

Multi-way Trees

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 5.2



© 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

- We've talked about binary trees
- Sometimes, we need to construct trees in which each node has an unbounded number of sons. We call these *multi-way trees*.
 - example: a file system, in which a directory can have any number of files or directories in it.
 - an XML item.
 - a JSON object is a multi-way tree (with additional structure, represented as a string).
 - in this lesson, we'll do a case study of one application of multi-way trees.

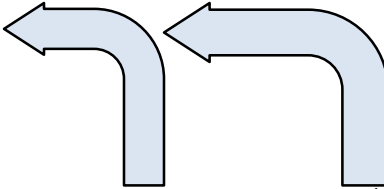
Learning Objectives

- At the end of this lesson, the student should be able to:
 - recognize situations in which a structure may have a component that is a list of similar structures
 - write a data definition for such values
 - write a template for such a structure
 - write functions on such structures

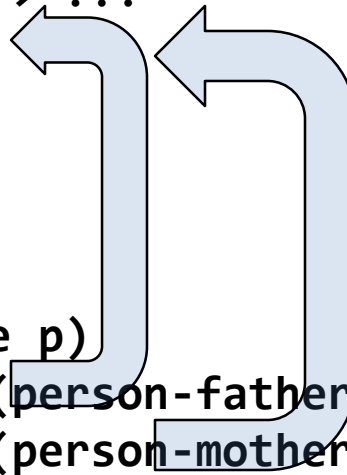
Ancestor Trees

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

```
;; A Person is either  
;; --"Adam"  
;; --"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)))]))
```



Here are ancestor trees, an application of binary trees, which we saw before. For this representation, we needed to introduce "adam" and "eve" as artificial "first people".

A Different Representation: Descendant Trees

```
;; A Person is represented as a struct
;; (make-person name children)
```

```
;; INTERPRETATION
```

```
;; name   : String (any string will do)  --the name of the person
;; children : PersonList                 --the children of the
;;                                           person
```

```
;; IMPLEMENTATION:
```

```
(define-struct person (name children))
```

```
;; CONSTRUCTOR TEMPLATES:
```

```
;; For Person:
```

```
;; (make-person String PersonList)
```

