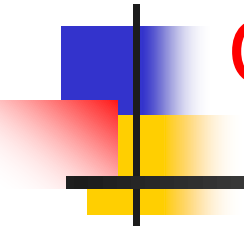
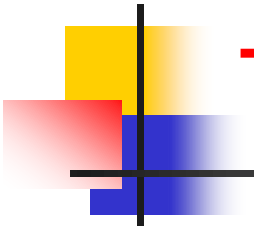


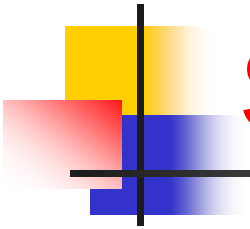
# Generative (Non-structural) Recursion

Comp 311
Rice University
Corky Cartwright



# The Recipe Until Now

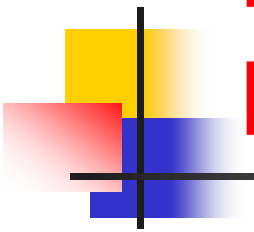
- Data analysis and design using structural recursion templates (with minor cheating)
- For each function in the design (visible interface)
  - Contract, purpose
  - Examples (stated as tests)
  - Template Instantiation
    - Precisely followed the structure of the data we consume
    - Using this template, we can do "almost everything”
  - Testing



# Structural Recursion

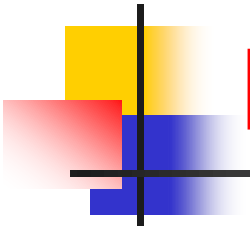
- Is the best problem-solving strategy
  - For the vast majority of functions over recursive data.
  - Yields satisfactory efficiency in most cases.
- Cannot, in principle, compute all computatble functions. Why not? Only primitive recursion!
- Ill-suited to an important class of problems that technically 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.

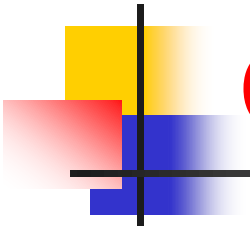
  Problem: efficiently sort a list of numbers
  Good solutions: merge-sort, quicksort



# Merge Sort

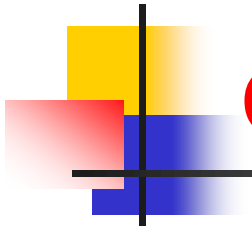
- Not going to present the actual program; you can do it as a “finger exercise” using the merge function you wrote for Assignment 2 and a modest amount of additional code.
- Idea:
  - Base case: list of length 0 or 1
  - Inductive case:
    split the list into two (almost) equal parts
    sort each part
    merge the two results

Why non-structural?



# Quick Sort

- Invented by C.A.R. ("Tony") Hoare
- Functional version is derived from the imperative (destructive) algorithm; less efficient but still works very 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)



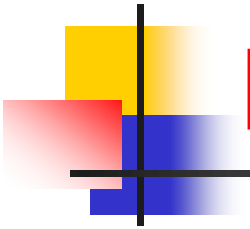
# Quicksort Breaks Structural Template

```
(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])))]))
```

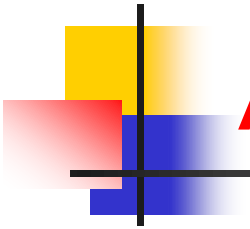
Why?



# Not so quick sort

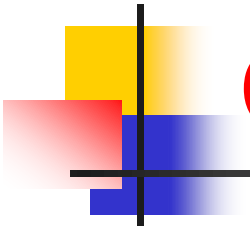
```
(define (qsort l)
  (cond [(empty? l) empty]
        [else
         (local ((define pivot (first l)))
           (append
             (qsort [filter (lambda (x) (<= x pivot)) l])
             (qsort [filter (lambda (x) (> x pivot)) l])))]))
```

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



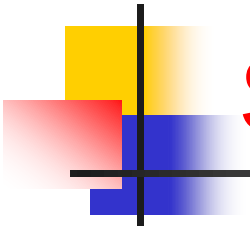
# A More General Recipe

- Data analysis and design
- Contract, purpose, header
- Examples
- Template Instantiation
  - **A bit more flexible than before (non-structural)**
- **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)))]))
```

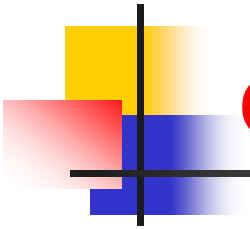


# Sample termination argument

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

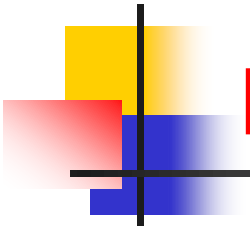
  `[filter (lambda (x) (<= x pivot)) other]` and
  `[filter (lambda (x) (> x pivot)) other]`

  are proper 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

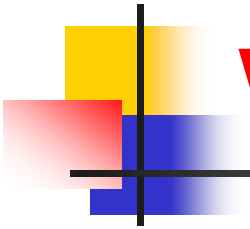
- Devise a metric (a size function) with some familiar well-founded structural type as the output (usually nat) for the problem and show that each recursive call involves a smaller problem than the original one.
- In pathological cases, this ordering may require the use of lexicographic ordering on *n*-tuples (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

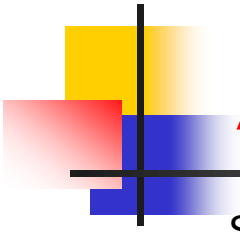
## Binary search fallacy

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 leads to non-termination.



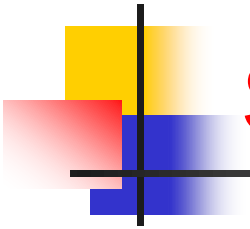
# Why Generative Recursion?

- 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 I could not follow the narrative. I had to infer what the code did.
    Rant: local functions in 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?

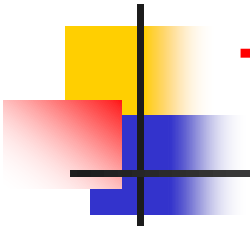
- Structural => well-founded? Not necessarily if structure can be infinite.
- Reasoning about limit points (infinite objects) is a technically hard question.
- If we avoid infinite trees, the answer is yes! But we cannot completely avoid infinite trees. Infinite trees (or similar infinite constructions) are required to formalize some forms of data like functions, real numbers, and infinite streams.
- 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 $A \rightarrow B$? The standard answer is non-structural and non-computational. Dana Scott (in 1970) showed how $A \rightarrow B$ can be defined computationally with (in my view) astounding consequences, namely the cardinality of $A \rightarrow B$ never exceeds the continuum (real numbers). This subject ("domain theory") is even beyond Comp 411.
- It is possible (but technically difficult) to formalize all forms of program data including computable functions (with the natural approximation ordering) with only well-founded orderings.



# Some Generative Algorithm Families

- Sorting and Searching
- Mathematical iteration: bisection, Newton's method.
- 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.



# 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. Termination proofs are usually easy.
- In some cases, there is no *credible* structural algorithm. The structural algorithm(s) may be goofy.
- Chapter 26 in HTDP has a very nice example
  - Greatest-common-divisor (GCD)
    gcd(6,9)=3, gcd (99, 18) = 9, etc.