UMass Boston Computer Science

CS450 High Level Languages

Compound Data Definitions

Tuesday, February 17, 2026

```
class Circle {
  Num radius;
  Color col;
}
```

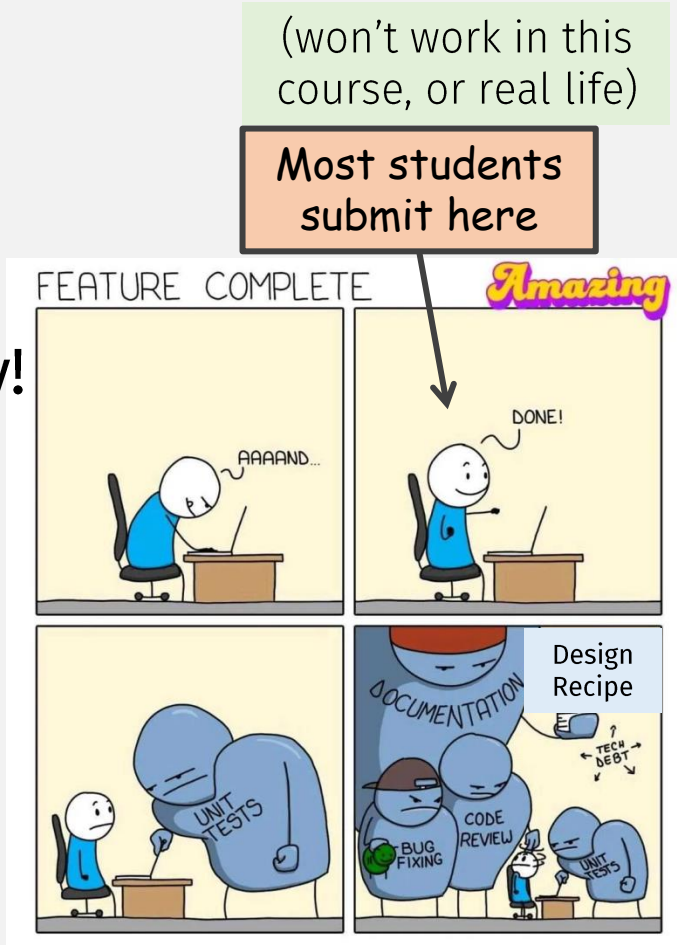
Logistics

- HW 1 grades out
  - Re-grade requests must address specific deduction

- HW 2 in
  - ~~due: Tues 2/17, 11am EST~~
  - Files should not start `big-bang` loop automatically! (will get GradeScope timeout)

- HW 3 out
  - due: Tues 2/24 11am EST
  - Similar to HW 2, but with compound data definitions (start from scratch!)

(won't work in this course, or real life)



HW Advice

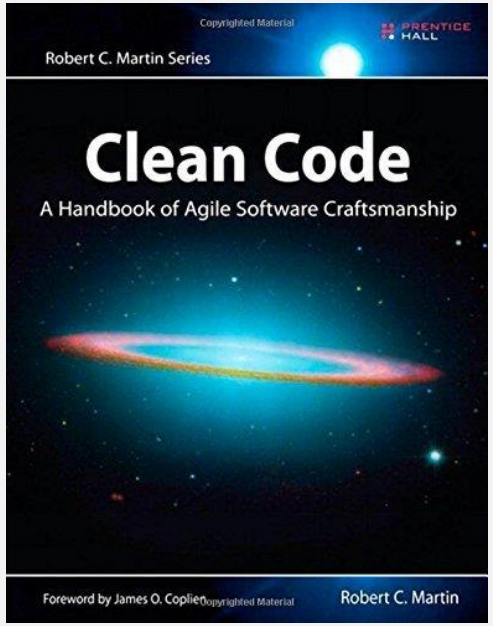
"Perhaps you thought that "**getting it working**" was the first order of business for a professional developer.