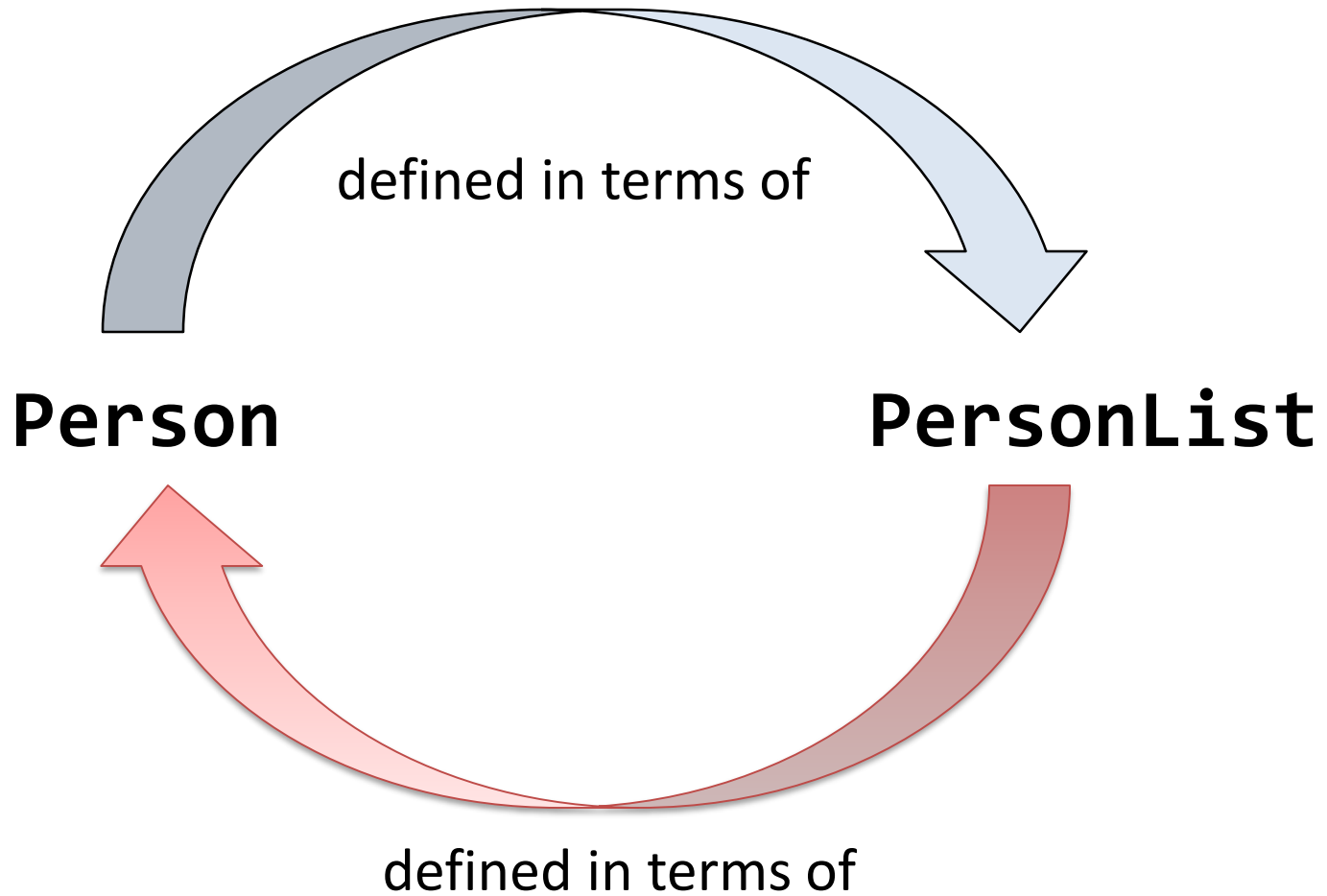
```
;; For PersonList:
```

```
;; empty
```

```
;; (cons Person PersonList)
```

Here is a different information analysis: instead of keeping track of each person's parents, let's keep track of each person's children. A person may have any number of children, including no children. So we can represent each person's children as a list of persons. So now we have a pair of **mutually-recursive** data definitions: **Person** and **PersonList**

This is mutual recursion



Template: functions come in pairs

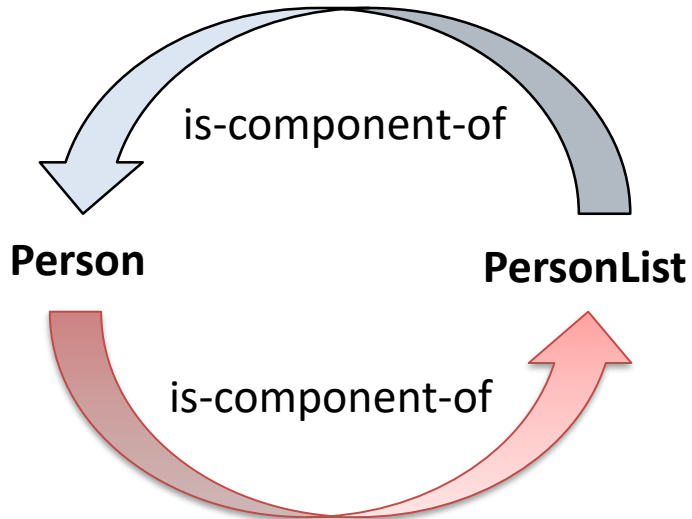
```
;; person-fn : Person -> ??  
(define (person-fn p)  
  (... (person-name p)  
        (plist-fn (person-children p))))
```

