The design recipe

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

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

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

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

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

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

**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 only one case, go to step 2.
   - If there is more than one case, you will need to use a cond to handle each case separately. If
   - 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.

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 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.

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 mystuct (field1 field2 field3 ...))*

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

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

**big-bang handlers**

Big-bang handlers and associated components: They include big-bang initial-world and world; to-draw is a function that draws the world; and should-I-stop? is a function that determines whether the program should exit; on-key, on-mouse, and on-receive are functions that handle keyboard, mouse, and network messages, respectively.

- **to-draw**: World -> Image
  - (to-draw world)
  - This function should not change the world (i.e., draws the world).

- **on-key**: World KeyEvent -> World
  - (on-key world)
  - This function should not change the world (i.e., draws the world).

- **on-mouse**: World MouseEvent -> World
  - (on-mouse world)
  - This function should not change the world (i.e., draws the world).

- **on-receive**: World Message -> World
  - (on-receive world)
  - This function should not change the world (i.e., draws the world).

- **should-I-stop?**: World -> Boolean
  - (should-I-stop? world)
  - This function should not change the world (i.e., draws the world).

**big-bang calls**

Big-bang calls in response to events. The functions, their signatures, and notes about when to use them are below.

- **to-draw**
  - (to-draw world)
  - Draw world.
- **on-key**
  - (on-key world)
  - Handle key event.
- **on-mouse**
  - (on-mouse world)
  - Handle mouse event.
- **on-receive**
  - (on-receive world)
  - Handle network message.
- **should-I-stop?**
  - (should-I-stop? world)
  - Exit program when #true.