The Observer Template for List Data

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 4.2

Key Points for Lesson 4.2

At the end of this lesson you should be able to:

- Write down the observer template for list data.
- Use the observer template for list data to write simple functions on lists.

Review: The Constructor Templates for XList

```
A XList is one of
-- empty
-- (cons X XList)
```

Here are the constructor templates for a list of X's. This means that any XList must look like one of these two forms

This definition is self-referential

```
A XList is one of
-- empty
-- (cons X XList)
```

Here are the constructor templates for a list of X's. This means that any XList must look like one of these two forms

Observer Template (1st attempt)

```
;; xlist-fn: XList -> ??
(define (xlist-fn xs)
  (cond
    [(empty? xs) ...]
    [else (... (first xs)
               (rest xs))]))
```

But we should do something cleverer!

- The constructor template was self-referential
- But this wasn't reflected in our observer template.
- **(rest xs)** is an XList, so we should expect to call **xlist-fn** recursively on it.
  - This is usually (though not always) what you want.

The Observer Template for List data

```
;; xlist-fn : XList -> ??
(define (xlist-fn xs)
  (cond
    [(empty? xs) ...]
    [else (... (first xs)
               (xlist-fn (rest xs)))]))
```

Here we add a recursive call to **list-fn** on **(rest xs)** .

Observe that **xs** is non-empty when **first** and **rest** are called, so their contracts are satisfied.

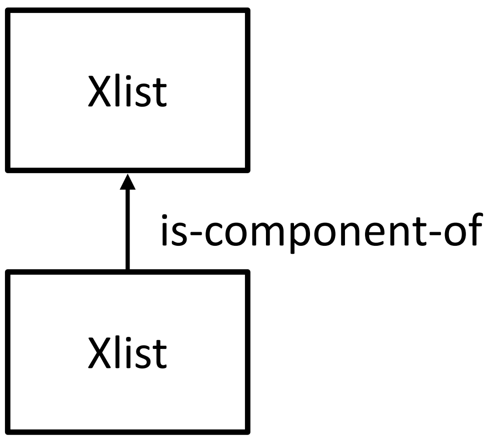
This template is *self-referential*

```
;; list-fn : XList -> ??
(define (xlist-fn xs)
  (cond
    [(empty? xs) ...]
    [else (... (first xs)
               (xlist-fn (rest xs)))]))
```

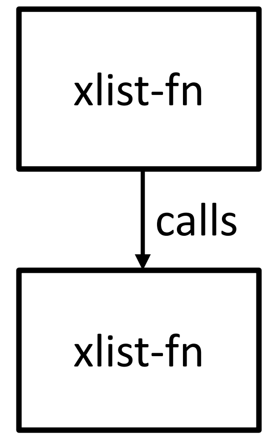
(rest xs) is a **XList**, so call **xlist-fn** on it

New Slogan: Self-reference in the constructor template leads to self-reference in the observer template.

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



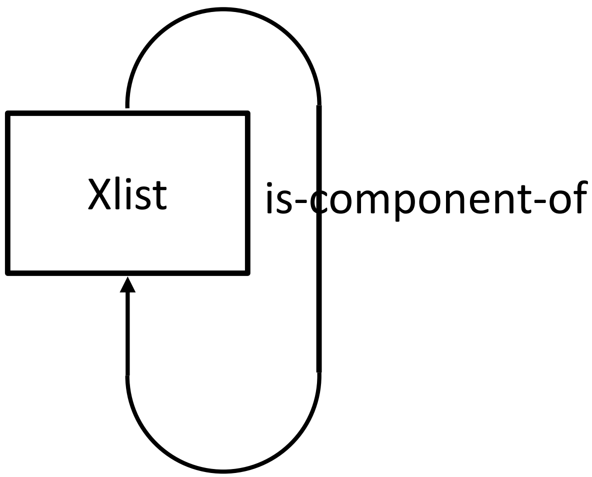
Data Hierarchy (a non-empty Xlist contains another Xlist)

