data Definitions

- Basic data
 - E.g., numbers, strings, etc
- Intervals
 - Data that is from a range of values, e.g., [0, 100)
- Enumerations
 - Data that is one of a list of possible values, e.g., “green”, “red”, “yellow”
- Itemizations
 - Data value that can be from a list of possible other data definitions
 - E.g., either a string or number (Generalizes enumerations)
- Compound Data - Combines values from other data definitions
 - Fixed size (e.g., struct)
 - Arbitrary size

today



Arbitrary Size Data - Lists

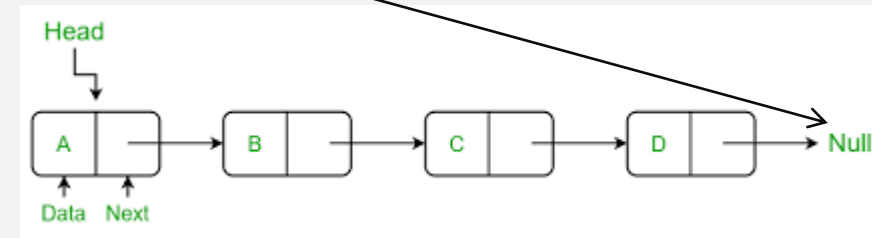
In C

```
struct node  
{ int data;  
  struct node *next; } *head;
```

Where's base case??

This is a terrible
data definition 😞

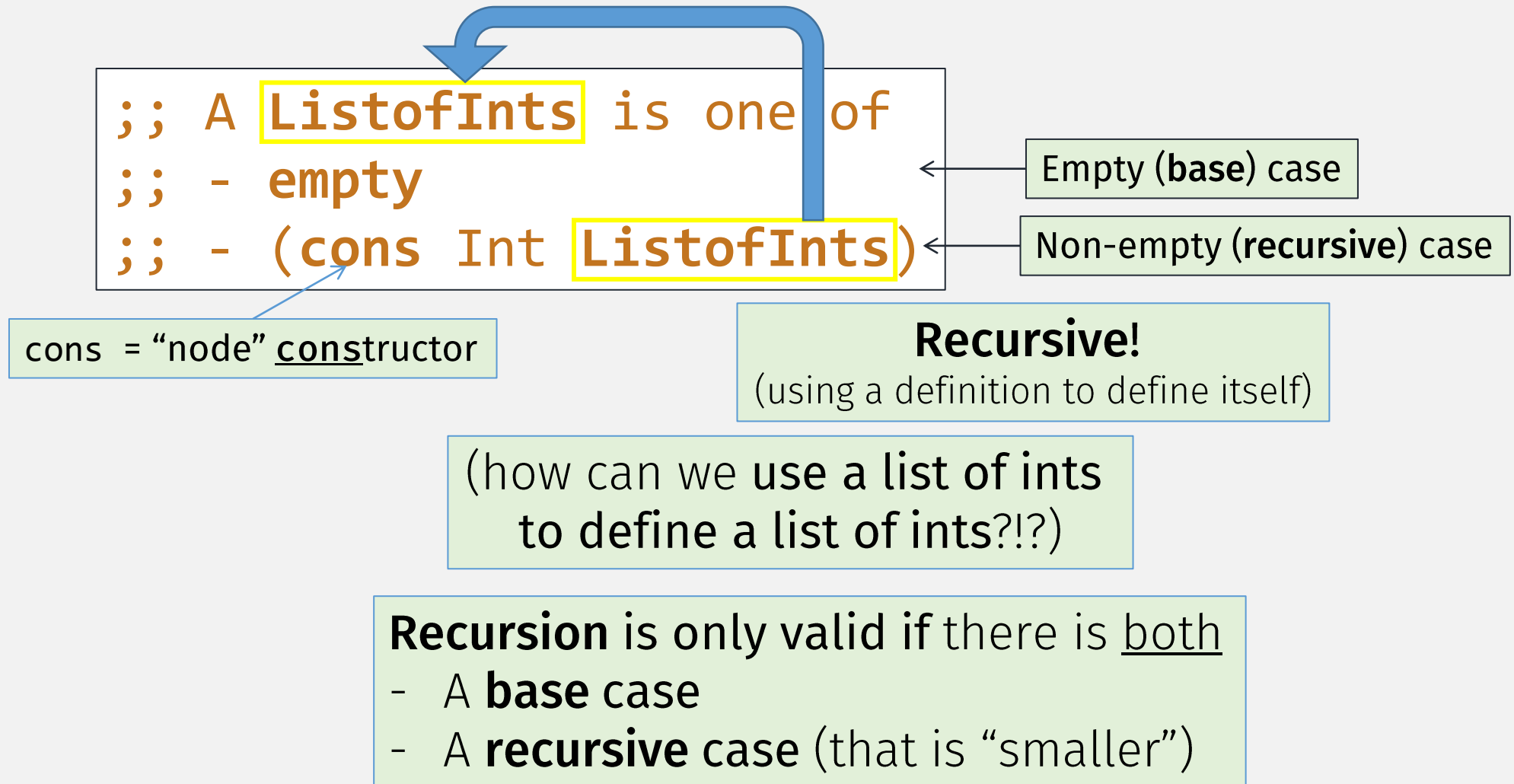
This is a **self-referential**
(i.e., **recursive!**) definition!