THIS COURSE

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

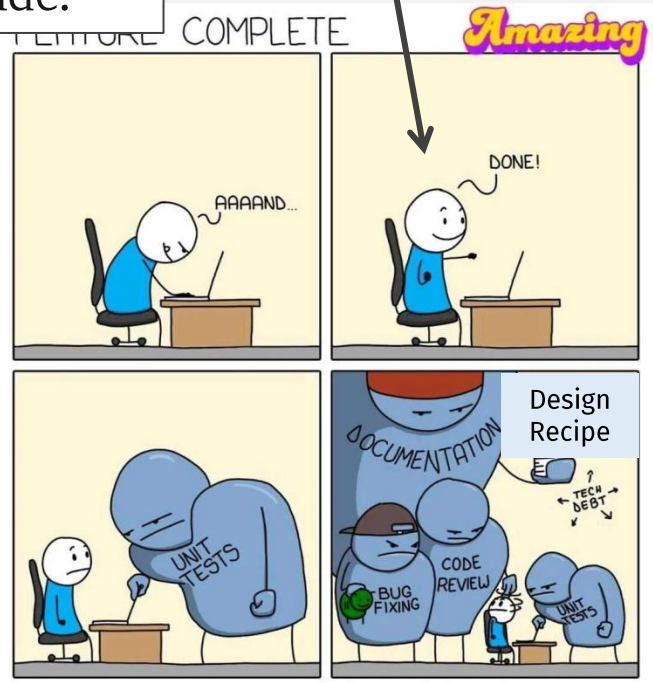
The functionality that you create today has a good chance of changing in the next release, but the **readability of your code** will have a profound effect on all the changes that will ever be made."

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



(won't work in this course, or real life)

Most students submit here



HW Observations

- Reminder: **Not ok to submit code** that … (larger deductions coming soon)
  - doesn't (or hasn't been) **run**
  - has **no tests**
  - has **failing / erroring tests**
  - doesn't **match Github** (???)
  - has **large blocks of commented code**
  - has **undescriptive commit msgs**
  - is uploaded with **GitHub "file upload" feature**

- See: **Incremental Programming Pledge!**

Last Time

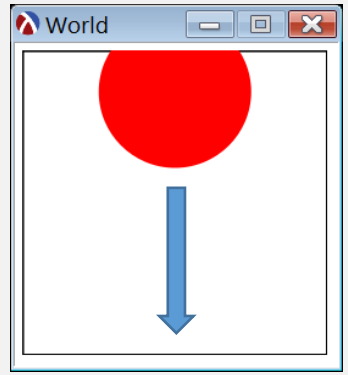
Kinds of Data Definitions

- **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)

Last Time

Falling "Ball" Example

```
;; A WorldState is a Non-negative Integer
;;          Represents:  the y Coordinate of the center of a
;;          ball in a `big-bang` animation.
```



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

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

```
;; A WorldState is an Integer ...
;; ... and another Integer???
```

We need a way to create **compound data**
i.e., a **data definition** that
combines values of other data defs

Last Time

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**

today

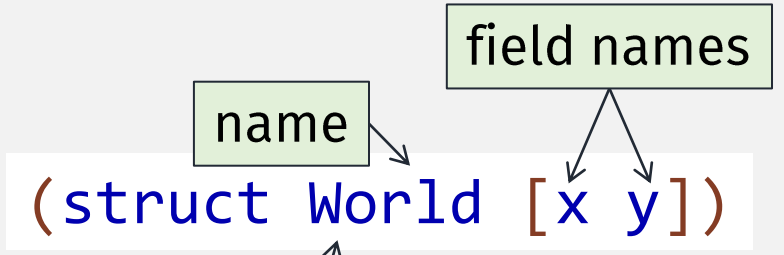
Falling “Ball” Example

???

```
;; 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
(struct World [x y])
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y)
;; ...
```

a `struct` defines a new kind of **compound data**

Parts of a `struct` definition



(Implicitly) defines:

- A **constructor** function → `World`
  - Creates instances of the struct
- **Accessor** functions → `World-x`, `World-y`
  - Get an instance’s field value
- A **predicate** → `World?`
  - Returns true for struct instances

“name” + “-” + …

… field names

“name” + “?”

