UMass Boston Computer Science

CS450 High Level Languages

Programming with Compound Data

Thursday, February 19, 2026

```
class Horse
  implements Animal {
    Int age;
    Float weight;
}
```

Logistics

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

- New Office Hour Time
  - Thurs 2-3:30pm

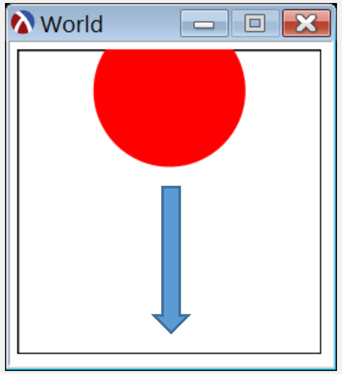
(OO languages love compound data)

```
class Horse
  implements Animal {
    Int age;
    Float weight;
}
```

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 <u>two</u> **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
<u>combines</u> <u>values</u> 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**

Last Time

Falling "Ball" Example

a `struct` defines a new kind of **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])
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y)
;; ...
```

Last Time

Parts of a `struct` definition

field names

name

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

Same as “**name**”

(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**” + “**?**”

Last Time

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))
```

Produces **instances** of the **struct** that are **values** of the new data definition

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 (most?) **values** <u>not</u> in the **Data Definition**
     - If needed, **define extra predicates** for **each enumeration** or **itemization**

Data Design Recipe - Compound Data Update

Data Definition (for compound data)

- ~~Has 4~~ Has **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 (most?) **values** not in the **Data Definition**
   - If needed, **define extra predicates** for **each enumeration** or **itemization**
5. (checked) **Constructor** for compound data def values

Data Design Recipe - Compound Data Predicate

Data Definition (for compound data)

- Has **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 (most?) **values** <u>not</u> in the **Data Definition**
     - For compound data …
  5. (checked) **Constructor** for **compound data def values**

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?` … but does not enforce field types?

This "deep" predicate checks too much …

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

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

Also not practical? **maybe exponential overhead …**

Instead, use **checked constructor: ensures** that **only valid instances** are **created!**

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

Data Design Recipe - Predicate Update

Data Definition (for compound data)

- Has **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** (shallow, conservative approximation of the Data Def)
     - **Evaluates to `true`** for **all values in the Data Def** … and maybe **some <u>not</u>**
       - False positives maybe **ok** Might let in some invalid values
     - **Evaluates to `false`** for (most?) **values <u>not</u> in the Data Def** … but maybe **not all**
       - False negatives <u>not</u> **ok** Must only reject invalid values
  5. (checked) **Constructor** for **compound data def values**

Function Design Recipe

Previously

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
  - **<u>extract</u> the individual pieces,** using accessors ← Done with template
  - **<u>combine</u> them,** with arithmetic

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int])
(struct World [x y])
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y))
```

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

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

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

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 WorldState is a (mk-WorldState [x : Int] [y : Int])
(struct World [x y])
(define/contract (mk-WorldState x y)
  (-> integer? integer? WorldState?)
  (World x y))
```

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

```
;; 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**

   (not submitted)

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**

```
;; A WorldState is a (mk-WorldState [x : Int] [y : Int])
(struct World [x y])
```

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

```
(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)))
```

Example + Template helps to write the function!

Extract Compound Pieces – `let`

alternatives

See also `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

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!

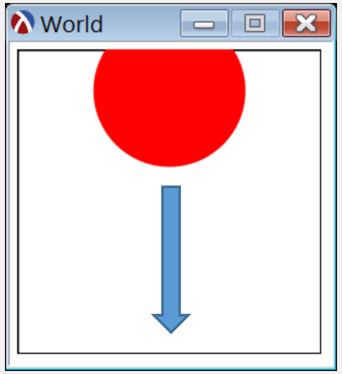
alternatives

```
(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 **two** **pieces of data:**
the **x** *and* **y coordinates**

Last Time

Some Pre-defined Enumerations

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

"KeyEvent" function"

... cannot do "WorldState" function "things"!

(result must be **WorldState**)

WorldState

```
; 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

Or even better: key=?

But remember:

1 function does
1 task which processes
1 kind of data

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

Must call separate: "WorldState fn"

... to do "**WorldState**" **function** "things"

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

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]))
```

Compound Data can be nested

But remember:

1 function does
1 task which processes
1 kind of data

Makes testing easier!

```
(check-equal? (key-handler ANY-GAMESTATE "r")
              (handle-red-key ANY-GAMESTATE))
(check-equal? (key-handler ANY-GAMESTATE "b")
              (handle-blue-key ANY-GAMESTATE))
```

(but still need some **"full stack"** tests)

```
(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]))
```

Function Design Recipe - Testing Update

- "Full Stack"
  - For "**top level**" **functions**
  - Tests: **all functionality** from **input to final result**
  - **Should be more comprehensive**

```
(check-equal? (key-handler SOME-GAMESTATE1 "r")
              NEXT-GAMESTATE2)
(check-equal? (key-handler SOME-GAMESTATE3 "b")
              NEXT-GAMESTATE4)
```

- "Incremental"
  - For "**helper**" (and top level) **functions**
  - Tests: more **local functionality – dictated by data design**
  - **delegate** to **other helpers**, should **test "control flow" paths**

```
(check-equal? (key-handler ANY-GAMESTATE "r")
              (handle-red-key ANY-GAMESTATE))
(check-equal? (key-handler ANY-GAMESTATE "b")
              (handle-blue-key ANY-GAMESTATE))
```

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
```

TEMPLATE

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

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

(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
```

```
(define/contract (handle-red-key g)
  (-> GameState? .... )

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

A “GameState” function …

```
;; 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
```

(hypothetically …)

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

Look at type(s) to help fill in template

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))))
```

