More About Recursive Data Types

CS 5010 Program Design Paradigms “Bootcamp”

Lesson 5.5

© Mitchell Wand, 2016-2017
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Introduction

- We've just seen several examples of data definitions for recursive data
- In this lesson we'll learn more about the characteristics of a good data type definition and explore how functions on a recursive data type work.

Key Points for Lesson 5.6

- At the end of this lesson you should be able to
  - list the properties that a data type with multiple constructors must have.
  - illustrate how a recursive function that follows the observer template always calls itself on smaller and smaller structures.

Multiple constructors

- In order to define recursive data, we introduced data definitions with *multiple constructors*, e.g.:

```
;; CONSTRUCTOR TEMPLATES:
;; empty
;; (cons bs inv)
;;      -- WHERE ...
```

```
;; CONSTRUCTOR TEMPLATES:
;; -- (make-leaf Number)
;; -- (make-node Tree Tree)
```

```
CONSTRUCTOR TEMPLATES
for NonEmptyXList:
-- (cons X empty)
-- (cons X NonEmptyXList)
```

Properties of a good data definition for recursive data

- There are one or more *base cases* that do not use recursion
- The cases are *mutually exclusive*
- It is easy to tell the alternatives apart
- There is one and only one way of building any value.
- There is an interpretation for each case.

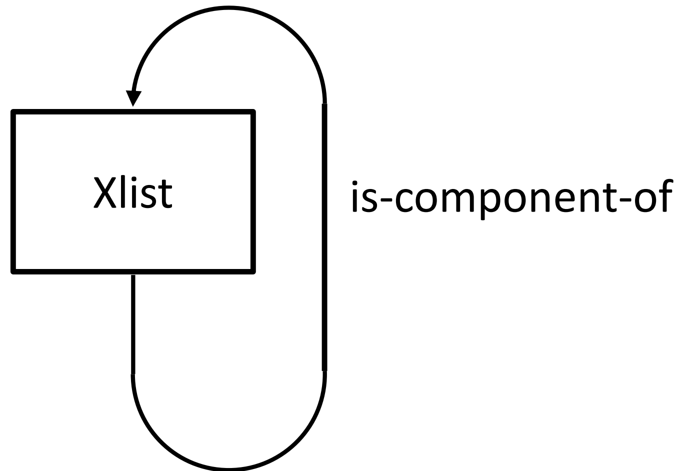
Did our definitions have these properties?

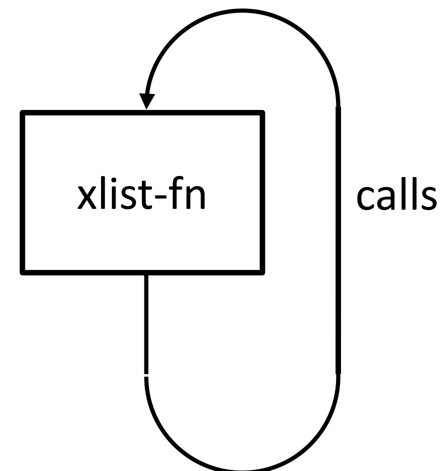
- Go back now and check to see that our definitions have these properties.
- All the definitions we (and you) will write will have these properties.

Observer templates always recur on smaller pieces of data

- Our observer templates embody the principle that the shape of the program follows the shape of the data.
- This means that our functions always recur on smaller pieces of the data.
- Let’s first see what that means for lists