Recursion is only valid if there is both

- A **base case**
- A **recursive case** (that is “smaller”)

Racket List Data Definition Example



Racket List Data Definition Example

```
;; A ListofInts is one of  
;; - empty  
;; - (cons Int ListofInts)
```

TEMPLATE??

(what kind of data
definition is this?)

Racket List Data Definition Example

```
;; A ListofInts is one of  
;; - empty  
;; - (cons Int ListofInts)
```

Empty (**base**) case

Non-empty (**recursive**) case

This is an **itemization**,
so template has cond

TEMPLATE??

Empty (**base**) case

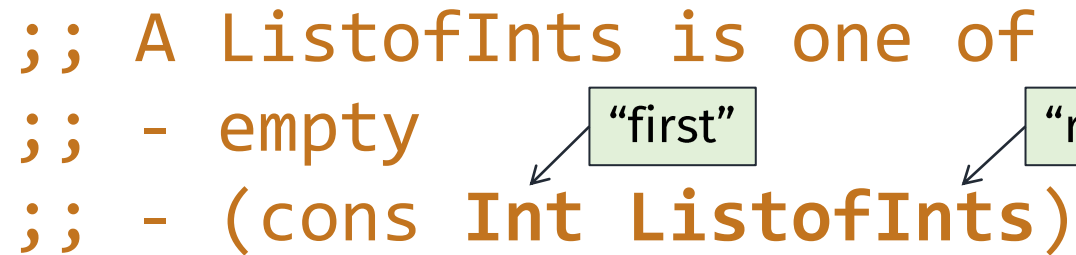
Non-empty (**recursive**) case

```
;; TEMPLATE for list-fn  
;; list-fn : ListofInts -> ???  
(define (list-fn lst)  
  (cond  
    [(empty? lst) ....]  
    [(cons? lst) .... (first lst) ....  
                      .... (rest lst) ....])))
```

The shape of the function
matches
The shape of the data definition!

