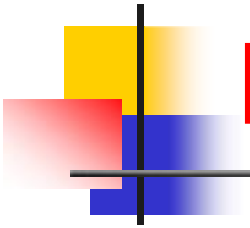
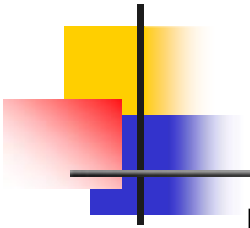
Comp 311
Accumulators, Help Functions and Mutually Referential Data Definitions

Corky Cartwright
Department of Computer Science
Rice University



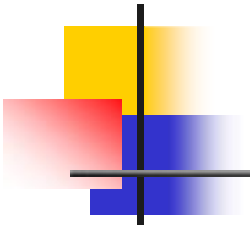
Review

- Data Definition for Deep Lists
- Recursion on Deep Lists
- Example: flatten
- Algebraic Data Types
- Simple Example of an Algebraic Data Type: $\mathbf{N}$ (Natural Numbers)
- More examples: Binary Trees in various forms including BSTs
  There is a beautiful online purely functional presentation of red-black trees (without deletions) online.



Accumulators and Help Functions

- Fast algorithms often use “temporaries” and auxiliary data structures
- Common examples abound. Some appear on HW2. An example that integrates well with the narrative in these lectures: a more efficient version of **`flatten`**.
- **Key idea**: introduce help functions with extra arguments that are bound to the requisite *temporary* data values that are updated (in recursive calls) as the computation proceeds.
- **Key restriction**: the temporaries and auxiliary data values must be updateable without mutation! The new values are constructed from existing values without modifying the latter.
- **Connection with imperative loops:** the variables that are mutated in each loop iteration correspond to accumulators holding temporary values. You can literally translate a loop to a help function using this correspondence. Observation: help functions with accumulators are more general than conventional loops!



Intuitions Guiding The Design of Help Functions

- To avoid using **`append`**, we must construct results from right-to-left rather than left-to-right and we must carry all of the values that are updated on each call (iteration in an imperative loop) as accumulators.
- Consequence:
  - The base case is an expression that constructs the result (typically using a few constant-time operations) from the accumulator values.
- To avoid stack overhead, all calls on the parent help function must be in *tail position.* If an expression appears in tail position, no more operations need to be performed to return the answer from the parent function eliminating the need to even return to the parent function (which would simply return to its caller). The optimized code returns immediately to the parent's caller (unless it also performed tail call optimization).

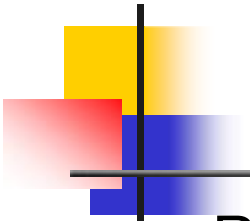
Familiar Example: **`flatten`**

- Our naïve formulation of **`flatten`** relies on **`append`** to concatenate sections of the result list. This process (which is applied recursively!) implies that the running time of **`flatten`** is quadratic. If we wrote imperative code we could clearly perform this computation in linear time since we can concatenate lists in linear time if we use a linked representation and maintain list headers with pointers to the last elements.

- **Observation:** our naïve functional formulation constructs the flattening of **`(first dl)`** by using **`cons`** to build a list of symbols that is initially empty. We could use the flattening of **`(rest dl)`** instead of **`empty`** as the seed value for flattening **`(first dl)`**. But this revision requires a help function that takes the accumulated list (the flattening of **`(rest dl)`**) as an extra argument. Let's write it. Note that our recursive calls must use the help function in order for the optimization to be performed "all the way down" to the leaves of the input deep list **`dl`**.

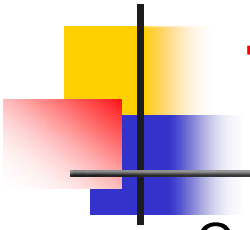
A Faster Version of `flatten`

```
;; flatten-help: deep-list (list-of symbol) -> (list-of symbol)
;; Contract: Given a deep-list dl and a (list-of symbol) accum,
;;   (flatten-help dl accum) returns the flattening of dl appended to accum.
;; Examples ...
;; Template Instantiation (of deep-list Template)
#|
(define (flatten-help dl accum)
  (cond [(empty? dl) ... ]
        [(cons? d1) ;; else could be used here
         (cond [(symbol? (first dl)) ... (first dl) ... (flatten-help (rest dl) accum) ...)]
               [(empty? (first dl)) ... (flatten-help (rest dl) accum) ... ]
               [(cons? (first dl)) ...
                (flatten-help (first dl) (flatten-help (rest dl) accum) ... ])]))
|#
;; Code:
(define (flatten-help dl accum)
 (cond [(empty? dl) accum]
       [(cons? d1)
        (cond [(symbol? (first dl)) (cons (first dl) (flatten-help (rest dl) accum)))]
              [(empty? (first dl)) (flatten-help (rest dl) accum))]
              [(cons? (first dl))
               (flatten-help (first dl) (flatten-help (rest dl) accum)])])))
;; flatten: deep-list -> (list-of symbol)
;; Contract: ...
;; Template Instantion ... [trivial]
;; Code
(define (flatten dl) (flatten-help dl empty))
```



