From Templates to Folds

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 6.6

Introduction

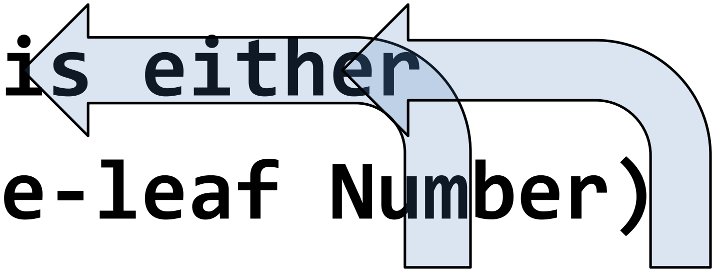
- Last week, we saw how the built-in mapping functions on lists, like **map**, **filter**, and **foldr**, made writing functions on lists easier.
- In this lesson we'll see how we can do something similar for any recursive data definition.

Learning Objectives

- At the end of this lesson you should be able to:
  - Write a fold function for any recursive data definition
  - Use the fold function to define useful functions on that data

Binary Trees

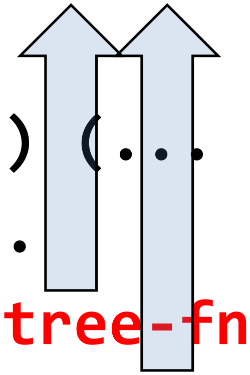
```
(define-struct leaf (datum))
(define-struct node (lson rson))
```



```
;; A Tree is either
;; -- (make-leaf Number)
;; -- (make-node Tree Tree)
```

Here is the definition of a binary tree again.

Template



```
tree-fn : Tree -> ???
(define (tree-fn t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
            (tree-fn (node-lson t))
            (tree-fn (node-rson t)))]))
```

And here is the template again.

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.

..or..

The shape of the program follows the shape of the data

The template has two blanks

```
tree-fn : Tree -> ???
(define (tree-fn t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
              (tree-fn (node-lson t))
              (tree-fn (node-rson t) ))]))
```

Two blanks: one blue and one orange

From templates to folds

- Observe that the template has two blanks: the blue one and the orange one.
- Any two functions that follow the template will be the same except for what goes in the blanks.
- So we can generalize them by adding arguments for each blank.

Template ➔ tree-fold

```
tree-fold : ... Tree -> ???
(define (tree-fold combiner base t)
  (cond
    [(leaf? t) (base (leaf-datum t))]
    [else (combiner
            (tree-fold combiner base
              (node-lson t))
            (tree-fold combiner base
              (node-rson t)))]))
```

Corresponding to each blank, we add an extra argument: **combiner** (in blue) for the blue blank and **base** (in orange) for the orange blank, and we pass these arguments to each of the recursive calls, just like we did for lists. The strategy for tree-fold is "Use observer template for Tree on t"

What's the contract for tree-fold?

Let's figure out the contract for tree-fold. Let's analyze the

Let's assume the whole function returns an X.

```
tree-fold
  :   contract for       contract for     Tree -> X
        combiner             base
(define (tree-fold combiner base t)
  (cond
    [(leaf? t) (base (leaf-datum t))]
    [else (combiner
           (tree-fold combiner base
             (node-lson t))
           (tree-fold combiner base
             (node-rson t)))]))
```

X

(Number -> X)

(X X -> X)

X

X

If the whole function returns an **X**, then **(base (leaf-datum t))** must return an **X**.

(leaf-datum t) returns a number, and **(base (leaf-datum t))** must return an **X**, so **base** must be **(Number -> X)**

Since **tree-fold** returns an **X**, the arguments to **combiner** are both **X**'s, and **combiner** itself must return an **X**.

So combiner must be an **(X X -> X)**

Be sure to reconstruct the original functions!

