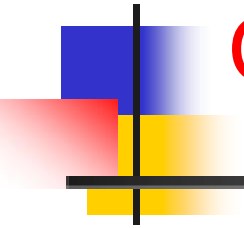
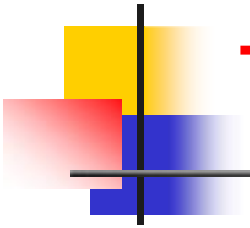


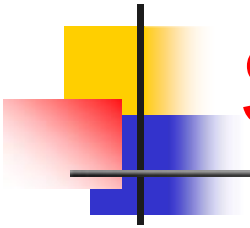
# Generative (Non-structural) Recursion

Comp 311
Rice University
Corky Cartwright



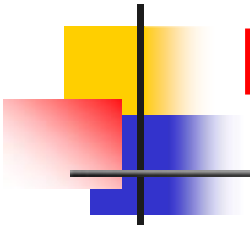
# The Recipe Until Now

- Data analysis and design using structural recursion templates (with minor cheating in **`merge`**)
- For each function in the design (visible interface)
  - Define the data
  - Type & behavioral contracts
  - Examples (stated as tests)
  - Template Instantiation
    - Follows the structure of the data we consume using the best well-founded ordering
    - Using this template, we can do "almost everything”
  - Coding
  - Testing (trivial if Examples are written using **`check-expect`**)



# Structural Recursion

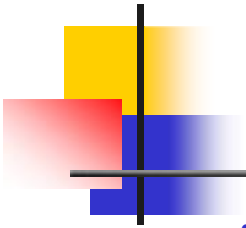
- Best problem-solving strategy
  - For the vast majority of functions over recursive data.
  - Yields satisfactory efficiency in most cases.
- Yet cannot, in principle, compute all computable functions.
  - Can only express primitive recursion for those of you who know some computability theory! All structurally recursive (primitive recursive) programs terminate
  - Ill-suited to an important class of problems that technically can be solved using structural recursion but can be solved more cleanly and efficiently using non-structural methods.



# Non-structural Functional Programs

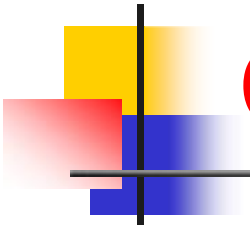
- Best explained by presenting some examples before discussing the general template.

  Problem: efficiently sort a list of numbers
  Good solutions: merge-sort, quicksort



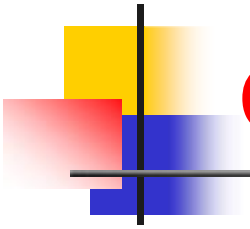
# Merge Sort

- You have already seen this example in Assignment 2. Idea:
  - Base case: list of length 0 or 1
  - Inductive case:
    split the list into two (almost) equal parts
    sort each part
    merge the two results
- Why non-structural? We recursively sort two “sub-lists” (but not tail lists), so this is not structural recursion. But the two lists are strictly smaller (about half the size) of the original list.
- Even the merge operation technically does not use structural recursion but it is so close (particularly when written an OO language) that it might be overlooked.



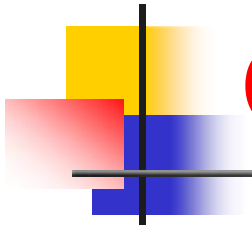
# QuickSort

- Invented by C.A.R. ("Tony") Hoare
- Functional version is derived from the imperative algorithm on arrays; less efficient on lists but still works surprisingly well
- Idea:
  - Base case: list of length 0 or 1
  - Inductive case:
    - partition the list into the singleton list containing first, the list of all items <= first, and the list of all items > first
    - sort the the lists of lesser and greater items
    - return (sorted lesser) + (first) + (sorted greater) where + means list concatenation (append)



