Recursive Data 2

Mutually Recursive Data Definitions

(HTDP sec 15.1)
Previously: Family Trees

- Specifically, *ancestor family trees*

  A family-tree-node is either
  - empty, or
  - (make-child fa mo da na ey)
    where na and ey are symbols
    and da is a number
    and fa and mo are family-tree-nodes

- A self-referential data type of our own invention
A function on ancestor family tree nodes

```
;; blue-eyed-ancestor? : ftn  ->  boolean

(define (blue-eyed-ancestor? a-ftree)
  (cond
   [(empty? a-ftree) false]
   [(symbol=? (child-eyes a-ftree) 'blue) true]
   [else (or
          (blue-eyed-ancestor? (child-father a-ftree))
          (blue-eyed-ancestor? (child-mother a-ftree))))]))
```

- The colored portions come directly from the template for ancestor family trees
A new data type: descendant family trees

- Like ancestor f.t.’s, with one key difference
  - each node now knows about its children, instead of its parents

- Ancestor trees were easy to represent
  - You can have at most two parents!

- Descendant trees will be harder
  - How do you encapsulate potentially many children in a structure?
Lists inside structures

- Sure, why not? Let’s write the data definition for a node:
  - (we’ll call it “parent” since each node may have potentially many children)

; a parent is (make-parent loc n d e)
; where n, e are symbols
; and d is a number
; and loc is a list of children

- Now we have a problem. What’s a “list of children”??
Lists inside structures (take 2)

- Let’s try again, starting with the data definition for a list of children:

```
; a list of children is either
;   - empty, or
;   - (cons p loc)
;     where p is a parent
;     and loc is a list of children
```

- We’re still stuck. Now we know what a list of children is, but “parent” is undefined.
Mutually Referential Data Definitions

- The point is, you need *both* parts of the data definition for it to be complete and legal.

```scheme
; a parent is (make-parent loc n d e)
; where n, e are symbols
; and d is a number
; and loc is a list of children
;
; a list of children is either
; - empty, or
; - (cons p loc)
; where p is a parent
; and loc is a list of children
```
Examples

(define-struct parent (children name date eyes))

(define Violet
  (make-parent empty 'VioletParr 1990 'brown))
(define Dash
  (make-parent empty 'DashiellParr 1995 'blue))
(define JackJack
  (make-parent empty 'JackParr 2002 'blue))

(define Elastigirl
  (make-parent
    (list Violet Dash JackJack) 'HelenParr 1962 'brown))
(define MrIncredible
  (make-parent
    (list Violet Dash JackJack) 'BobParr 1958 'blue))
The template should match the data definition
- Because the d.d. has two parts, so must the template

; template for functions on descendant tree nodes
; dtn-func : parent -> ???
(define (dtn-func p)
  ...
  (loc-func (parent-children p))
  ...
  (parent-name p)
  ...
  (parent-date p)
  ...
  (parent-eyes p) ...
)

; template for functions on lists of children
; loc-func : list of children -> ???
(define (loc-func loc)
  (cond
    [(empty? loc) ... ]
    [else ...
      (dtn-func (first loc)) ...
      (loc-func (rest loc)) ...
    ]))

(Does the second one look familiar? It should—it’s just the template for lists, with an extra recursive call.)
Example function: blue-eyed-descendant?

Unlike blue-eyed-ancestor?, blue-eyed-descendant? must follow this two-part template.
  (once again, colored portions come from the template)

;;; blue-eyed-descendant? : parent  ->  boolean
;;; to determine whether a-parent any of the descendants (children, ;;; grandchildren, and so on) have 'blue in the eyes field
(define (blue-eyed-descendant? a-parent)
  (cond
   [(symbol=? (parent-eyes a-parent) 'blue) true]
   [else (blue-eyed-children? (parent-children a-parent))]))

;;; blue-eyed-children? : list-of-children  ->  boolean
;;; to determine whether any of the structures in aloc is blue-eyed
;;; or has any blue-eyed descendant
(define (blue-eyed-children? aloc)
  (cond
   [(empty? aloc) false]
   [(blue-eyed-descendant? (first aloc)) true]
   [else (blue-eyed-children? (rest aloc))])))