Multi-way Trees

CS 5010 Program Design Paradigms

Lesson 6.5

© Mitchell Wand, 2012-2014
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Introduction

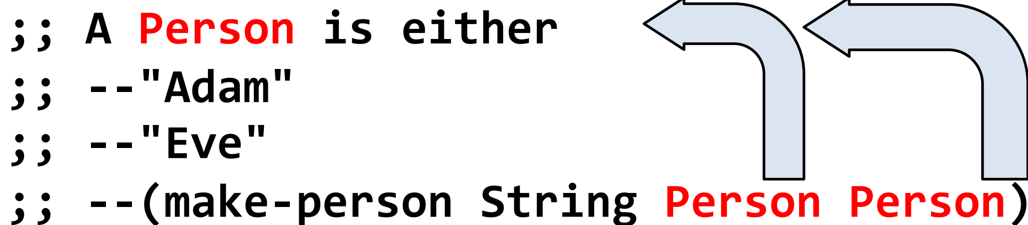
- We've talked about binary trees
- Sometimes, we need to construct trees in which each node has an unbounded number of sons. We call these *multi-way trees*.
  - example: a file system, in which a directory can have any number of files or directories in it.
  - example: S-expressions, in which a LoSS may contain any number of strings or SoS's.
  - an XML item.
  - in this lesson, we'll do a case study of one application of multi-way trees.

Learning Objectives

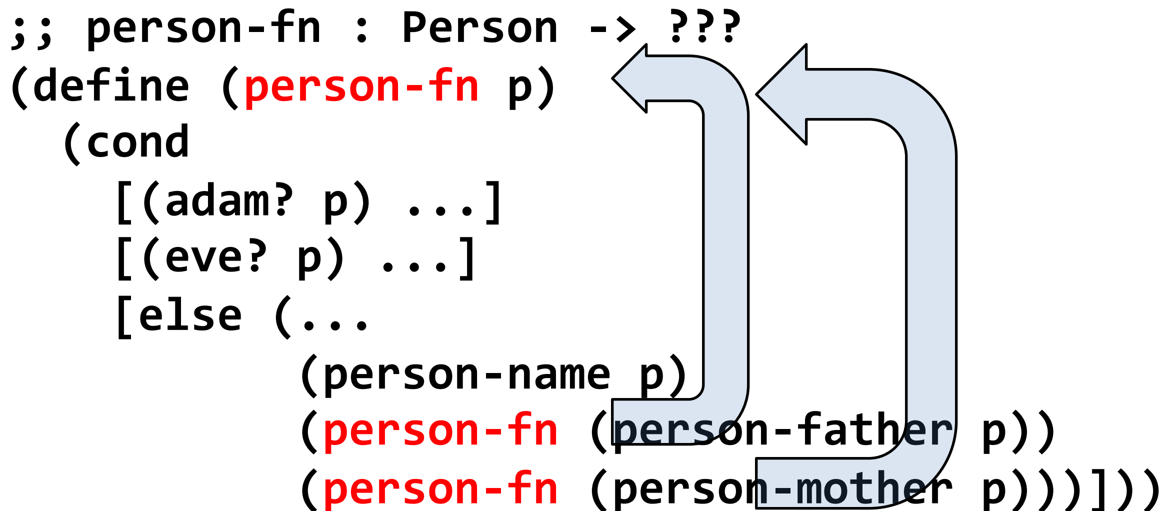
- At the end of this lesson, the student should be able to:
  - recognize situations in which a structure may have a component that is a list of similar structures
  - write a data definition for such values
  - write a template for such a structure
  - write functions on such structures

