Lists of Structures

CS 5010 Program Design Paradigms

Lesson 4.3

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.
- We'd like to know which books sell and when they sold.
- To do this we've decided to keep track of the state of the bookstore, which is a date and the inventory of books in stock on that date.

Books

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

;; A Book is a
;;  (make-book NonNegInt String String NonNegInt NonNegInt)
;; Interpretation:
;; --isbn is the ISBN of the book
;; --author is the author's name
;; --title is the title
;; --on-hand is the number of copies on hand
;; --price is the price in USD*100

;; book-fn : Book -> ??
;; (define (book-fn b)
;;   (... (book-isbn b) (book-author b
;;        (book-title b) (book-on-hand
;;        (book-price b)))
```

Here is the data definition for a book in a bookstore, with structure definition, data definition, interpretation, and template.

BookstoreState

```
;; A BookstoreState is a
;; (make-bookstore-state Date Inventory)
(define-struct bookstore-state (date inventory))
;; A BookstoreState represents the state of a bookstore
;; on a given date.
;; -- date is the date being described
;; -- inventory is the inventory of books as of 9am on
;;    the given date.

;; A Date is a ...

;; An Inventory is a ListOfBook, in ISBN order
```

Here's where we specify the order of the books. It is the *user* of the list who gets to specify the order in which the items appear. Other functions that use ListOfBook might expect the books in some other order.

If you don't know what an ISBN is, go look it up.

ListOfBook

```
;; A ListOfBook one of
;; -- empty
;; -- (cons Book ListOfBook)
```

Don't need a separate interpretation for ListOfBook– a ListOfBook always represents a sequence of books in the standard way.

Notice that the data definition doesn't say WHICH list of books this is. It could be all the books in the bookstore, just the paperbacks, the ones that have been ordered in the last 30 days, etc. etc. In a BookstoreState, it is the list of all the books in stock as of 9am on the given state.

Also, the data definition doesn't say in which order the books appear in the list. A user of ListOfBook gets to specify the order in which the books appear. In our example, BookstoreState expects the books to appear in ISBN order.

Template for ListofBooks

```
;; A ListOfBooks (LOB) is either
;; -- empty
;; -- (cons Book LOB)

;; lob-fn : LOB -> ??
;; (define (lob-fn lob)
;;   (cond
;;     [(empty? lob) ...]
;;     [else (...
;;             (book-fn (first lob))
;;             (lob-fn (rest lob)))]))
```

We've inserted a call to a function here.

(rest lob) is a LOB, so we wrap it in a lob-fn.

Similarly, (first lob) is a Book, so we wrap it in a book-fn.

The template recipe, updated

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many cond clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate ``natural recursions'' for the template to represent the self-references of the data definition.
Do any of the fields contain compound or mixed data?	If the value of a field is a foo, add a call to a foo-fn to use it.

Observe that this is just what we did for self-references, because a list is a kind of mixed data.

Example: if book-fn is just a selector, you can put it in directly

```
;; books-authors : LOB -> ListOfString
;; STRATEGY: Use template for LOB on lob
(define (books-authors lob)
  (cond
    [(empty? lob) empty]
    [else (cons
           (book-author (first lob))
           (books-authors (rest lob)))]))
```

book-author is certainly a **book-fn**!

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 template.
- Self-reference in the 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