Mutually Referential Data Definitions

Corky Cartwright
Department of Computer Science
Rice University

Announcements and Plan

- Homework due Thursday at 11am.
  - Why? Monday is a holiday, so we cannot hold labs on Monday.
  - Monday labs moved to Wednesday; same time and place.
- Plan for today
  - What is mutual referencing/recursion
  - Simple and deep examples illustrating the approach.

Example of a Mutually Referential Data Definition

```
; Descendant trees
; A parent is a structure
;     (make-parent loc n)
; where loc is a list-of-children,
;       n is a symbol

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

Terminology and Template

- Common terminology: mutually **recursive** instead of mutually **referential**
- Writing **one** function on any of these types requires writing a set of functions for **all** the mutually recursive types
- Each reference to a mutually recursive type in a **definition** corresponds to a different recursive call to the appropriate function **in** the corresponding **template**

Template(s)

```
; A parent is a structure
;      (make-parent n loc)
; where n is a symbol (the name of the
  parent) and loc is a list-of-children,
(define-struct parent (name children))

; parent-fn: parent -> ...
; (define (fun-parent ... p ...)
;    (... (parent-name p) ...
      ... (loc-fn (parent-children p)) ...))
```

Templates, cont.

```
; A list-of-children is either
;     empty, or
;     (cons p loc) where
; where p is a parent and loc is a list-of-children
; fun-loc: list-of-children -> ...
; (define (loc-fn ... loc ...)
;  (cond [(empty? loc) ...]
;        [else
;          ... (parent-fn ... (first loc) ...) ...
;          ... (loc-fn ... (rest loc) ...) ...)]))
```

Function calls in template(s)

- Mutually recursive calls are part of template
  - Use of a mutually recursive type is just the same as a recursive use of a type itself
  - A set of mutually recursive type definitions is really one big recursive type definition with multiple parts and each part has a template
- The form of the function calls in the template(s) is crucial for ensuring termination

More about termination

- For the inductive (self-referential) types we saw before today, a recursive functions terminates if
  - it handles the base case(s) cleanly, and
  - ir only make recursive calls on substructures of its primary argument, *e.g.*, the `rest` of a non-empty list
- Mutually recursive (referential) definitions are the same
  - Example: Imagine a type box that can contain bags, and a type bag that can contain boxes. Why does the template ensure termination?
    - Any box will be bigger than any bag it contains
    - Similarly for bags.
    - No infinite descending chains of containment.

Code

- Write a function that counts the people in a descendant tree

```
; parent-count : parent -> natural
; children-count : list-of-children -> natural
(define (parent-count p)
  (add1 (children-count (parent-children p))))
(define (children-count aloc)
  (cond [(empty? aloc) 0]
        [else (+ (parent-count (first aloc))
                 (children-count (rest aloc)))]))

; Note:  Mutual "defines" should be contiguous
```

Another Example (Unix File System)

```
; A file is either:
;   a rawFile, or
;   a dir (short for directory)

; A dir is a structure
; (make-dir lonf) where lonf is a list-of-nFiles
(define-struct dir (nFiles))

; A list-of-nFiles is either:
; ...

; A nFile is a structure
; (make-nFile name f) where name is a symbol and f is a
  file.
(define-struct nFile (name file))
```

Template

```
; A file is either:
;   a list-of-char, or
;   a dir

; file-fun : f -> ...
(define (file-fun ... f ...)
  (cond [(rawFile? f) ...]
        [(dir? f) ...
         ... (dir-fun ... f ...)) ... ]))

; A dir is
;(make-dir lonf) where lonf is a list-of-nFiles

; dir-fun : dir -> ...
(define (dir-fun ... d ...)
   ... (nFiles-fun ... (dir-nFiles d) ...) ... )
```

Template cont.

```
; A list-of-nFiles is either:
; ...
(define (lonf-fun ... lonf ... )
  (cond [(empty? lonf) ... ]
        [(cons? lonf) ...
         ... (nFile-fun ... (first lonf) ... ) ... )
         ... (lonf-fun ... (rest lonf) ...) ... ]))

; A nFile is a structure
; (make-nFile name f) where name is a symbol and f is a file.
(define (nFile-fun ... nf ...)
  ... (nFile-name nf) ...
  ... (file-fun ... (nFile-file nf) ... ) ... )
```

Example function on file system

```
; find?: dir symbol -> boolean
; Purpose: (find? d n) determines whether a file with name an
   occurs in directory d.
; Instantiated template
#|
(define (find? d n ) ... (nFiles-find? (dir-nFiles d) n) ... )

(define (nFiles-find? lonf n)
  (cond [(empty? lonf) ...]
        [(cons? lonf)
         ... (nFile-find? (first lonf) n) ...
         ... (nFiles-find? (rest lonf) n) ... ]))

(define (nFile-find? nf n)
  ... (nFile-name nf) ...
  ... (file-find? (nFile-file nf) n) ... )
```

Example function cont.

```
(define (find? d n) ... (nFiles-find? (dir-nFiles d) n) ... )

(define (nFiles-find? lonf n)
  (cond [(empty? lonf) ...]
        [(cons? lonf)
         ... (nFile-find? (first lonf) n) ...
         ... (nFiles-find? (rest lonf) n) ... ]))

(define (nFile-find? nf n)
  ... (nFile-name nf) ...
  ... (nFile-find? (nFile-file nf) n) ... )

(define (file-find? f n)
  (cond [(rawFile? f) ... ]
        [(dir? f) ... (find? f n) ... ]))
|#
```

Code

```
(define (find? d n) (nFiles-find? (dir-nFiles d) n))

(define (nFiles-find? lonf n)
  (cond [(empty? lonf) false]
        [(cons? lonf)
         (or (nFile-find? (first lonf) n)
             (nFiles-find? (rest lonf) n))]))

(define (nFile-find? nf n)
  (or (equal? (nFile-name nf) n)
      (file-find? (nFile-file nf) n)))

(define (file-find? f n)
  (cond [(rawFile? f) false]
        [(dir? f) (find? f n)]))
|#
```

For Next Class

- Homework deferred one day
- Labs:
  - Don't forget Tuesday lab
  - Monday labs meet Wednesday