Ancestor Trees

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

;; A Person is either
;; --"Adam"
;; --"Eve"
;; --(make-person String Person Person)
```



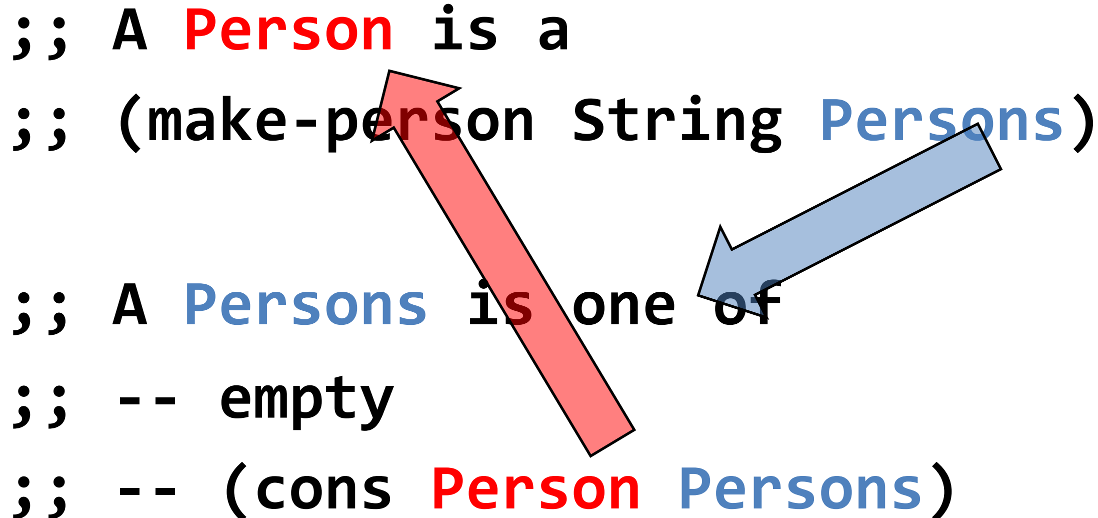
```
;; person-fn : Person -> ???
(define (person-fn p)
  (cond
    [(adam? p) ...]
    [(eve? p) ...]
    [else (...
           (person-name p)
           (person-fn (person-father p))
           (person-fn (person-mother p)))]))
```



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 Info Analysis: Descendant Trees

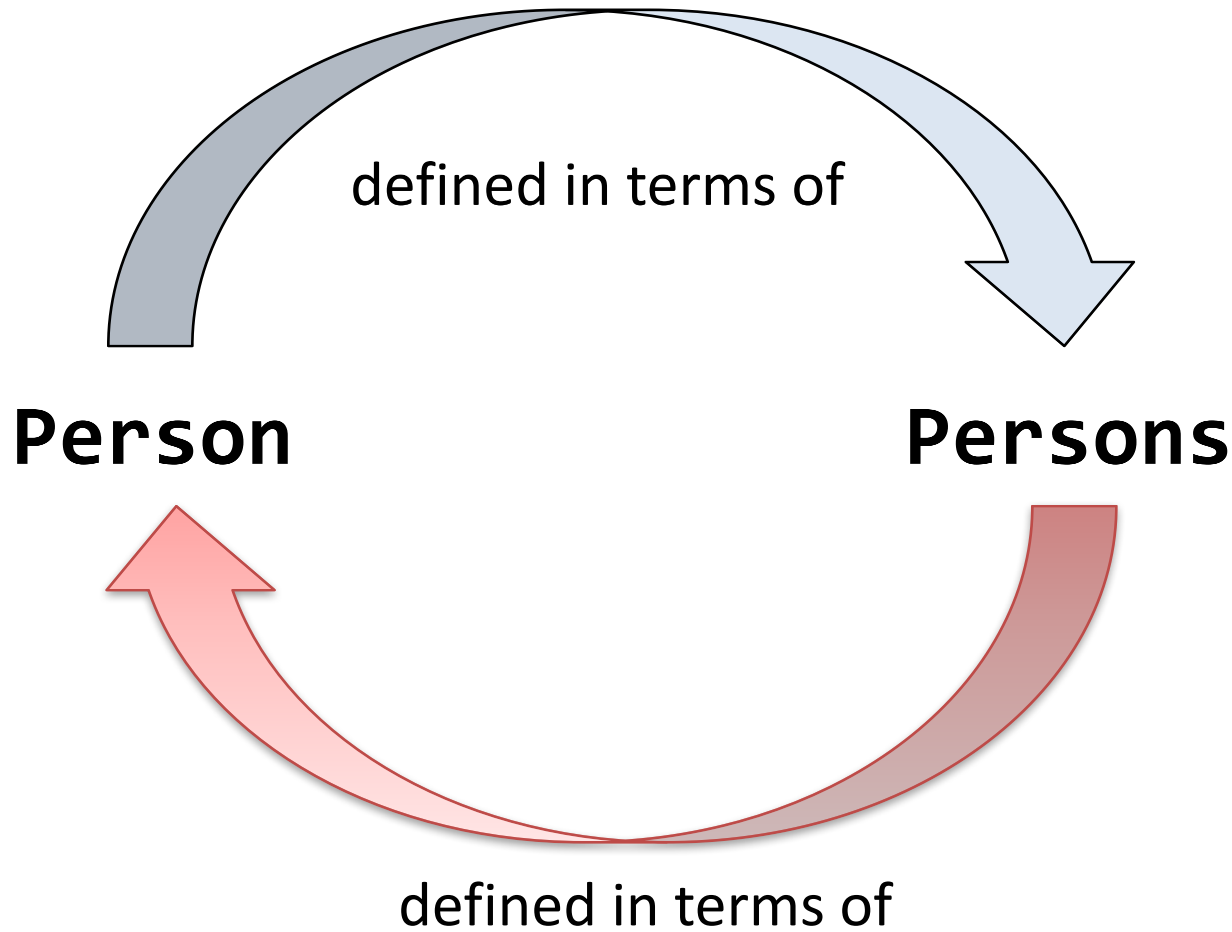
```
(define-struct person (name children))