Falling “Ball” Example

```
;; 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
(struct World [x y])
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y)
;; ...
```

a `struct` defines a new kind of **compound data**

Checked constructor (programmer must define)

Unchecked (internal) constructor (implicitly defined by `struct`)

```
(define INIT-WORLDSTATE (mk-WorldState 0 0))
```

Instances of the `struct` are values of that kind of data

Data Design Recipe

Data Definition

- 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**

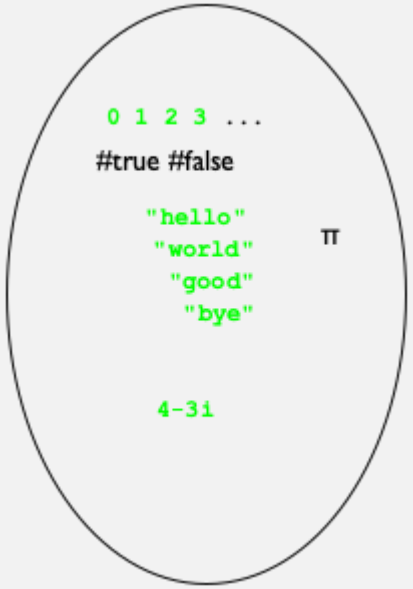
Data Design Recipe - Compound Data Update

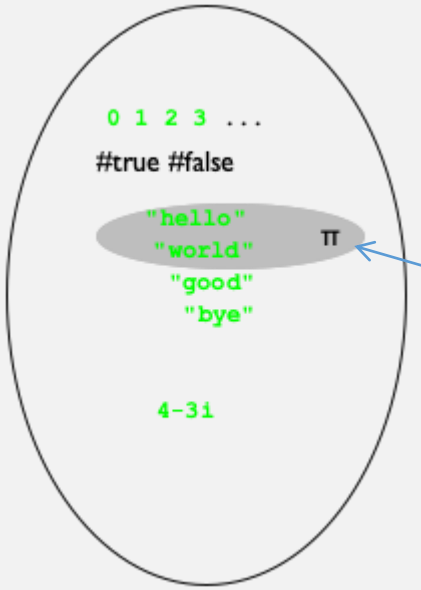
Data Definition

- Has ~~4~~ **maybe 5** 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**
  5. (checked) **Constructor** for compound data def values

Interlude: Data Definitions (HtDP Ch 5.7)

All possible data values





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

We are **defining** (and naming) the **valid data values our program!**

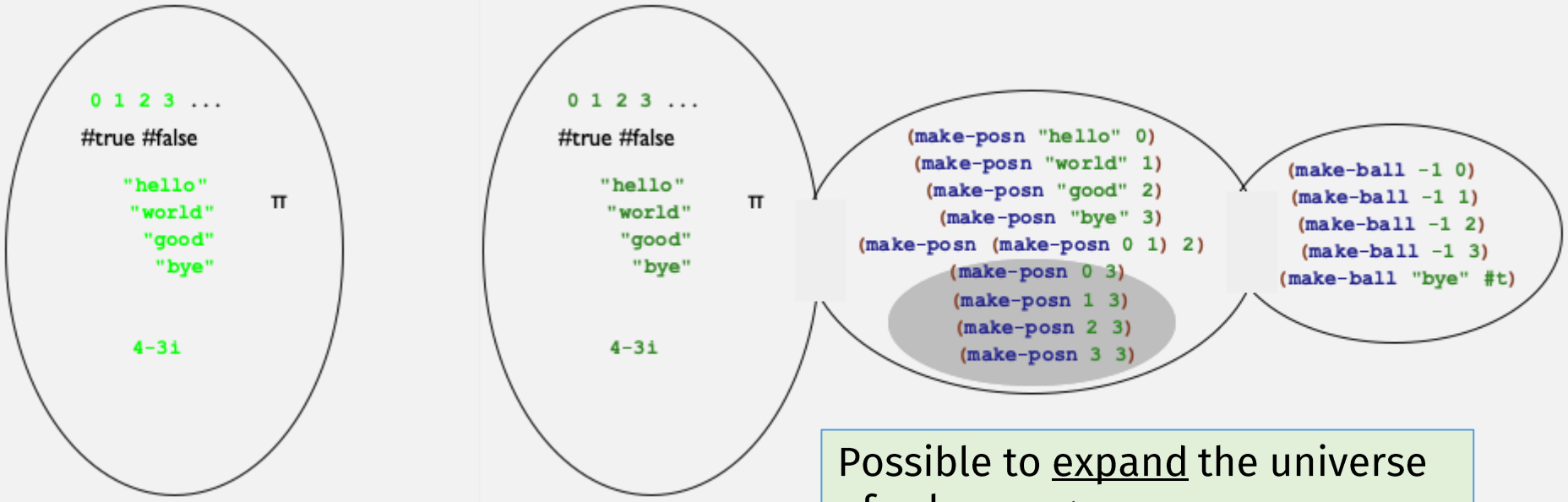
All programs manipulate some set of data values …

So this must be the **first step of programming**!