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





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

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

Let’s watch this in action: remember **nl-sum**

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

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
```

Clearly, this function will halt for any NumberList

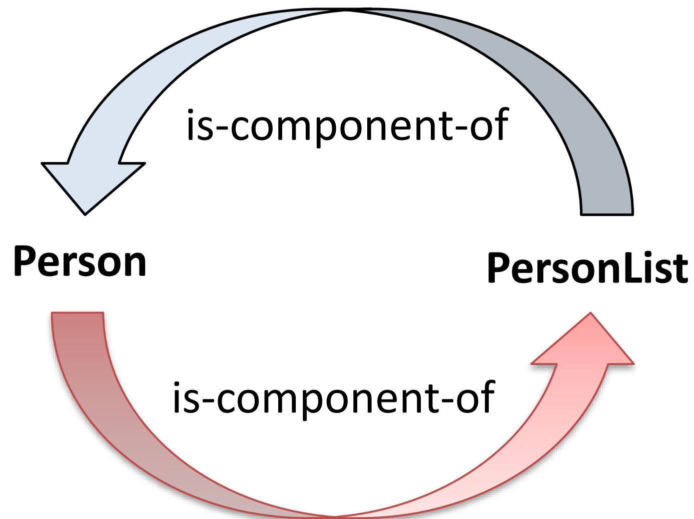
- Why?
- Because at every step it works on a shorter and shorter list, so eventually it reaches **empty?** and the function halts.

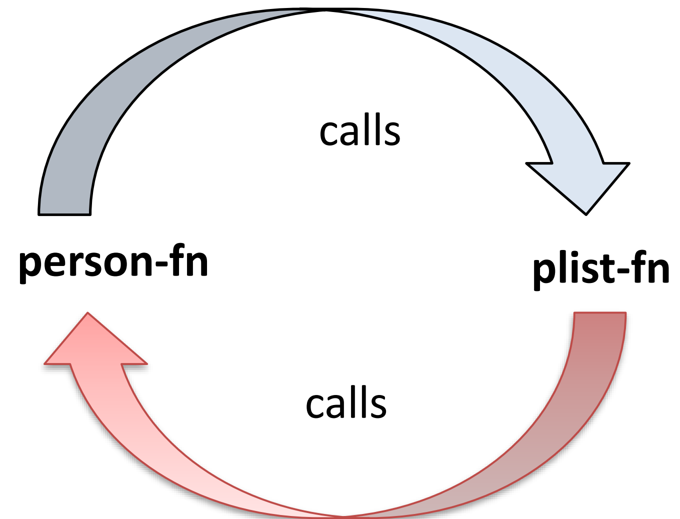
Let's try something more complicated

- What about descendant trees?
- Remember the constructor templates:

```
;; CONSTRUCTOR TEMPLATES:
;; For Person:
;;   (make-person String PersonList)
;; For PersonList:
;;   empty
;;   (cons Person PersonList)
```

The Shape of the Program Follows the Shape of the Data: Descendant Trees





Data Hierarchy: **Person** contains **PersonList** as a component; and **PersonList** has **Person** as a component

Call Tree: **person-fn** calls **plist-fn**, and **plist-fn** calls **person-fn**.

Template: functions come in pairs

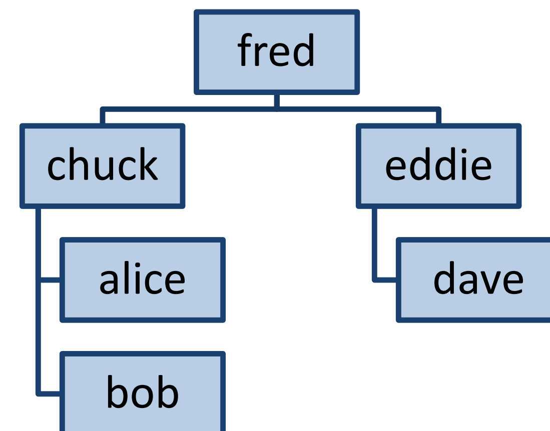
```
;; person-fn : Person -> ??
(define (person-fn p)
  (... (person-name p)
       (plist-fn (person-children p))))