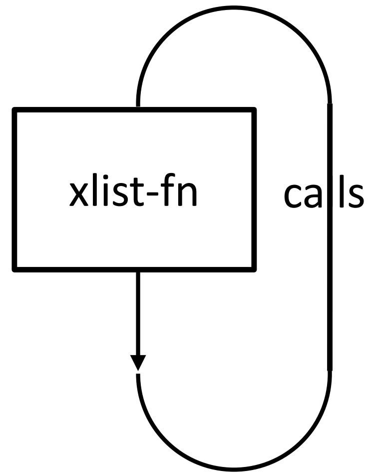


Call Tree (**xlist-fn** calls itself on the component)

or, folding in the recursion....:

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





Data Hierarchy (a non-empty Xlist contains another Xlist)

Call Tree (**xlist-fn** calls itself on the component)

From Observer Template to Function Definition

- The observer template has two blanks in it.
- Often we can get our function definition by simply filling in the blanks.
- Each blank corresponds to a question
- It's the same question for every function:

Let’s do some examples

- We’ll be working with the list template a lot, so let’s do some examples to illustrate how it goes.
- We’ll do 5 examples, starting with one that’s very simple and working up to more complicated ones.

Here are the questions for the XList template:

```
;; xlist-fn : XList -> ??
(define (xlist-fn xs)
  (cond
    [(empty? xs) ...]
    [else (... (first xs)
               (xlist-fn (rest xs)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Data Definitions

```
;; A NumberList is represented as a list of Number.

;; CONSTRUCTOR TEMPLATE AND INTERPRETATION
;; empty                    -- the empty sequence
;; (cons n ns)
;;   WHERE:
;;    n  is a Number        -- the first number
;;                             in the sequence
;;    ns is a NumberList    -- the rest of the
;;                             numbers in the sequence

;; OBSERVER TEMPLATE:
;; nl-fn : NumberList -> ??
(define (nl-fn lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (nl-fn (rest lst)))]))
```

Example 1: nl-length

```
nl-length : NumberList -> Number
GIVEN: a NumberList
RETURNS: its length
EXAMPLES:
(nl-length empty) = 0
(nl-length (cons 11 empty)) = 1
(nl-length (cons 33 (cons 11 empty))) = 2
STRATEGY: Use template for NumberList on lst
```

We write "**nl-**" as an abbreviation for **NumberList**

Example 1: nl-length

```
;; nl-length : NumberList -> Number
;; Given a NumberList, find its length
(define (nl-length lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (nl-length (rest lst)))]))
```

We start by copying the template and changing the name of the function to **nl-length**.

Example 1: nl-length

```
;; nl-length : NumberList -> Number
;; Given a NumberList, find its length
(define (nl-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
              (nl-length (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Next, we answer the template questions.

The code is self-referential, too

```
;; nl-length : NumberList -> Number
;; Given a NumberList, find its length
(define (nl-length lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (first lst)
               (nl-length (rest lst)))]))
```

Self-reference in the constructor template leads to self-reference in the observer template;
Self-reference in the observer template leads to self-reference in the code.

Let's watch this work

```
(nl-length (cons 11 (cons 22 (cons 33 empty))))
= (+ 1 (nl-length (cons 22 (cons 33 empty))))
= (+ 1 (+ 1 (nl-length (cons 33 empty))))
= (+ 1 (+ 1 (+ 1 (nl-length empty))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
= (+ 1 2)
= 3
```

See how each recursive call to **nl-length** works on a shorter and shorter list.