Don’t do “**Player**” **function** “things” in a “**GameState**” **function!**

A “GameState” function …

```
;; 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
```

(hypothetically …)

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

```
(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))))
```

(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”)

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
```

```
;; Invariant2: p1 "blue" + p2 "blue" <= MAX-TOKENS
```

"invariant" = "must always be true!"

Can these be "**any**" `Player` values?

Assume hypothetically …

```
;; A Player is a
;;   (mk-Player [red : TokenCount]
;;              [blue : TokenCount])
```

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

Can this be automatically checked?

Previously

```
(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
```

```
;; Invariant2: p1 "blue" + p2 "blue" <= MAX-TOKENS
```

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

Can this be automatically checked?

One possibility:
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)))
```

With Invariant Check

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

```
;; Invariant2: p1 “blue” + p2 “blue” <= MAX-TOKENS
```

But remember:

- **compound data contracts** should be **“shallow”**
- i.e., **don’t traverse** an **entire** (nested) **data structure**
- **Programmer must decide** what is **“too deep”**

This one **probably ok** because … still **“constant” time check**

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

“Dependent” Contract

Can this be automatically checked?

One possibility:
define a separate “**output**” **predicate**

```
(define (GameState/invariant? x)
  (and (GameState? x)
       (GameState-red-invariant? x)
       (GameState-bluue-invariant? x)))
```

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

(with better-named **helper functions**!
Avoid large unreadable boolean expressions!)

Data Design Recipe - Predicate Update

Data Definition (for compound data)

- Has **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** (shallow, conservative approximation of the Data Def)
     - **Evaluates to `true`** for **all values in the Data Def**, and maybe **some not**
       - False positives maybe **ok** — Might let in some invalid values
     - **Evaluates to `false`** for (most?) **values not in the Data Def**, but maybe **not all**
       - False negatives not ok — Must only reject invalid values
     - Consider "dependent" "output" contracts … to check **invariants**
  5. (checked) **Constructor** for **compound data def values**

Follow data definitions whenever possible, but …

Sometimes you need "**if**" … ?

```
(define/contract (key-handler g k)
  (-> GameState? key-event? GameState?)
  (cond
       ....
    [(key=? k SPEND-KEY) (handle-spend g)]
       ....
    [else w]))
```

a "GameState" function!

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
```

```
(define/contract (handle-spend g)
  (-> 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
```

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

Look at type(s) to help fill in template

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-spend 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-spend g)
  (-> GameState? GameState?)
  (mk-GameState
   (Player-fn (GameState-p1 g))
   (Player-fn (GameState-p2 g))
   (PlayerID-fn (GameState-active g))))
```

Don’t do “**Player**” **function** “things” in a “**GameState**” **function!**

A “GameState” function needs “Player” fn

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

(assuming this is “active” player)

A “”Player” function …

```
;; A Player is a
;;  (mk-Player [red : TokenCount]
;;             [blue : TokenCount])
```

Assume hypothetically …

```
(define/contract (player-fn p)
  (-> Player? ....)

         .... (Player-red p) ....)
         .... (Player-blue p) ....))
```

(template)

A “”Player” function …

Are there any invariants to maintain?

Player should only “spend” if they have sufficient tokens, e.g., 1 red and 1 blue

“Dependent” contract won’t suffice here … because we don’t want to error

We really need “`if`”!

```
(define/contract (player-spend p)
  (-> Player? Player?)
  (mk-Player
   (spend-token (Player-red p))
   (spend-token (Player-blue p))))
```

(from hw1)

A “”Player” function …

Are there any invariants to maintain?

Player should only “spend” if they have sufficient tokens, e.g., 1 red and 1 blue

“Dependent” contract won’t suffice here … because we don’t want to error

(not great)
Don’t write this!

No giant boolean expressions!

We really need “if”!

```
(define/contract (player-spend p)
  (-> Player? Player?)
  (if (and (not (zero? (Player-red p)))
           (not (zero? (Player-blue p))))
      (mk-Player
       (spend-token (Player-red p))
       (spend-token (Player-blue p)))
```

A “”Player” function …

Are there any invariants to maintain?

Player should only “spend” if they have sufficient tokens, e.g., 1 red and 1 blue

“Dependent” contract won’t suffice here … because we don’t want to error

(not great)
Don’t write this!

Name doesn’t accurately describe what the function does!

We really need “`if`”!

```
(define/contract (player-spend p)
  (-> Player? Player?)
  (if (can-spend? .... )
      (mk-Player
       (spend-token (Player-red p))
       (spend-token (Player-blue p)))
        ) )
```

A “”Player” function …

Are there any invariants to maintain?

Player should only “spend” if they have sufficient tokens, e.g., 1 red and 1 blue

“Dependent” contract won’t suffice here … because we don’t want to error

Name accurately describes what the function does!

Write this!

We really need “**if**”!

```
(define/contract (player-maybe-spend p)
  (-> Player? Player?)
  (if (can-spend? p)
      (player-spend p)
      p))
```

Function Design Recipe - “`if`” edition

- Avoid if possible …
  - Most of the time, function can follow some data definition template!
- Sometimes needed …
  - E.g., to enforce compound invariants, without error
- Use helper predicate(s) to clearly describe invariant
  - E.g., “`can-spend?`”
  - No huge, unreadable boolean expressions!
- Function name and purpose stmt must indicate “`if`” usage!
  - E.g., “`maybe-`”
- 1 per function only
  - no nested “`if`”s!

Random

Ball Animation

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

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

(not needed for hw … but can be helpful for testing)

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???

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*

Not secure!
e.g., **for generating** passwords

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 cryptography.

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 <u>no examples</u>

```
(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 : -> ???
```