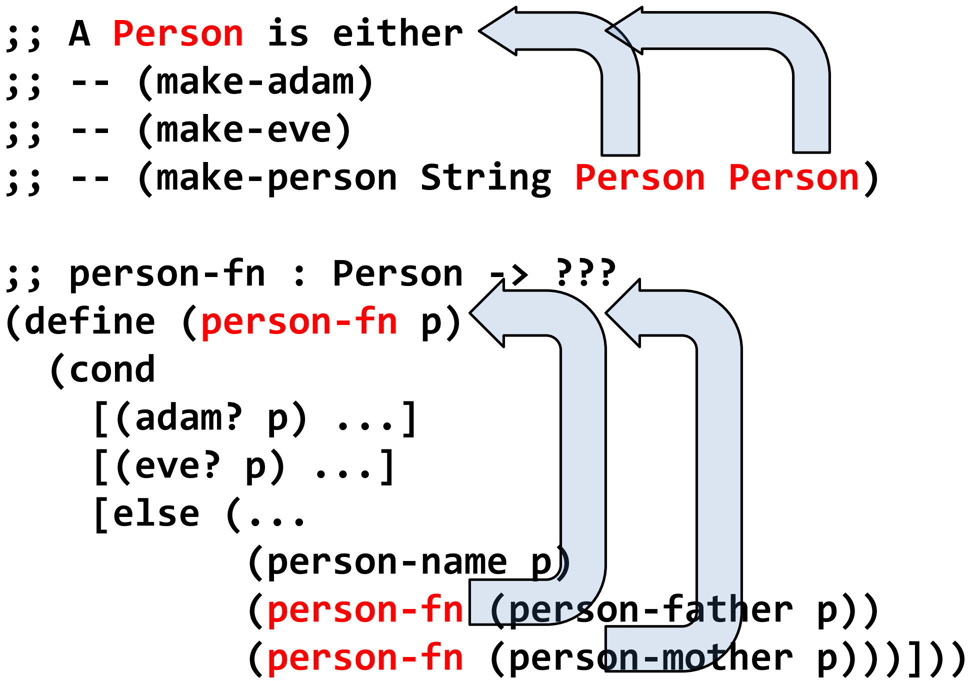
```
(define (tree-sum t)
  (tree-fold + (lambda (n) n) t))

(define (tree-min t)
  (tree-fold min (lambda (n) n) t))

(define (tree-max t)
  (tree-fold max (lambda (n) n) t))
```

Here are our original functions, **sum**, **tree-min**, and **tree-max**, rewritten using **tree-fold**.

The strategy for each of these is "Call a more general function."

Another example of trees: Ancestor Trees

```
(define-struct person (name father mother))
(define-struct adam ())
(define-struct eve  ())
```



```
;; A Person is either
;; -- (make-adam)
;; -- (make-eve)
;; -- (make-person String Person Person)

;; person-fn : Person -> ???
(define (person-fn p)
  (cond
    [(adam? p) ...]
    [(eve? p) ...]
    [else (...
           (person-name p)
           (person-fn (person-father p))
           (person-fn (person-mother p)))]))
```

The Structure of the Program Follows the Structure of the Data

Template for Person

```
;; person-fn : Person -> ???
(define (person-fn p)
  (cond
    [(adam? p) ...]
    [(eve? p) ...]
    [else (...
            (person-name p)
            (person-fn (person-father p))
            (person-fn (person-mother p)))]))
```

Here's the template for our ancestor trees. We have three blanks: one blue, one purple, and one orange.

From template to fold:

```
;; person-fold : ... Person -> ???
(define (person-fold adam-val eve-val combiner p)
  (cond
    [(adam? p) adam-val]
    [(eve? p) eve-val]
    [else (combiner
            (person-name p)
            (person-fold adam-val eve-val combiner
             (person-father p))
            (person-fold adam-val eve-val combiner
             (person-mother p)))]))
```

Corresponding to our three blanks we add three arguments: the value for **adam** (in blue), the value for **eve** (in purple) and the **combiner** (in orange).

What's the contract for person-fold?

We can work out the contract for **person-fold** the same way that we did for **tree-fold**. Here again we've marked some of the sub-expressions with the kind of value they return.

```
;; person-fold
;;  : X X (String X X -> X) Person -> X
(define (person-fold adam-val eve-val combiner p)
  (cond
    [(adam? p) adam-val]
    [(eve? p) eve-val]
    [else (combiner
           (person-name p)
           (person-fold adam-val eve-val combiner
            (person-father p))
           (person-fold adam-val eve-val combiner
            (person-mother p)))]))
```

X

(String X X -> X)

X

String

X

Observe, as before, that the arguments to **combiner** match **combiner**'s contract, and that all three branches of the **cond** return an **X**, so the whole function is guaranteed to return an **X**.

Summary

- You should be able to:
  - Write a fold function for any recursive data definition
  - Use the fold function to define useful functions on that data

Next Steps

- Study the file 06-6-tree-folds.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practices 6.6 and 6.7
- Do Problem Set 6