;; A Person is a
;; (make-person String Persons)

;; A Persons is one of
;; -- empty
;; -- (cons Person Persons)
```



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.

Two *mutually recursive* data definitions

This is mutual recursion



The template recipe

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.

Here is the recipe for templates again. Let's apply it to our Person trees.

Template: functions come in pairs

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

;; persons-fn : Persons -> ??
(define (persons-fn ps)
  (cond
    [(empty? ps) ...]
    [else (... (person-fn (first ps))
               (persons-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.

The template questions

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

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

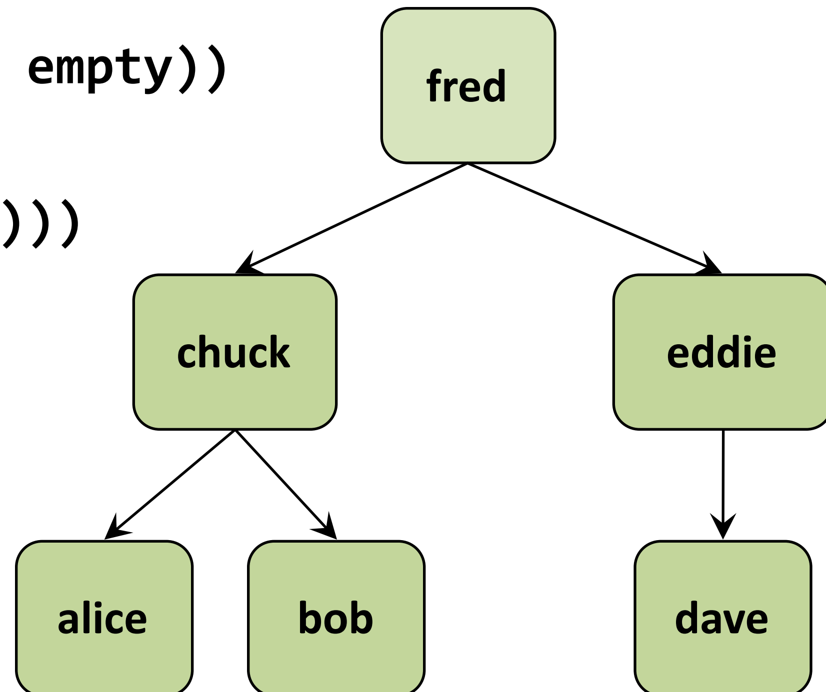
```
;; persons-fn : Persons -> ??
(define (persons-fn ps)
  (cond
    [(empty? ps) ...]
    [else (... (person-fn (first ps))
               (persons-fn (rest ps)))]))
```

What's the answer for the empty Persons?

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*, or 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 -> Persons
;; 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

persons-all-children

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

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

Putting it together

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

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

We could use HOFs, too

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

;; persons-all-children : Persons -> Persons
;; STRATEGY: Use HOF map on ps
(define (persons-all-children ps)
  (foldr append empty
    (map person-children ps)))
```

Of course, a **Persons** is a list, so we can use our list abstractions to define **persons-all-children**. This will often be the case.

descendants

Here's a slightly harder task.