Example 2: nl-sum

```
nl-sum : NumberList -> Number
GIVEN: a list of numbers
RETURNS: the sum of the numbers in the list
EXAMPLES:
(nl-sum empty) = 0
(nl-sum (cons 11 empty)) = 11
(nl-sum (cons 33 (cons 11 empty))) = 44
(nl-sum (cons 10 (cons 20 (cons 3 empty)))) = 33
STRATEGY: Use template for NumberList on lst
```

Here's another example

Example 2: nl-sum

```
;; nl-sum : NumberList -> Number
(define (nl-sum lst)
  (cond
    [(empty? lst) 0].]
    [else (+.. (first lst)
               (nl-sum (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Watch this work:

```
(nl-sum (cons 11 (cons 22 (cons 33 empty))))
= (+ 11  (nl-sum (cons 22 (cons 33 empty))))
= (+ 11  (+ 22    (nl-sum (cons 33 empty))))
= (+ 11  (+ 22    (+ 33    (nl-sum empty))))
= (+ 11  (+ 22    (+ 33    0)))
= (+ 11  (+ 22    33))
= (+ 11  55)
= 66
```

Example 3: double-all

```
double-all : NumberList -> NumberList
GIVEN: a NumberList,
RETURNS: a sequence just like the original, but
         with each number doubled
EXAMPLES:
(double-all empty) = empty
(double-all (cons 12 empty))
          = (cons 24 empty)
(double-all (cons 33 (cons 12 empty)))
          = (cons 66 (cons 24 empty))
STRATEGY: Use template for NumberList on lst
```

Example 3: double-all

```
;; double-all : NumberList -> NumberList
(define (double-all lst)
  (cond
    [(empty? lst) empty ]
    [else (cons (* 2 (first lst))
               (double-all (rest lst)))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Watch this work:

```
(double-all (cons 12 (cons 22 (cons 33 empty))))
= (cons 24 (double-all (cons 22 (cons 33 empty))))
= (cons 24 (cons 44 (double-all (cons 33 empty))))
= (cons 24 (cons 44 (cons 66 (double-all empty))))
= (cons 24 (cons 44 (cons 66 empty)))
```

Example 4: remove-evens

- For this one, we'll need to specialize to integers.

```
An IntList is one of
-- empty
-- (cons Integer IntList)
```

Example 4: remove-evens

remove-evens is not a perfect name for this function, since it's a verb rather than a noun.

```
remove-evens : IntList -> IntList
GIVEN: a IntList,
RETURNS: a list just like the original, but with all
 the even numbers removed
EXAMPLES:
(remove-evens empty) = empty
(remove-evens (cons 12 empty)) = empty
(define list-22-11-13-46-7
  (cons 22 (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-evens list-22-11-13-46-7)
  = (cons 11 (cons 13 (cons 7 empty)))
STRATEGY: Use observer template for IntList
```

Example 4: remove-evens

```
remove-evens : IntList -> IntL
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (remove-evens (rest lst))
              (cons (first lst)
                    (remove-evens (rest lst))))]))
```

What's the answer for the empty list?

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 4: remove-evens

```
remove-evens : IntList -> IntList
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (remove-evens (rest lst))
              (cons (first lst)
                    (remove-evens (rest lst))))]))
```

This code seems a little complicated. Could we make it more readable?

Example 4: remove-evens

```
remove-evens : IntList -> IntList
(define (remove-evens lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (remove-evens (rest lst))]
    [else (cons (first lst)
                (remove-evens (rest lst)))]))
```

Here's a clearer version, which is also acceptable for this class. The template is just a way for you to get started writing your function definition. It's OK to vary it a little if it leads to more readable code.

Example 5: remove-first-even

```
remove-first-even : IntList -> IntList
GIVEN: a IntList,
RETURNS: a list just like the original, but with all the
  even numbers removed
EXAMPLES:
(remove-first-even empty) = empty
(remove-first-even (cons 12 empty)) = empty
(define list-22-11-13-46-7
  (cons 22 (cons 11 (cons 13 (cons 46 (cons 7 empty))))))
(remove-first-even list-22-11-13-46-7)
  = (cons 11 (cons 13 (cons (cons 46 (cons 7 empty))))))
STRATEGY: Use template for IntList on lst
```

Why is this not a good set of examples?

Answer: None of them show what happens when the first element of the list is odd

Example 5: remove-first-even

```
remove-first-even : IntList -> IntList
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [else (if (even? (first lst))
              (rest lst)
              (cons (first lst)
                    (remove-first-even (rest lst))))]))
```

What's the answer for the empty list?

