

# Lists of Structures

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 4.3



© Mitchell Wand, 2012-2017

This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

# Introduction

- Lists of structures occur all the time
- Programming with these is no different:
  - write down the data definition, including interpretation and template
  - Follow the Recipe!

# Learning Objectives

- At the end of this lesson you should be able to:
  - write down a template for lists of compound data
  - use the template to write simple functions on lists of compound data

# Programming with lists of structures

- Programming with lists of structures is no different from programming with lists of scalars, except that we make one small change in the recipe for templates

# Example: modeling a bookstore

- Let's imagine a program to help manage a bookstore.
- Let's build a simple model of the inventory of a bookstore.

# Step 1: Data Design

- First, we'll give data definitions for the various quantities we need to represent:

# Preliminary Data Definitions

**;; An Author is represented as a String (any string will do)**

We might refine this definition later, eg keep track of  
FirstName, LastName, etc.

**;; A Title is represented as a String (any string will do)**

**;; An International Standard Book Number (ISBN) is represented**

**;; as a positive integer (PosInt).**

Actually, an ISBN is a sequence of exactly 13 digits, divided into four fields (see [https://en.wikipedia.org/wiki/International\\_Standard\\_Book\\_Number](https://en.wikipedia.org/wiki/International_Standard_Book_Number)). We don't need to represent all this information, so we will simply represent it as a **PosInt**.

**;; A DollarAmount is represented as an integer.**

**;; INTERP: the amount in USD\*100.**

**;; eg: the integer 3679 represents the dollar amount \$36.79**

**;; A DollarAmount may be negative.**

# BookStatus

```
;; A BookStatus is represented as
;; (book-status isbn author title cost price on-hand)

;; INTERP:
;; isbn    : ISBN          -- the ISBN of the book
;; author : Author        -- the book's author
;; title  : Title         -- the book's title
;; cost   : DollarAmount -- the wholesale cost of the book (how much
;;                               the bookstore paid for each copy of the
;;                               book)
;; price  : DollarAmount -- the price of the book (how much the
;;                               bookstore charges a customer for the
;;                               book)
;; on-hand: NonNegInt     -- the number of copies of the book that are
;;                               on hand in the bookstore)
```

Note that we are not modelling a Book (that's something that exists on a shelf somewhere 😊). We are modelling the status of all copies of this book.



# BookStatus (cont'd)

```
;; IMPLEMENTATION:
```

```
(define-struct book-status (isbn author title cost price on-hand))
```

```
;; CONSTRUCTOR TEMPLATE:
```

```
;; (make-book-status ISBN Author Title DollarAmount DollarAmount NonNegInt)
```

```
;; OBSERVER TEMPLATE:
```

```
;; book-status-fn : BookStatus -> ??
```

```
(define (book-status-fn b)  
  (...  
    (book-status-isbn b)  
    (book-status-author b)  
    (book-status-title b)  
    (book-status-cost b)  
    (book-status-price b)  
    (book-status-on-hand b)))
```

# Inventory

```
;; An Inventory is represented as a list of
;;   BookStatus, in increasing ISBN order, with at
;;   most one entry per ISBN.

;; CONSTRUCTOR TEMPLATES:
;; empty
;; (cons bs inv)
;; -- WHERE
;;   bs is a BookStatus
;;   inv is an Inventory
;;   and
;;   (bookstatus-isbn bs) is less than the ISBN of
;;   any book in inv.
```

# Inventory (cont'd)

```
;; OBSERVER TEMPLATE:
```

```
;; inv-fn : Inventory -> ??
```

```
(define (inv-fn inv)
```

```
  (cond
```

```
    [(empty? inv) ...]
```

```
    [else (...
```

```
      (first inv)
```

```
      (inv-fn (rest inv)))]))
```