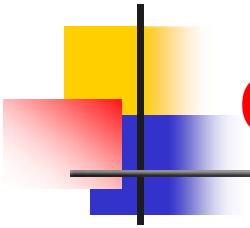
Mutually Referential Data Definitions

- Real world data tends to have more 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, *i.e.*, sequences of characters, arrays of pixels.
- Applications typically need to build rich hierarchical or linked representations. Circular linking (general graphs) is messy but occasionally necessary; trees or DAGs (directed-acyclic graphs) have a simple inductive structure and for this reason are preferred in programming tools.



Terminology

- Common terminology: mutually *recursive* instead of mutually *referential*.
- Which is better? I prefer *recursive* because it suggests repeating hierarchical structure which is the normal and attractive form of diversity typically encountered in computation. Random interconnections are difficult to process.
- **Key insight**: defining **one** operation 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 (echoes of OO design).
- 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.



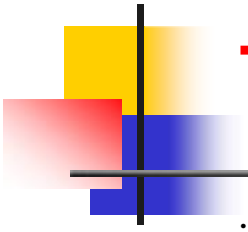
Canonical Example: Abstract Syntax

A Simplified AST 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.

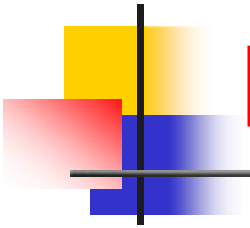
```
; Template for processing such an expression

(define (f ... exp ...)
  (cond ;; six different forms of an AST
    [(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 potentially ugly; the primary 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. For some functions, such as evaluating oridinary arithmetic expressions, this template works well.

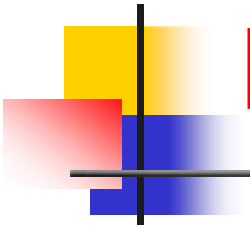
But do we want to design all frameworks for expression processing based on this limited form of data? What about binding local variables and retrieving their values. What about expression languages that support defining new functions. What about expression languages that support passing functions as arguments? Suddenly the simple case splitting model embodied in the preceding template looks much too rigid and narrow. For many problems, we need a more general framework for handling many different linguistic constructions. The sublanguage in this example only contains applications of primitive functions to numbers.

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



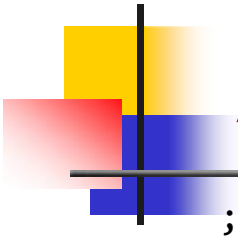
Function calls in templates

- Mutually recursive calls are part of each template
  - Processing a mutually recursive type is very similar to processing a “singly” recursive type.
  - A set of mutually recursive type definitions is really one big recursive type definition with multiple parts where 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 taken from a well-founded set (no infinite descending chains).
- This goal is not always achievable; the desired behavior may include divergence. Can you think of a real world sequential program consuming a finite input that does not always terminate? (Hint: what about interpreting programs in a Turing complete language? Note that such an interpreter does not use ordinary structural recursion on program syntax.)



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
  - it only makes 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 (implemented using **`(define-struct bag (boxes))`** that contains a list of 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.
  - Note: do not confuse the name **`bag`** with the mathematical concept of a *bag* (multi-set).



A Richer 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 (list-of char)
(define-struct rawFile (text))

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

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

In contrast to simple arithmetic expressions, this domain has two different types that are primary. There are important operations that make sense for inputs of type **`dir`** that do not make sense for a **`rawFile`**.

Observation: files should not have names! Windows blew it!

Mutually Recursive Templates

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

;;
```

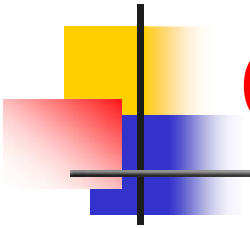
Sample function on file system

```
; find?: file symbol -> boolean
; Contract: (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

```
;; find?: file symbol -> boolean
(define (find? f n)
  (cond [(rawFile? f) false]
        [(dir? f) (nFiles-find? (dir-nFiles d) n)))

;; nFiles-find?: (list-of nFile) symbol -> boolean
(define (nFiles-find? nfl n)
  (cond [(empty? nfl) false]
        [(cons? nfl)  ;; we could have used else instead
         (or (nFile-find? (first nfl) n)
             (nFiles-find? (rest nfl) n)]))

;; nFile-find?: nFile symbol -> boolean
(define (nFile-find? nf n)
  (or (equal? (nFile-name nf) n)
      (find? (nFile-file nf) n))
```

Aside: What is the meaning of **or** ? Does it behave like a program-defined function?