This is OK: you don't have to recur if you don't need to.

If we knew the first of the list, and the answer for the rest of the list, how could we combine them to get the answer for the whole list?

Example 5: remove-first-even

```
remove-first-even : IntList -> IntList
(define (remove-first-even lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (rest lst)]
    [(cons (first lst)
           (remove-first-even (rest lst)))]))
```

Again, here's another version of remove-first-even that is acceptable. It's OK to vary the template, but you'll be less likely to make mistakes if you stick close to the template.

Example 6: insert

```
;; DATA DEFINITION

;; A SortedIntList is an IntList that is in ascending
;; order.
```

This assumes that we already have a definition for **IntList**.

Example 6: insert

```
;; insert : Integer SortedIntList -> SortedIntList
;; GIVEN: An integer and a sorted sequence of integers
;; RETURNS: A new SortedIntList just like the
;;   original, but with the new integer inserted.
;; EXAMPLES:
;; (insert 3 empty) = (list 3)
;; (insert 3 (list 5 6)) = (list 3 5 6)
;; (insert 3 (list -1 1 5 6))
;;    = (list -1 1 3 5 6)
;; STRATEGY: Use observer template for
;; SortedIntList
```

Function Definition for **insert**

```
(define (insert n seq)
  (cond
    [(empty? seq) (cons n empty)]
    [(< n (first seq)) (cons n seq)]
    [else (cons (first seq)
                (insert n (rest seq)))]))
```

Watch this work:

```
(insert 27 (cons 11 (cons 22 (cons 33 empty))))
= (cons 11 (insert 27 (cons 22 (cons 33 empty))))
= (cons 11 (cons 22 (insert 27 (cons 33 empty))))
= (cons 11 (cons 22 (cons 27 (cons 33 empty))))
```

Observe that this computation may take time proportional to the length of the sequence (in this case, 3).

Example 7: Insertion Sort

```
;; mysort : IntList -> SortedIntList
;; GIVEN: An integer sequence
;; RETURNS: The same sequence,
;;  but sorted by <= .
;; EXAMPLES:
;; (mysort empty) = empty
;; (mysort (list 3)) = (list 3)
;; (mysort (list 2 1 4)) = (list 1 2 4)
;; (mysort (list 2 1 4 2)) = (list 1 2 2 4)
;; STRATEGY: Use observer template for
;;  IntList
```

sort is predefined in ISL, so we need to use a different name.

Function definition for **mysort**

```
(define (mysort ints)
  (cond
    [(empty? ints) empty]
    [else (insert (first ints)
                  (mysort (rest ints)))]))
```

The second argument to **insert** is always supposed to be a **SortedIntList**. Why is this true? (Hint: look at the contract for **mysort**.)

Watch this work:

```
  (mysort (list 2 1 4 2))
= (insert 2 (mysort (list 1 4 2)))
= (insert 2 (insert 1 (mysort (list 4 2))))
= (insert 2 (insert 1 (insert 4 (mysort (list 2)))))
= (insert 2 (insert 1 (insert 4 (insert 2 (mysort empty)))))
= (insert 2 (insert 1 (insert 4 (insert 2 empty))))
= (insert 2 (insert 1 (insert 4 (list 2))))
= (insert 2 (insert 1 (list 2 4)))
= (insert 2 (list 1 2 4)))
= (list 1 2 2 4))
```

How many steps does this take?

- If you call **mysort** on a list of length N, it will take N steps to get to the end, leaving N calls to **insert** still to be executed.
- Each call to **insert** takes a number of steps proportional to the length of its argument, which again can be of length N.
- There are N calls to **insert**, so the whole computation takes time proportional to N^2.
- This can all be made precise; you should have learned this in your undergraduate algorithms course.

Summary

- You should now be able to:
  - write down the template for a list data definition
  - use structural decomposition to define simple functions on lists

Next Steps

- Study 04-1-lists.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson