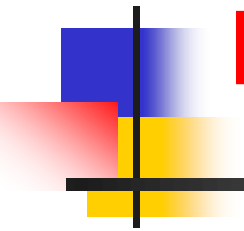
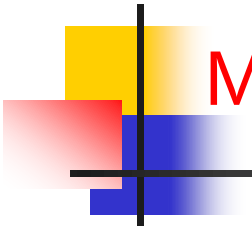


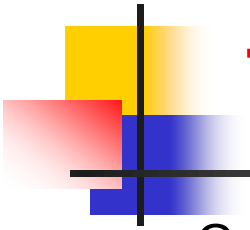
Mutually Referential Data Definitions and Help Functions

Corky Cartwright
Department of Computer Science
Rice University



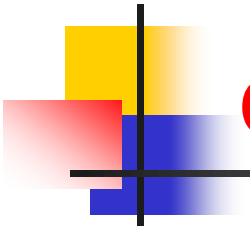
Mutually Referential Data Definitions

- Real world data tends to have more variety (diversity?) than simple lists or binary trees.
- My favorite example: program expressions, often called *abstract syntax*.
- Critical insight in defining program data; it has far more structure than what normal input/output media support, namely sequences of characters, arrays of pixels.
- Applications typically need to build rich hierarchical or linked representations. Circular linking (general graphs) is messy; trees or dags (directed-acyclic graphs) have a simple inductive structure



Terminology

- Common terminology: mutually *recursive* instead of mutually *referential*.
- Which is better? I prefer *recursive* because it suggests repeating structure which is the normal and attractive form of diversity typically encountered in computation. Random interconnections are difficult to process.
- Key insight: writing **one** function over a recursively interconnected collection of types requires writing a collection of functions, one for each form of data in the web of mutually recursive types. Many different forms of data (constructors) and best handled by writing a separate function for each kind.
- Each reference to a given mutually recursive type in a data domain *definition* corresponds to a different recursive call to the appropriate function in the corresponding *template*.
- Sound OO? There is a deep connection between the OO perspective and the functional one.



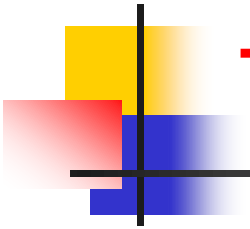
Canonical Example: Abstract Syntax

A Misleading Example (which is typical of introductions to abstract syntax):

```
; An expression is one of:
; - a number
; - a symbol
; - (make-mul e1 e2) where e1 and e2 are expressions
; - (make-add e1 e2) where e1 and e2 are expressions
; - (make-div e1 e2) where e1 and e2 are expressions
; - (make-sub e1 e2) where e1 and e2 are expressions
; given

(define-struct mul (left right))
(define-struct add (left right))
(define-struct div (left right))
(define-struct sub (left right))

; Examples
;  5
;  'f
;  (make-mul 5 3)
;  (make-add 5 3)
;  (make-div 5 3)
;  (make-sub 5 3)
```



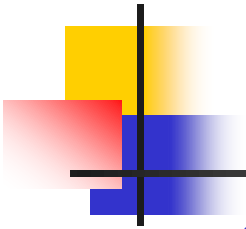
Templates.

```
; Template for processing such an expression
#|
(define (f ... exp ...)
  (cond
    [(number? exp) ... ]
    [(symbol? exp) ... ]
    [(mul? exp) ... (f ... (mul-left exp) ...) ... (f ... (mul-right exp) ...) ... ]
    [(add? exp) ... (f ... (add-left exp) ...) ... (f ... (add-right exp) ...) ... ]
    [(div? exp) ... (f ... (div-left exp) ...) ... (f ... (div-right exp) ...) ... ]
    [(sub? exp) ... (f ... (sub-left exp) ...) ... (f ... (sub-right exp) ...) ... ]))
```

This template is rather large (six cases) and ugly; the only saving grace is that the four operations (+, *, -, /) all have very similar form: they all process pairs of numbers. These four operations are often chosen as a starting point because they are familiar infix algebraic operations that we all regularly use and understand.

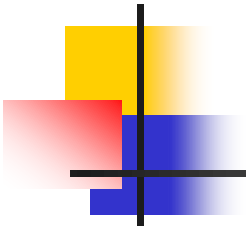
But do we want to design our framework for expression processing based on this limited form of data? What about binding local variables and retrieving their values. What about defining new functions. What about passing functions as arguments? Can we accommodate recursion in the definition of program functions? Suddenly the simple case splitting model suggested by the preceding template looks much too rigid and narrow. We need a framework for handling many different linguistic constructions. The sublanguage with which we started only contains applications of primitive functions to numbers.

We will look at much richer data definitions of abstract syntax later in the course.



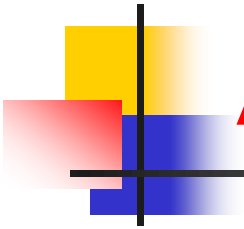
Function calls in templates

- Mutually recursive calls are part of each 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
- To ensure termination, the structure of the function calls in the template(s) is crucial for ensuring termination; each recursive call should reduce “a measure of the arguments” with values in a well-founded set.
- Not always possible; the desired behavior may include divergence. Can you think of a real world sequential program consuming a finite input that does not always terminate?



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.
  - Aside: what is a bag?



Another Example (Unix File System)

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

; A rawFile is (make-rawFile text) where text is a
; a char-list

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

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

Mutually Recursive Templates

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

; dir-f : dir -> ...
(define (dir-f ... d ...)
   ... (nFiles-f ... (dir-nFiles d) ...) ... )

; nFiles-f: nFile-list -> ...
(define (nFiles-f ... nFiles ... ) ;; nFiles is nFile-list
  (cond [(empty? nFiles) ... ]
        [(cons? nFiles) ...
         ... (file-f ... (nFile-file (first nFiles)) ... ) ... )
         ... (nFiles-f... (rest nFiles) ...) ... ]
```

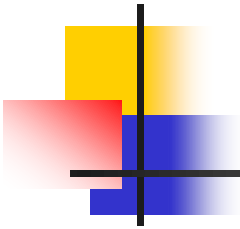
Example function on file system

```
; find?: file symbol -> boolean
; Purpose: (find? f n) determines whether a file (which must
; be a directory for this query to be interesting) contains
; file with the name n.

; Instantiated template
#|
(define (find? f n )
  (cond [(rawFile? f) false]
        [(dir? f) ... (nFiles-find? (dir-nFiles f) n) ... )

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

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



Code

```
(define (find? f n)
  (cond [(rawFile? f) false]
        [(dir? f) (nFiles-find? (dir-nFiles d) n)]))

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

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

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