# Quicksort Breaks Structural Template

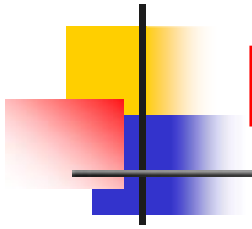
```
(define (qsort l)
  (cond [(empty? l) empty]
        [else
         (local ((define pivot (first l))
                 (define other (rest l)))
           (append
             (qsort [filter (lambda (x) (<= x pivot)) other])
             (list pivot)
             (qsort [filter (lambda (x) (> x pivot)) other])))]))
```



# Quicksort Still Terminates

```
(define (qsort l)
  (cond [(empty? l) empty]
        [else
         (local ((define pivot (first l))
                 (define other (rest l)))
           (append
             (qsort [filter (lambda (x) (<= x pivot)) other])
             (list pivot)
             (qsort [filter (lambda (x) (> x pivot)) other])))]))
```

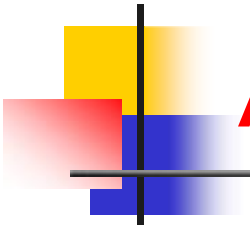
Why?



# Not so QuickSort

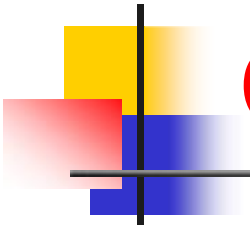
```
(define (qsort l)
  (cond [(empty? l) empty]
        [else
         (local ((define pivot (first l))
                 (define other (rest l))
           (append
             (qsort [filter (lambda (x) (<= x pivot)) rest])
             (list pivot)
             (qsort [filter (lambda (x) (> x pivot)) rest])))])))
```

What if `(first l)` is the largest element in l?



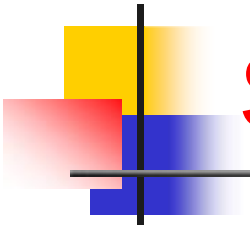
# A More General Recipe

- Data analysis and design
- Contract & purpose, header
- Examples
- Template Instantiation
  - **Much more flexible than before (non-structural) recursive calls are applied to simpler argument lists**
- Coding
- **Explicit termination argument**
  - typically a well-founded measure of the argument list that strictly decreases
- Testing



# Generative Template

```
(define (gen-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
    (determine-solution  problem)]
    [else
     (combine-solutions
        ... problem ...
        (gen-recursive-fun (gen-problem-1 problem))
         …
        (gen-recursive-fun (gen-problem-n problem)))]))
```

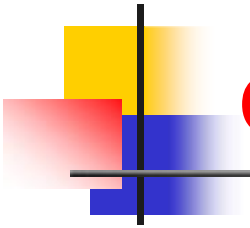


# Sample termination argument

- Quicksort terminates because each recursive call `(qsort l)` reduces the metric `(length l)`. In particular, both

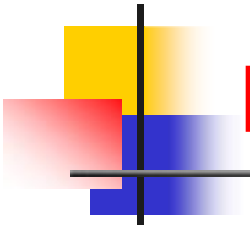
  `[filter (lambda (x) (<= x pivot)) other]` and
  `[filter (lambda (x) (> x pivot)) other]`

  are sublists of `other` which is shorter than `l`
- Without such an argument a non-structural program must be considered incomplete.



# General framework for proving termination

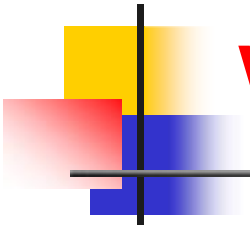
- Devise a metric (a size function) with some familiar well-founded structural type as the output (usually nat) for the problem and show that each recursive call involves a smaller problem than the original one.
- In pathological cases, this ordering may require the use of lexicographic ordering on *n*-tuples (or unbounded sequences) of data values. These pathologies are *rare* in practice. Not aware of a single occurrence in DrJava code base.



# Precise Termination Arguments

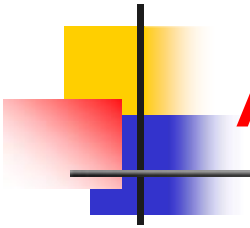
Binary search fallacy

- If we start with an interval S wide, then we only need limited number of steps to reach an interval R wide. In particular, the intervals will proceed as S, S/2, S/4, ..., and will reach size smaller than R in $\log_2$ (R/S) steps …
- We are engaging in perilous handwaving, because when S reaches 2, the details involving comparison (<, <=) and interval representation are critical. Sloppy reasoning/coding confusing properties of the rationals rather than natural numbers leads to non-termination.



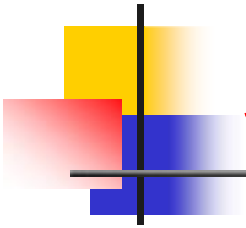
# Why Generative Recursion?

- What if we can choose between
  - a structural solution and
  - a generative solution?
- Often, the second is much faster
  - Sorting
  - Simpler example from book: greatest-common-divisor (GCD)
    gcd(6,9)=3, gcd (99, 18) = 9, etc.
    structural version so brain-damaged for this problem that I could not follow the narrative. I had to infer what the code did.
    Rant: local functions in the HTDP book often have no contracts!
  - Even better example: searching an ordered list where direct access has constant cost, *e.g.*, an array. (Binary Search)



# Are all data types structural?

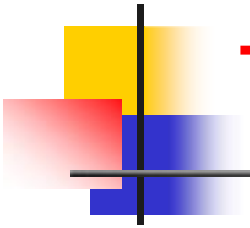
- Structural (inductively defined) => well-founded? Yes, if all constructors are strict (preserve divergence). Not necessarily if constructors can be non-strict ("lazy") implying tree structure can be infinite (such as infinite lists).
- Reasoning about limit points (infinite objects) is a technically hard question.
- We avoid infinite trees if possible, eliminating the ugly case of limit points! But in the reali world, we cannot completely avoid infinite trees. Infinite trees (or similar infinite constructions) are required to formalize some forms of data like functions, real numbers, and infinite streams and trees.
- Question: Is the structural ordering always useful in proving properties of a type? In my view, yes. But structurally inductive reasoning becomes delicate because passing to the limit (reasoning about infinite objects can be delicate). Fixed-point induction works tolerably well. Co-induction is another option, but not to my personal taste.
- How do we define the domain of functions $A \to B$? The standard answer is non-structural and non-computational. Dana Scott (in 1970) showed how the define $A \to B$ can be defined using limits with (in my view) astounding consequences, namely the cardinality of $A \to B$ never exceeds the continuum (real numbers). This subject ("domain theory") is even beyond the scope Comp 411. Very few Computer Science departments have courses that cover this material.
- It is possible (but technically difficult) to formalize all forms of program data including computable sequential functions (with the natural approximation ordering) with only well-founded orderings.



# Some Generative Algorithm Families

- Sorting and Searching
- Mathematical iteration: bisection, Newton's method. (Dirty secret: real numbers truly are limits of Cauchy sequences.)
- Backtracking (traversing a maze, 8 queens)
- Dynamic Programming (memoization)

Generally the structural algorithms are so trivial that they typically aren't discussed as *algorithms.* Nothing interesting to say. Some algorithms are structural (like depth-first search) but this fact is not immediately apparent.



# The Tradeoff (if we can chose)

- How do we chose between
  - a structural solution and
  - a generative solution?
- Speed vs. clarity (structural recursion); speed often wins in practice. Generative solutions are typically nearly as clean as structural solutions. (Are quicksort and mergesort significantly harder to understand than insertion sort?) Termination proofs are usually easy.
- In some cases, there is no *credible* structural algorithm, *e.g.*, for most graph problems. The structural algorithm may be goofy and non-intuitive.
- Chapter 26 in HTDP presents a structural algorithm for computing GCD, which is a good example. I studied it last year but I don't remember how to do it. Euclid's algorithm is so much nicer.