(Also, can't do **"error handling"** without knowing **valid / invalid data values**)

Interlude: Data Definitions (HtDP Ch 5.7)

All possible <u>basic</u> data values



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

Predicates for Compound Data

```
;; 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
(struct World [x y])
```

predicate?

```
(define (WorldState? arg)
  (and (World? arg)
       (integer? (World-x arg))
       (integer? (World-y arg))))
```

???

`struct` already defines `World?` … what about checking types of fields?

This "deep" predicate checks too much …

Compound data predicates should be "**shallow**" checks, i.e., `World?`

… because it's the job of "field data type" processing functions to check those kinds of data

also, maybe exponential overhead …

Checked constructor ensures that only valid instances may be created!

```
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y))
```

Data Design Recipe - Predicate Update

Data Definition

- Has **maybe 5** parts:
  1. **Name**
  2. Description of **all possible values** of the data
  3. **Interpretation** explaining the real world concepts the data represents
  4. **Predicate**
     - Evaluates to **`true`** for **some values in the Data Definition**
       - False positives **ok** Might let in some invalid values
     - Evaluates to **`false`** for **some values <u>not</u> in the Data definition**
       - False negatives **not ok** Must only reject invalid values
  5. (checked) **Constructor** for **compound data def values**

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 `check-equal?`) **the function behavior**
5. **Code** – implement the **rest of the function** (arithmetic)
6. **Tests** – check (using `check-equal?` and other test forms) the **function behavior**

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 `check-equal?`) **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 `check-equal?` and other test forms) the **function behavior**

Functions For Compound Data

- A **function that processes compound data** must …
  - **extract the individual pieces,** using accessors
  - **combine them,** with arithmetic

Functions For Compound Data - Template

- A **function that processes compound data** must
  - **extract the individual pieces,** using accessors ← Done with template
  - **combine them,** with arithmetic

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

```
;; 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
(struct World [x y])
```

```
;; TEMPLATE for WorldState-fn: WorldState -> ???
(define          (WorldState-fn w)

  .... (World-x w) ....
  .... (World-y w) .... )
```

Functions For Compound Data - Template

- A **function that processes compound data** must
  - **extract the individual pieces,** using accessors ← Done with template
  - **combine them,** with arithmetic

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

```
;; 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
(struct world [x y])
```

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

Signatures / Contracts Redundant?

Redundant?

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

Function Design Recipe - Signature / Contract Update

Submitted code no longer needs both Signature and Contract

- The **Contract is the Signature!**
- This assumes:
  - **Contract predicates represent valid Data Definitions!**
- **NOTE –** this **does not change the Design Recipe!**
  - … only submission requirements

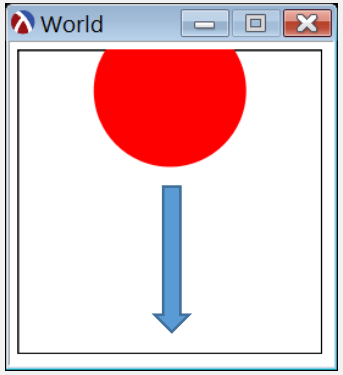
```
;; TEMPLATE for WorldState-fn: WorldState -> ???
(define/contract (WorldState-fn w)
  (-> WorldState? ??? )
  .... (world-x w) ....
  .... (world-y w) .... )
```

Function Design Recipe

Still must program with these steps, in this order!

