The design recipe

DATA

① Data definition
Give the data definition a name, and state the set of values that are part of it.

② Interpretation
State how values should be interpreted, covering each field/clause if there are multiple of them.

③ Examples
Provide a set of representative examples, building up complex examples iteratively.

④ Template
Provide a template for functions that accept this data definition as input.

FUNCTIONS

① Signature
Give the name of the function, the argument types that it expects as input, and what type it returns.

② Purpose statement
Describe in one sentence what the function does. This should be roughly the length of a tweet.

③ Tests
Provide a set of representative tests, covering different kinds of input and different behaviors.

④ Code
Starting with the template for the input data definition, write the code for the function.

Creating templates

When creating templates for data definitions, there is a recipe to follow:

① How many cases of data are there?
If there is more than one case of data, you will need to use a `cond` *to handle each case separately. If there is only one case, go to step ③.*

② How do I tell them apart?
Determine what distinguishes each case, and use the appropriate predicates for the `cond` *cases.*

③ What data can I pull out?
If the data is complex (e.g., a structure or list), use the appropriate accessors to pull out the information inside. If the data is atomic (e.g., a Number, a String, an Image, or a Boolean), you cannot pull out any more data.

④ Are any of the pieces *another* data definition?
If the data items you pulled out represent another data definition, you should call that data definition's template. Note that self-referential data definitions will have self-referential templates.

`define-struct` functions

`define-struct` *creates a structure for you; it can contain any number of named fields:*

```
(define-struct mystruct [field1 field2 field3 ...])
```

However, it does not *create a data definition; you need to do that separately:*

```
;; A MyStruct is a (make-mystruct Type1 Type2 Type3 ...)
;; Interpretation: ...
;;  - The first field is ...
;;  - The second field is ...
;;  - The third field is ...
;; Examples:
(define MYSTRUCT-1 (make-mystruct ...))
;; Template:
;; (define (mystruct-temp ms)
;;   ... (mystruct-field1 ms) ... (mystruct-field2 ms)
;;   ... (mystruct-field3 ms) ...)
```


`define-struct` functions

`define-struct` *automatically creates a number of functions for you to use:*

CONSTRUCTOR

```
; make-mystruct: Type1 Type2 Type3 -> MyStruct
```

Creates a mystruct structure with the given fields

PREDICATE

```
; mystruct?: Any -> Boolean
```

Returns whether or not the given input a make-mystruct

SELECTORS

```
; mystruct-field1: MyStruct -> Type1
; mystruct-field2: MyStruct -> Type2
; mystruct-field3: MyStruct -> Type3
```

Pulls out the first, second, and third fields of the structure, respectively.

`big-bang` handlers

`big-bang` *handlers are the functions you write that big-bang calls in response to events. The functions, their signatures, and notes about when to use them are below.*

```
(require 2htdp/universe)

(big-bang initial-world
  [to-draw    draw-world]    ; required
  [on-tick    tick-world]    ; optional
  [on-key     handle-key]    ; optional
  [on-mouse   handle-mouse]  ; optional
  [on-receive handle-network]; optional
  [stop-when  should-I-stop?]; optional
 )

; draw-world : World -> Image
; Returns the visualized world as an image (i.e., draws the
; world). This function should not change the world

; tick-world : World -> World
; Produce the next state of the world after the clock ticks.
; This function should not draw any images
```


`big-bang` handlers

```
; handle-key : World KeyEvent -> World
```
Produce the next state of the world after the user presses the given key. A `KeyEvent` *is a String that represents one of the keys. This function should not draw any images.*

```
; handle-mouse : World MouseEvent -> World
```
Produce the next state of the world after the user presses a mouse button. A `MouseEvent` *is one of* `"button-down"`, `"button-up"`, `"drag"`, `"move"`, *or* `"enter"`. *This function should not draw any images*

```
; on-receive: World Message -> World
```
Produce the next state of the world after a message is received from the server. This function should not draw any images.

```
; should-I-stop?: World -> Boolean
```
Determines whether the program should exit, given the current state of the world. If this returns `#true`, *the program exits immediately; if not, it keeps running. This function should not draw any images.*