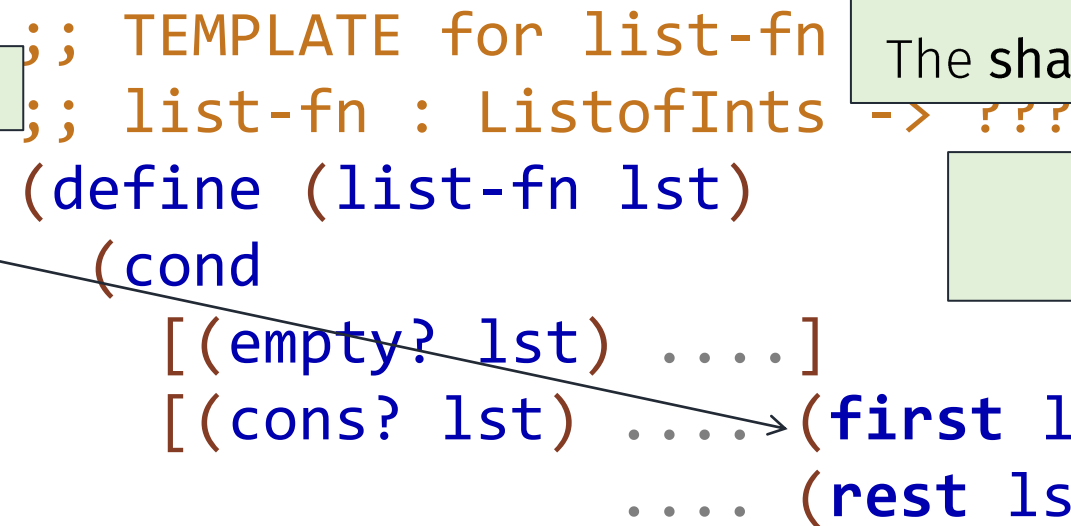
Racket List Data Definition Example

```
;; A ListofInts is one of  
;; - empty  
;; - (cons Int ListofInts)
```



This is both
itemization,
so template has `cond` and
compound data,
so template has “getters”

```
;; TEMPLATE for list-fn  
;; list-fn : ListofInts -> ???  
(define (list-fn lst)  
  (cond  
    [(empty? lst) ....]  
    [(cons? lst) .... (first lst) ....  
                      .... (rest lst) ....])))
```



The shape of the function
matches
The shape of the data definition!

Wait, where is the
recursion???

Racket List Data Definition Example

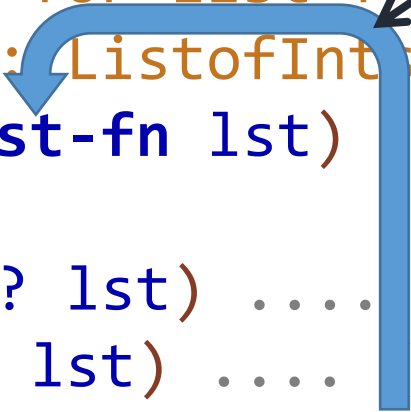
```
;; A ListofInts is one of  
;; - empty  
;; - (cons Int ListofInts)
```



TEMPLATE??

... is also recursive!

```
;; TEMPLATE for list-fn  
;; list-fn : ListofInts -> ???  
(define (list-fn lst)  
  (cond  
    [(empty? lst) ....]  
    [(cons? lst) .... first lst) ....  
      .... (list-fn (rest lst)) .... ]))
```

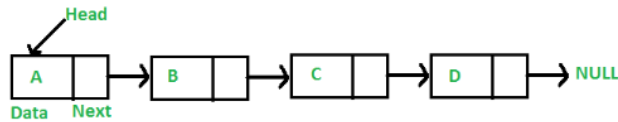


The shape of the function
matches
The shape of the data definition!

So recursion in the data definition
... means recursion in the
(template) function!

Racket Recursive List Fn Example: sum

Given a singly linked list. The task is to find the sum of nodes of the given linked list.



Task is to do $A + B + C + D$.

Examples:

[geeksforgeeks.com](https://www.geeksforgeeks.com)

Input: 7->6->8->4->1

Output: 26

Sum of nodes:

$7 + 6 + 8 + 4 + 1 = 26$

Input: 1->7->3->9->11->5

Output: 36

Examples!