1. **Name**
2. **Signature** – types of the **function input(s)** and **output** (not submitted in comments, if there are valid **contracts**)
3. **Description** – explain (in **English prose**) the **function behavior**
4. **Examples** – show (using `check-equal?`) **the function behavior**
5. **Template** – sketch out the **function structure** (using input's **Data Definition**) (not submitted)
6. **Code** – implement the **rest of the function** (arithmetic)
7. **Tests** – check (using `check-equal?` and other test forms) the **function behavior**

Falling “Ball” Example



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

`WorldState` would need <u>two</u> pieces of data:
the **x** *and* **y** coordinates

```
;; 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
```

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

(assuming constant velocity)

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

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

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

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

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

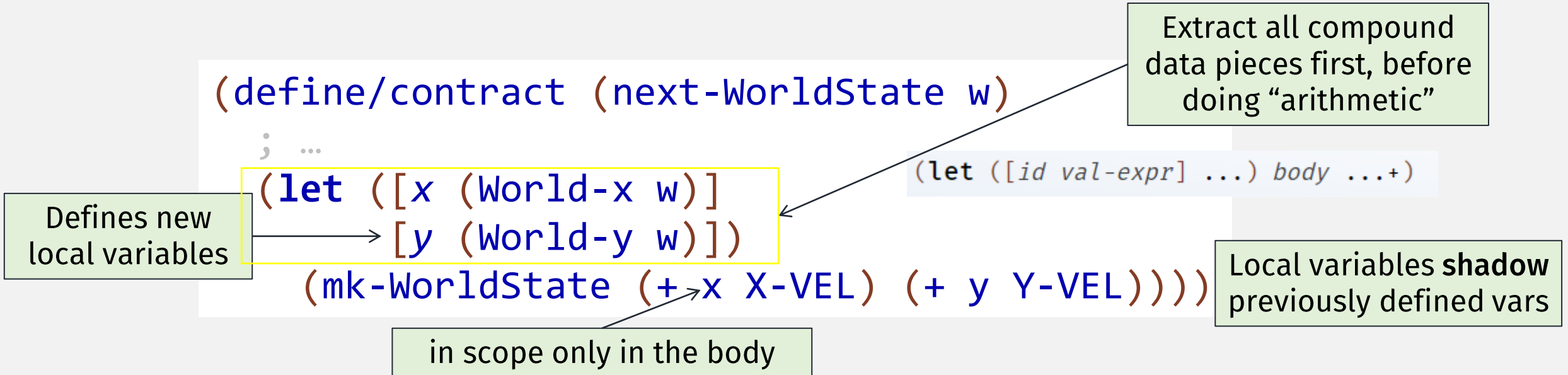
```
(check-equal?
  (next-WorldState
    (mk-WorldState 0 0))
  (mk-WorldState X-VEL Y-VEL))
```

```
;; next-WorldState
;; 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)))
```

Extract Compound Pieces – `let`

alternatives



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

alternatives

```
(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?)

Extract Compound Pieces – Pattern Match!

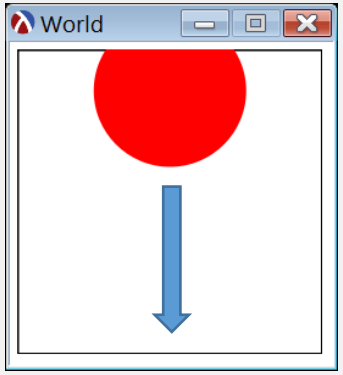
alternatives

```
(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!

Falling “Ball” Example



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

… on a key-press?

`WorldState` would need <u>two</u> pieces of data:
the **x** *and* **y** coordinates

Last Time

Some Pre-defined Enumerations

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

"Key event fn"

(result must be **WorldState**)

WorldState

But remember:

1 function does
1 task which processes
1 kind of data

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

Template

Give to: **big-bang on-key** clause

Must call separate: "WorldState fn"

Do not put **all code in one function!** e.g.,
Do not process **"WorldState" data** in a **"KeyEvent" function!**

Or even better: key=?

```
; – "\t" (tab),
; – "\r" (return), and
; – "\b" (backspace).
; Represents: keys on the keyboard
```

Compound Data can be nested

But remember:

1 function does
1 task which processes
1 kind of data

Need a different function (that uses `GameState` template) **to process** `GameState` **data**