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

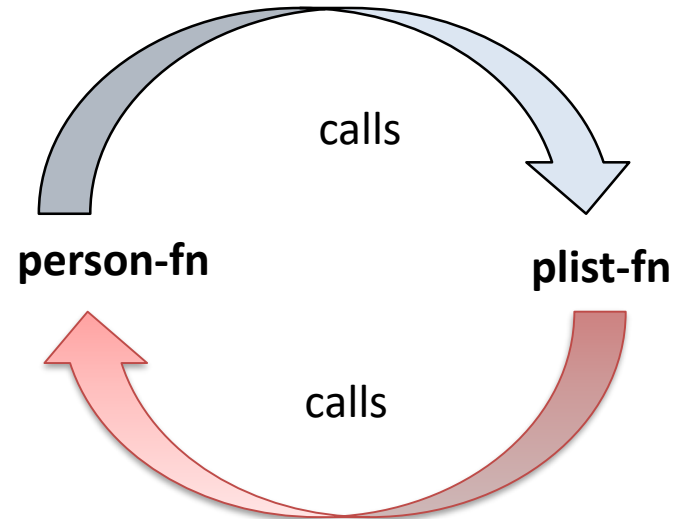
Here is the pair of templates that we get by applying the recipe to our data definition.

They are mutually recursive, as you might expect.

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



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**.

The template questions

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

Given the answer for a person's children, how do we find the answer for the person?

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

What's the answer for the empty PersonList?

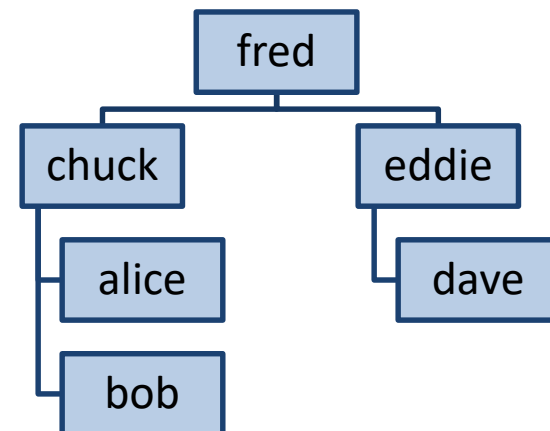
Given the answer for the first person in the list and the answer for the rest of the people in the list, how do we find the answer for the whole list?

Examples

```
(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)))
```



Vocabulary

- A tree where each node contains a list of subtrees is called a *multi-way tree*, a *general tree*, or sometimes a *rose tree*.
- Observe that the "base case" is a tree containing an empty list of subtrees.

Grandchildren

Here's a simple function we might want to write.

```
;; grandchildren : Person -> PersonList
;; GIVEN: a Person
;; RETURNS: a list of the grandchildren of the given
;; person.
;; EXAMPLE: (grandchildren fred) = (list alice bob dave)
;; STRATEGY: Use template for Person on p
(define (grandchildren p)
  (... (person-children p)))
```

Q: Given p's children, how do we find p's grandchildren?

A: We need a function which, given a list of persons, produces a list of all their children

all-children

```
;; all-children : PersonList -> PersonList
;; GIVEN: a list of persons
;; RETURNS: a list of all their children.
;; (all-children (list fred eddie))
;; = (list chuck eddie dave)
(define (all-children ps)
  (cond
    [(empty? ps) empty]
    [else (append
            (person-children (first ps))
            (all-children (rest ps))))]))
```

This one was too easy!
It didn't require mutual
recursion.

Putting it together

```
;; grandchildren : Person -> PersonList
;; STRATEGY: Use template for Person on p
(define (grandchildren p)
  (all-children (person-children p)))

;; all-children : PersonList -> PersonList
;; STRATEGY: Use template for PersonList on ps
(define (all-children ps)
  (cond
    [(empty? ps) empty]
    [else (append
            (person-children (first ps))
            (all-children (rest ps))))]))
```

Here's a slightly harder task.

descendants

- Given a person, find all his/her descendants.
- What's a descendant?
 - a person's children are his/her descendants.
 - any descendant of any of a person's children is also that person's descendant.
- Hey: this definition is recursive!