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst) ....
      .... (list-fn (rest lst)) ....])))
```

Racket Recursive List Fn Example: sum

Design Recipe:
Now fill in
template!
(with arithmetic)

```
;; Returns sum of list of ints
;; sum-lst: ListofInts -> Int
(define (sum-lst lst)
  (cond
    [(empty? lst) ...]
    [else .... (first lst) ....
               .... (sum-lst (rest lst)) .... ])))
```

Racket Recursive List Fn Example: sum

```
;; A ListofInts is one of  
;; - empty  
;; - (cons Int ListofInts)
```

```
;; Returns sum of list of ints  
;; sum-lst: ListofInts -> Int  
(define (sum-lst lst)  
  (cond  
    [(empty? lst) 0]  
    [else .... (first lst) ...  
               .... (sum-lst (rest lst)) ....]))
```

NEXT: How to
“combine” pieces?

Think about
data types!

Racket Recursive List Fn Example: sum

```
;; Returns sum of list of ints
;; sum-lst: ListofInts -> Int
(define (sum-lst lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
              (sum-lst (rest lst)))]))
```

List contracts

shallow check,
but also (delayed)
“deeper check”
(only when needed)

```
:: Returns sum of list of ints  
(define/contract (sum-lst lst)  
  (-> (listof integer?) integer?)  
  (cond  
    [(empty? lst) 0]  
    [else (+ (first lst)  
              (sum-lst (rest lst)))])))
```

Multi-ball Animation

Design a **big-bang** animation that:

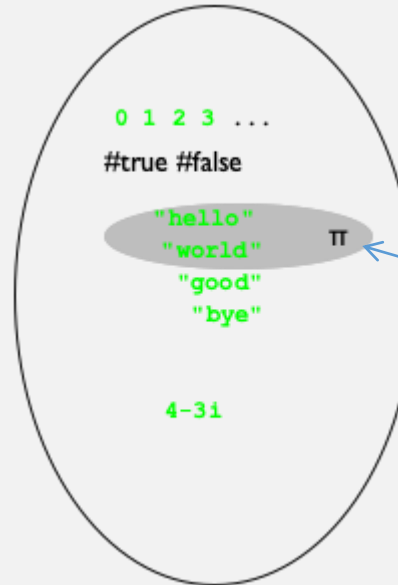
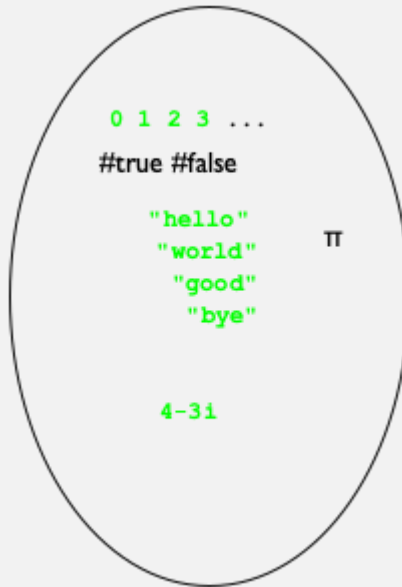
- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball “hits” an edge:
 - if it's a vertical edge, the x velocity should flip direction
 - If it's a horizontal edge, the y velocity should flip direction

∴ A `WorldState` is ... an unknown number of balls!

;; A `WorldState` is ... a list of balls!

Interlude: Data Definitions (ch 5.7)

All possible data values



A data definition
= (a named) subset of all possible values

We are defining which data values are valid for our program!

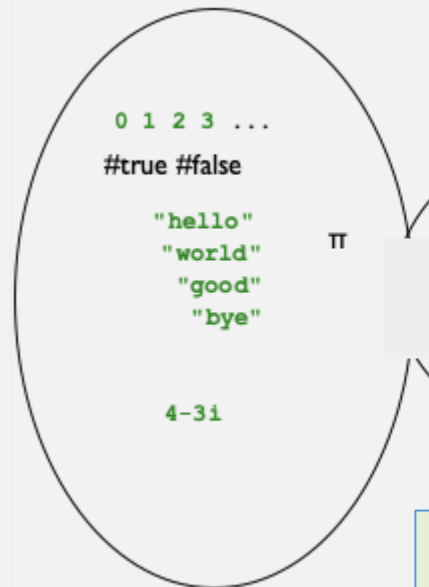
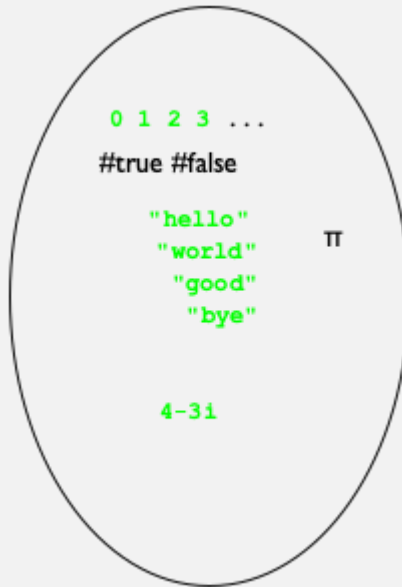
All programs are data manipulators ...

So this must be the first step of programming!

Also makes “error handling” easy