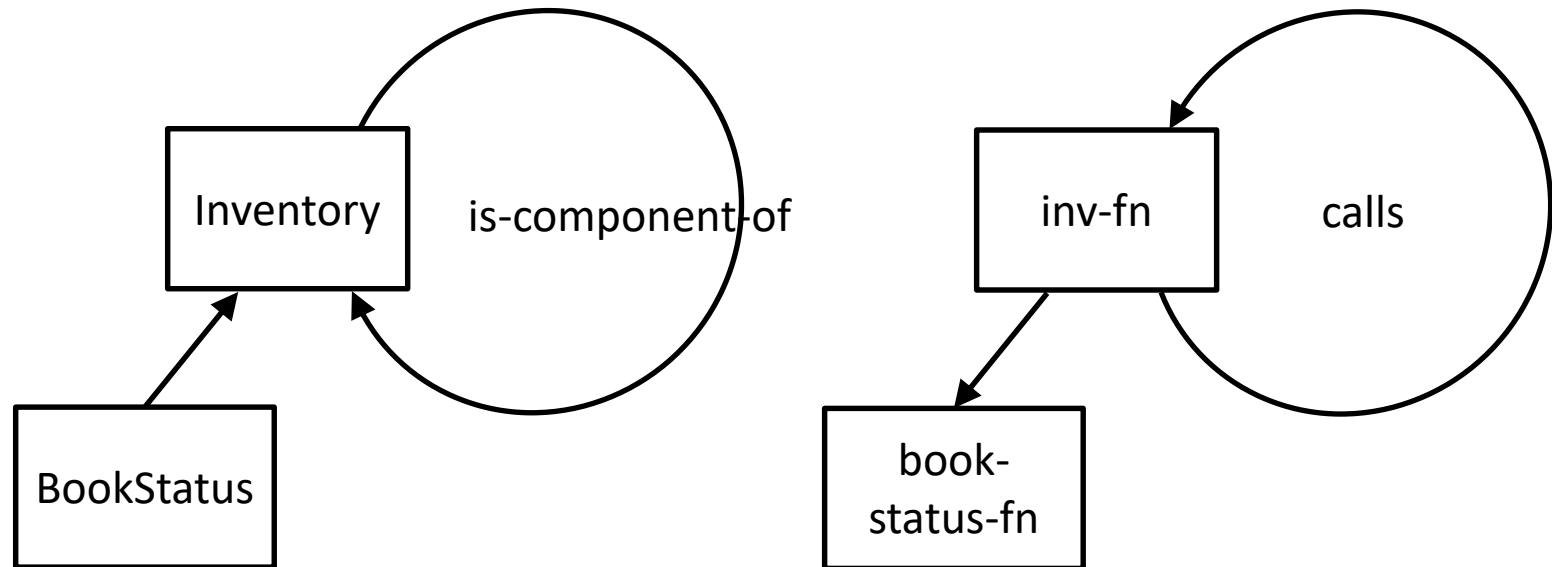
# Inventory (cont'd)

```
(define (inv-fn inv)
  (cond
    [(empty? inv) ...]
    [else (...
             (book-status-fn (first inv))
             (inv-fn (rest inv)))]))
```

Since (first inv) is a BookStatus, it would also be OK to write the observer template like this. These templates are there to serve as a guide for you, so we are going to try not to be too picky about them.

But you *must* put the recursive call to **inv-fn** in your observer template.

# Remember: The Shape of the Program Follows the Shape of the Data



Data Hierarchy (a non-empty inventory contains a BookStatus and another Inventory)

Call Tree (**inv-fn** calls itself and **book-status-fn**)

# Example function: **inventory-authors**

```
;; inventory-authors : Inventory -> AuthorList
;; GIVEN: An Inventory
;; RETURNS: A list of the all the authors of the books in the
;; inventory. Repetitions are allowed. Books with no copies in stock
;; are included. The authors may appear in any order.
;; EXAMPLE: (inventory-authors inv1)
;;           = (list "Felleisen" "Wand" "Shakespeare" "Shakespeare")
;; STRATEGY: Use observer template for Inventory
```

```
(define (inventory-authors inv)
  (cond
    [(empty? inv) empty]
    [else (cons
            (book-status-author (first inv))
            (inventory-authors (rest inv)))]))
```

# An Inventory– but *which* inventory?

- So far we've decided how to represent an inventory.
- But what store is it the inventory of?
- And what date does it represent?

# BookstoreState

```
;; A Date is represented as a ....

;; A BookstoreState is represented as a (bookstore-state date stock)
;; INTERP:
;; date   : Date           -- the date we are modelling
;; stock  : Inventory      -- the inventory of the bookstore as of 9am ET on
;;                               the given date.

;; IMPLEMENTATION:

(define-struct bookstore-state (date stock))

;; CONSTRUCTOR TEMPLATE
;; (make-bookstore-state Date Inventory)

;; OBSERVER TEMPLATE
;; state-fn : BookstoreState -> ??
(define (state-fn bss)
  (... (bookstore-state-date bss)
       (bookstore-state-stock bss)))
```

Now that we have a history of the inventory, we can do more things, like track the value of the inventory over time, compare the sales of some book over some time period, etc., etc.



# Module Summary: Self-Referential or Recursive Information

- Represent arbitrary-sized information using a *self-referential* (or *recursive*) data definition.
- Self-reference in the data definition leads to self-reference in the observer template.
- Self-reference in the observer template leads to self-reference in the code.
- Writing functions on this kind of data is easy: just Follow The Recipe!
- But get the template right!

# Summary

- At the end of this lesson you should be able to:
  - write down a template for lists of compound data
  - use the template to write simple functions on lists of compound data
- The Guided Practices will give you some exercise in doing this.

# Next Steps

- Study 04-2-books.rkt in the Examples file
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 4.4
- Go on to the next lesson