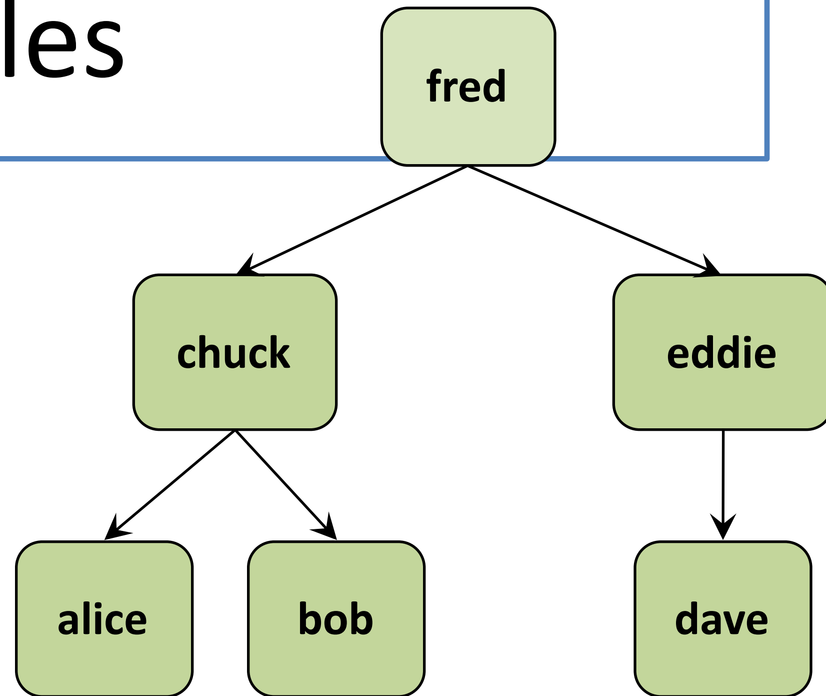
- Given a person, find all his/her descendants.
- What’s a descendant?
  - a person’s children are his/her descendants.
  - any descendant of any of a person’s children is also that person’s descendant.
- Hey: this definition is recursive!

Contracts and Purpose Statements

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

;; persons-descendants : Persons -> Persons
;; GIVEN: a Persons
;; 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 **persons-descendants** here.

Examples




```
(person-descendants fred)
= (list chuck eddie alice bob dave)

(persons-descendants (list chuck eddie))
= (list alice bob dave)
```

The template questions

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

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

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

What's the answer for the empty Persons?

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 -> Persons
;; STRATEGY: Use template for Person on p
(define (person-descendants p)
  (append
   (person-children p)
   (persons-descendants (person-children p))))

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

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

The answers come right from the definition!

Or, with the HOFs

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

;; Persons -> Persons
;; STRATEGY: Use HOF map followed by foldr
(define (persons-descendants ps)
  (foldr append empty
    (map person-descendants ps)))
```

As we did before, we could replace the structural decomposition on Persons with Higher-Order Function Composition. The functions are still mutually recursive.

Tests

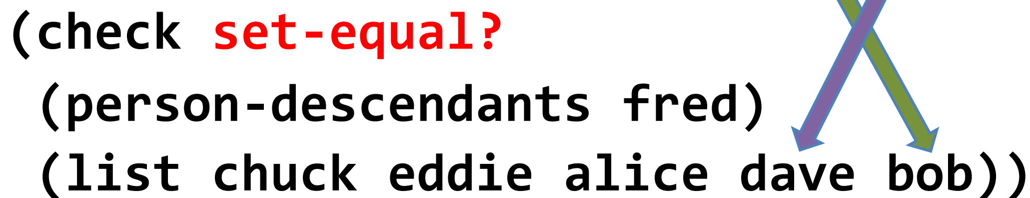
```
(check-equal?
 (person-descendants fred)
 (list chuck eddie alice bob dave))

(check-equal?
 (persons-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.

Better Tests

```
(require "sets.rkt")   ;; or whatever...

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



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

(check set-equal?
 (persons-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 require a library that provides **set-equal?**-- the file **sets.rkt**, which we worked with last week, will do nicely. We've put a working copy of **sets.rkt** in the Examples file for this week.

Here are some tests for **(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 06-5-descendants.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.5
- Do the problem set