Interlude: Data Definitions (ch 5.7)

All possible basic data values



Possible to expand the universe of values, e.g., new **compound data definitions** (struct, or other data structure)

A Venn diagram showing the expansion of the universe of values. It consists of two overlapping ovals. The left oval contains the same basic data values as the first diagram. The right oval contains compound data definitions. The intersection of the two ovals is shaded gray and contains the following text:

- (make-posn 0 3)
- (make-posn 1 3)
- (make-posn 2 3)
- (make-posn 3 3)

The right oval also contains the following text:

- (make-posn "hello" 0)
- (make-posn "world" 1)
- (make-posn "good" 2)
- (make-posn "bye" 3)
- (make-posn (make-posn 0 1) 2)

A Venn diagram showing the expansion of the universe of values. It consists of two overlapping ovals. The left oval contains the same basic data values as the first diagram. The right oval contains compound data definitions. The intersection of the two ovals is shaded gray and contains the following text:

- (make-posn 0 3)
- (make-posn 1 3)
- (make-posn 2 3)
- (make-posn 3 3)

The right oval also contains the following text:

- (make-ball -1 0)
- (make-ball -1 1)
- (make-ball -1 2)
- (make-ball -1 3)
- (make-ball "bye" #t)

A Venn diagram showing the expansion of the universe of values. It consists of two overlapping ovals. The left oval contains the same basic data values as the first diagram. The right oval contains compound data definitions. The intersection of the two ovals is shaded gray and contains the following text:

- (make-posn 0 3)
- (make-posn 1 3)
- (make-posn 2 3)
- (make-posn 3 3)

The right oval also contains the following text:

- (list 1)
- (list 1 2)
- (list 1 2 3)
- ...

Multi-ball Animation

Design a **big-bang** animation that:

- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
- If any ball “hits” an edge:
 - if it's a vertical edge, the x velocity should flip direction
 - If it's a horizontal edge, the y velocity should flip direction

∴ A `WorldState` is ... an unknown number of balls!

;; A `WorldState` is ... a list of balls!

Ball