;; plist-fn : PersonList -> ??
(define (plist-fn ps)
  (cond
    [(empty? ps) ...]
    [else (... (person-fn (first ps))
               (plist-fn (rest ps)))]))
```

Here is the pair of templates that we get by applying the recipe to our data definition.

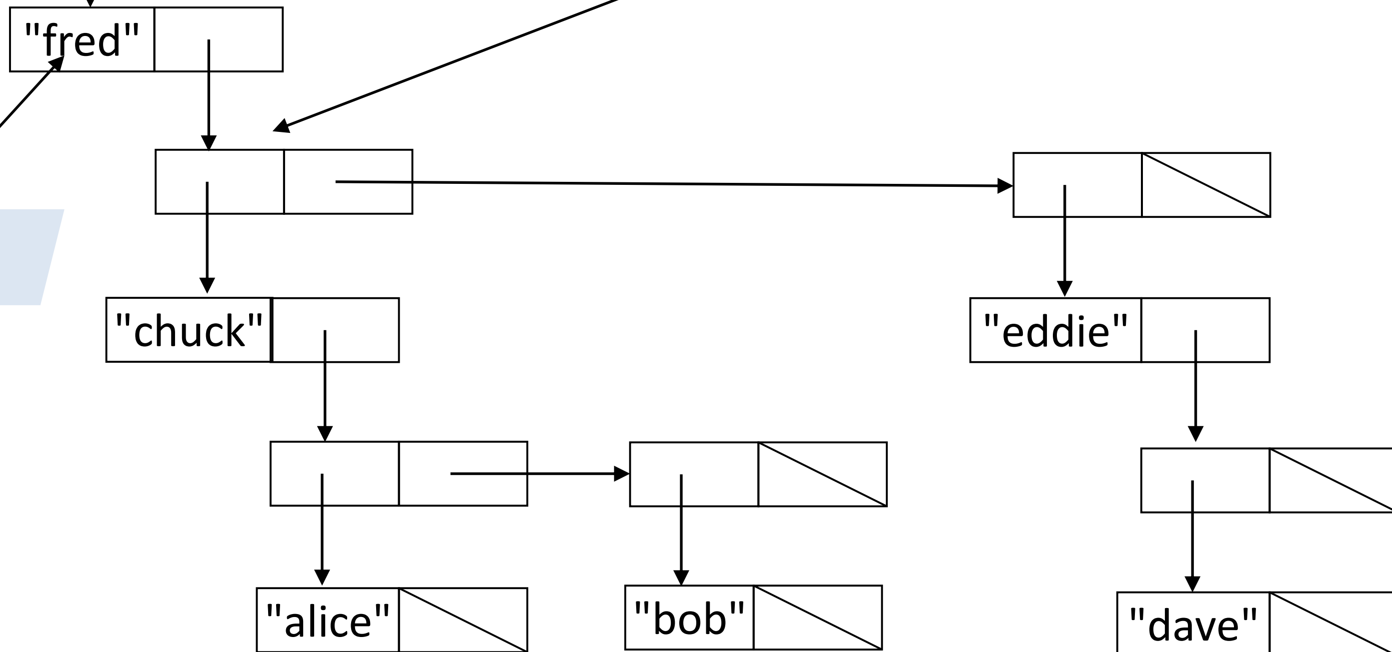
They are mutually recursive, as you might expect.

Let’s look at our old example again

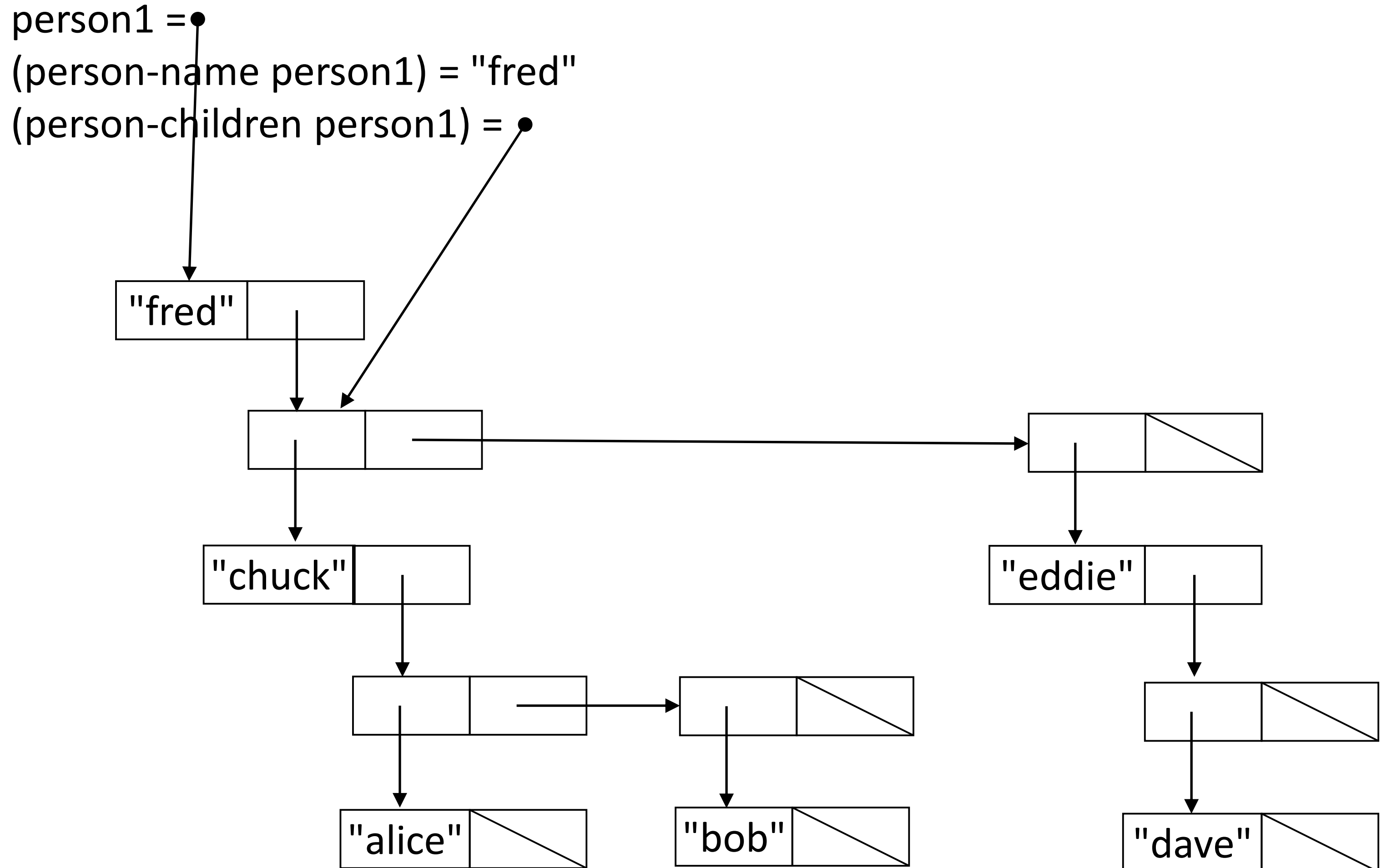
```
(define alice (make-person "alice" empty))
(define bob (make-person "bob" empty))
(define chuck (make-person "chuck" (list alice bob)))