Contracts and Purpose Statements

```
;; person-descendants : Person -> PersonList  
;; GIVEN: a Person  
;; RETURNS: the list of his/her descendants
```

```
;; all-descendants : PersonList -> PersonList  
;; GIVEN: a PersonList  
;; RETURNS: the list of all their descendants
```

Here are the contracts and purpose statements.

The task description talked about "all the descendants of a person's children". A person's children are a list of persons, so that gives us a clue that we will need the function we've called **all-descendants** here.

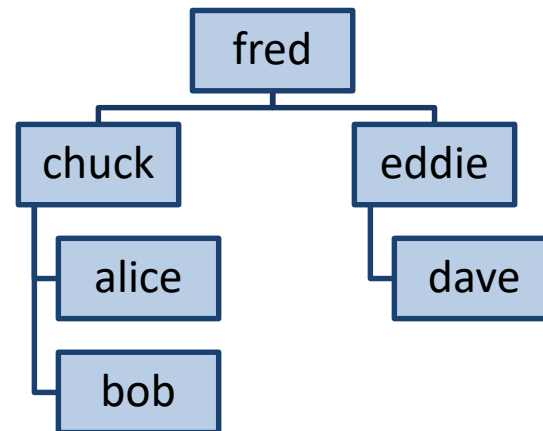
Examples

`(person-descendants fred)`

`= (list chuck eddie alice bob dave)`

`(all-descendants (list chuck eddie))`

`= (list alice bob dave)`



The template questions

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

Given the answer for a person's children, how do we find the answer for the person?

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

What's the answer for the empty PersonList?

Given the answer for the first person in the list and the answer for the rest of the people in the list, how do we find the answer for the whole list?

Function Definitions

```
;; Person -> PersonList
;; STRATEGY: Use template for Person on p
(define (person-descendants p)
  (append
   (person-children p)
   (all-descendants (person-children p))))
```

We fill in the blanks in the template with the answers to the template questions.

```
;; PersonList -> PersonList
;; STRATEGY: Use template for PersonList on ps
(define (all-descendants ps)
  (cond
   [(empty? ps) empty]
   [else (append
           (person-descendants (first ps))
           (all-descendants (rest ps))))]))
```

The answers come right from the definition!

Tests

```
(check-equal?  
  (person-descendants fred)  
  (list chuck eddie alice bob dave))
```

```
(check-equal?  
  (all-descendants (list chuck eddie))  
  (list alice bob dave))
```

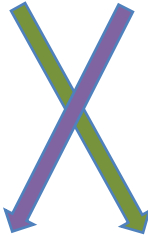
Are these good tests?

- Could a program fail these tests but still be correct? If so, how?
- Answer: Yes! It could produce the list of descendants in a different order, or with duplications.

Better Tests

```
(check same-people?  
 (person-descendants fred)  
 (list chuck eddie alice bob dave))
```

```
(check same-people?  
 (person-descendants fred)  
 (list chuck eddie alice dave bob))
```



```
(check same-people?  
 (all-descendants (list chuck eddie))  
 (list alice bob dave))
```

There are two ways we could solve this problem:

1. We could have our purpose statement specify the order in which the descendants are to be listed.
2. We could use smarter tests that would accept the answer list in any order.

Here we've adopted the second approach. Instead of **check-equal?**, we use **check**, which takes as its first argument a predicate to be used to compare the actual and expected answers. We'll have to define a function **same-people?**. We've done this in the example file for this lesson. And of course we have to have tests for **same-people?**; otherwise we wouldn't have any reason to believe the results of the tests that rely on it.

Here are some tests for **(person-descendants fred)** that list the answer in two different orders.

Summary

- You should now be able to:
 - recognize situations in which a structure may have a component that is a list of similar structures
 - write a data definition for such values
 - write a template for such a structure
 - write functions on such structures

Next Steps

- Study the file 05-2-descendants.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 5.2