```
;; A WorldState is a  
(struct world [x y xvel yvel] #:transparent)  
;; when ball  
;; x: XCoord - represents x coordinate of ball center in animation  
;; y: YCoord - represents y coordinate of ball center in animation  
;; xvel: Integer - represents x velocity, where  
;;                positive = to the right, negative = to the left  
;; yvel: Integer - represents y vel, where  
;;                positive = down, negative = up
```

```
;; A ListofBall is one of:  
;; - empty  
;; - (cons Ball ListofBall
```

```
;; A WorldState is a ListofBall
```

```
(define (main)
  (big-bang (list (random-ball))
    [on-mouse handle-mouse]
    [on-tick next-World]
    [to-draw World->Image]))
```

```
;; A WorldState is a ListofBall
```

These need to be
updated to handle new
WorldState data def

“next World”

List template!

```
;; list-fn: WorldState -> ???  
;; list of Ball template  
(define (list-fn w)  
  (cond  
    [(empty? w) ....]  
    [else .... (first w) ....  
               .... (list-fn (rest w)) .... ])))
```

“next World”

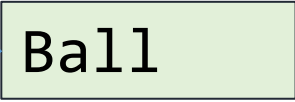
List template!

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) ....]
    [else .... (first w) ....
               .... (next-World (rest w)) ....])))
```

“next World” example

1 function does
1 task which processes
1 kind of data

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) ....]
    [else .... (first w) ....
               .... (next-World (rest w)) ....]]))
```



```
(check-equal?
  (next-World (list (mk-Ball 0 0 1 1)))
  (list (next-Ball (mk-Ball 0 0 1 1))))
```


Last
Time

“next Ball”

This was the previous “next-World” function!

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

“next World”

Think about
data types!

Fill in template ...

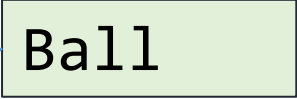
;; A WorldState is a ListofBall

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) ....]
    [else .... (first w) ....
               .... (next-World (rest w)) .... ])))
```

“next World”

Think about
data types!

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) empty]
    [else .... (first w) ....
               .... (next-World (rest w)) ....]]))
```



1 function does
1 task which processes
1 kind of data

“next World”

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) empty]
    [else .... (next-Ball (first w)) ....
               .... (next-World (rest w)) .... ])))
```

NEXT: How to
“combine” pieces?

Think about
data types!

Want:
Ball + ListofBall
-> ListofBall

1 function does
1 task which processes
1 kind of data

“next World”

```
;; next-World : WorldState -> WorldState
;; Computes the next world state on a tick
(define (next-World w)
  (cond
    [(empty? w) empty]
    [else (cons (next-Ball (first w))
                  (next-World (rest w)))]))
```

NEXT: How to
“combine” pieces?

Think about
data types!

“Render World”

List template!


```
;; list-fn : list-fn -> ???  
;; TEMPLATE for list functions  
(define (list-fn lst)  
  (cond  
    [(empty? lst) .... ]  
    [else  
     .... (first lst) ....  
     .... (list-fn (rest lst)) .... ]))
```

“Render World”

```
;; World->Image : World -> Image
;; Draws Ball images into a scene
(define (World->Image w)
  (cond
    [(empty? w) ... ]
    [else
     .... (first w) ....
     .... (World->Image (rest w)) .... ]))
```

“Render World”

```
;; World->Image : World -> Image
;; Draws Ball images into a scene
(define (World->Image w)
  (cond
    [(empty? w) EMPTY-SCENE]
    [else
     .... (first w) ....
     .... (World->Image (rest w)) .... ]))
```



“Combine” the pieces

“Render World”

```
;; World->Image : World -> Image
;; Draws Ball images into a scene
(define (World->Image w)
  (cond
    [(empty? w) EMPTY-SCENE]
    [else
     (Ball->Image (first w) ....
                  .... (World->Image (rest w)) .... ]))
```

Want:

Ball -> Image and:
Image + Image -> Image

??

“Combine” the pieces

“Render World”

```
;; World->Image : World -> Image
;; Draws Ball images into a scene
(define (World->Image w)
  (cond
    [(empty? w) EMPTY-SCENE]
    [else (add-Ball-to-scene
            (first w)
            (World->Image (rest w)))]))
```

Or:
Ball + Image -> Image

??

“Combine” the pieces

“handle Mouse”

For multi-arg function, choose which (argument's) template to use

Enumeration



```
;; handle-mouse : WorldState Xcoord Ycoord MouseEvent
;; Inserts new random ball onclick
(define (handle-mouse w x y mevt)
  (cond
    [(click? mevt) (add-random-ball w)]
    [else w]))
```

Enumeration

```
;; handle-mouse : WorldState Xcoord Ycoord MouseEvent  
;; Inserts new random ball onclick  
(define (handle-mouse w x y mevt)  
  (cond  
    [(click? mevt) (add-random-ball w)]  
    [else w]))
```

-> WorldState

Enumeration template
(collapsed)

1 function does
1 task which processes
1 kind of data

Multi-ball Animation: more?

Design a **big-bang** animation that:

- Start: a single ball, moving with random x and y velocity
- On a click: add a ball at random location, with random velocity
 - And random size?
 - And random color?
- If any ball “hits” an edge:
 - if it's a vertical edge, the x velocity should flip direction
 - If it's a horizontal edge, the y velocity should flip direction

;; A WorldState is ... a list of balls!

In-class exercise 2/25
on gradescope