Uses `KeyEvent` **template**

```
(define/contract (key-handler g k)
  (-> GameState? key-event? GameState?)
  (cond
    [(key=? k GET-RED)   (handle-red-key g)    ]
    [(key=? k GET-BLUE)  (handle-blue-key g)   ]
    [else w])
```

A “GameState” data def + function …

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
                  [active : PlayerID])
;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
```

```
(define/contract (GameState-fn g)
  (-> GameState? .... )

      ....    (GameState-p1 g)    ....
      ....    (GameState-p2 g)    ....
      ....      (GameState-active g)  ....   )
```

TEMPLATE

(extracts pieces of compound data)

A “GameState” function …

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
                  [active : PlayerID])

;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
```

But remember:

1 function does
1 task which processes
1 kind of data

```
(define/contract (handle-red-key g)
  (-> GameState? GameState?)
  (mk-GameState
      ....    (GameState-p1 g)     ....
      ....    (GameState-p2 g)     ....
      ....      (GameState-active g)  ....   )
```

A “GameState” function …

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
                  [active : PlayerID])

;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
```

But remember:

1 function does
1 task which processes
1 kind of data

```
(define/contract (handle-red-key g)
  (-> GameState? GameState?)
  (mk-GameState
   (Player-fn (GameState-p1 g))
   (Player-fn (GameState-p2 g))
   (PlayerID-fn (GameState-active g))))
```

A “GameState” function …

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
;;                [active : PlayerID])
;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
```

NOTE: don’t “prematurely optimize!”

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs… We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
>
> — Donald Knuth

(can always refactor to be “cleaner” later)

Pass as many compound data pieces as needed …

(trust the recipe … follow the data design … resist temptation to “prematurely optimize”)

```
(define/contract (handle-red-key g)
  (-> GameState? GameState?)
  (mk-GameState
   (Player-fn (GameState-p1 g) (GameState-p2 g) (GameState-active g))
   (Player-fn (GameState-p2 g) (GameState-p1 g) (GameState-active g))
   (PlayerID-fn (GameState-active g) (GameState-p1 g) (GameState-p2 g))))
```

Data Definition Invariants

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
                  [active : PlayerID])
;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
(struct GameState [p1 p2 active])
```

```
;; Invariant1: p1 “red” + p2 “red” <= MAX-TOKENS
```

```
;; Invariant1: p1 “blue” + p2 “blue” <= MAX-TOKENS
```

“invariant” = “must always be true!”

Every function that creates a **GameState** is responsible for maintaining these invariants!

Can this be automatically checked?

```
(define/contract (mk-GameState p1 p2 id)
  (-> Player? Player? PlayerID? GameState?)
  (GameState p1 p2 id))
```

Data Definition Invariants

(hypothetically …)

```
;; A GameState is a
;;  (mk-GameState [p1 : Player] [p2 : Player]
;;                [active : PlayerID])
;; where:
;; - p1 : represents “Player 1” data …
;; - p2 : represents “Player 2” data …
;; - active : it’s this player’s turn
(struct GameState [p1 p2 active])
```

```
;; Invariant1: p1 “red” + p2 “red” <= MAX-TOKENS
```

```
;; Invariant1: p1 “blue” + p2 “blue” <= MAX-TOKENS
```

“invariant” = “must always be true!”

Every function that creates a **GameState** is responsible for maintaining these invariants!

Can this be automatically checked?

We can define a separate “output” predicate

```
(define (GameState/invariant? x)
  (and (GameState? x)
       (<= (+ (red-count (GameState-p1 x))
              (red-count (GameState-p2 x)))
           MAX-TOKENS)
       (<= (+ (blue-count (GameState-p1 x))
              (blue-count (GameState-p2 x)))
           MAX-TOKENS)))
```

```
(define/contract (mk-GameState p1 p2 id)
  (-> Player? Player? PlayerID? GameState/invariant?)
  (GameState p1 p2 id))
```

Data Design Recipe - Compound Data Update

Data Definition

- Has ~~4 5~~ **maybe 6** 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**
  5. (checked) **Constructor** for compound data def values
  6. consider a predicate that includes invariants
     - Must be careful of excessive checking, excessive overhead

In-class exercise 2/17

on gradescope

Write templates!