(define dave (make-person "dave" empty))
(define eddie
 (make-person "eddie" (list dave)))

(define fred
 (make-person
  "fred"
  (list chuck eddie)))
```



Let's look at the data structure



(person-children p) is always a smaller structure than **p**.



The same thing works for NonNegInts

```
;; sum :
;;   NonNegInt NonNegInt -> NonNegInt
(define (sum x y)
 (cond
   [(zero? x) y]
   [else (+ 1 (sum (- x 1) y))]))
```

Example

```
(sum 3 2)
= (+ 1 (sum 2 2))
= (+ 1 (+ 1 (sum 1 2)))
= (+ 1 (+ 1 (+ 1 (sum 0 2))))
= (+ 1 (+ 1 (+ 1 2)))
= 5
```

At every recursive call, the value of the first argument decreases, so eventually it reaches 0.

Fibonacci

```
;; fib : NonNegInt -> NonNegInt
;; GIVEN: a non-negative integer n
;; RETURNS: the n-th fibonacci number
;; EXAMPLES:
;; fib(0) = 1, fib(1) = 1, fib(2) = 2, fib(3) = 3,
;; fib(4) = 5, fib(5) = 8, fib(6) = 13
;; STRATEGY: Recur on n-1 and n-2
(define (fib n)
  (cond
   [(= n 0) 1]
   [(= n 1) 1]
   [else (+ (fib (- n 1))
            (fib (- n 2)))]))
```

At every recursive call, the value of **n** decreases, so eventually it reaches 0 or 1.

Bad Fibonacci

```
;; bad-fib : NonNegInt -> NonNegInt
```

```
;; STRATEGY: Recur on n-1 and n-2
(define (bad-fib n)
  (cond
   [(= n 0) 1]

   [else (+ (bad-fib (- n 1))
            (bad-fib (- n 2)))]))
```

At every recursive call, the value of **n** decreases.

But wait! When n = 1, this gets into an infinite loop.

What went wrong?

```
(bad-fib 1)
= (bad-fib -1)
= (bad-fib -3)
= ...
```

The contract says the argument to bad-fib is supposed to be a **NonNegInt**.

When n=1, the call **(bad-fib (- n 2))** VIOLATES THE CONTRACT, so all bets are off.

It's really important to make sure that your recursive calls don't violate the function's contract. We'll see much more about this in Module 07.

Summary

- At the end of this lesson you should be able to
  - list the properties that a data type with multiple constructors must have.
  - illustrate how a recursive function that follows the observer template always calls itself on smaller and smaller structures.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board