Midterm Review

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 8.8

Introduction

- This lesson is a review of the main points of the first part of this course.
- These are mostly slides you should remember, or remixes of some of those slides.

The Point

1. It’s not calculus. Getting the right answer is not enough.

2. The goal is to write beautiful programs.

3. A beautiful program is one that is readable, understandable, and modifiable by people.

Remember the Point!

Principles for writing beautiful programs

1. Always remember: Programming is a People Discipline

2. Represent Information as Data; Interpret Data as Information

3. Programs should consist of functions and methods that consume and produce values

4. Design Functions Systematically

5. Design Systems Iteratively

6. Pass values when you can, share state only when you must. We haven't gotten to this one yet

How to Design Functions Systematically

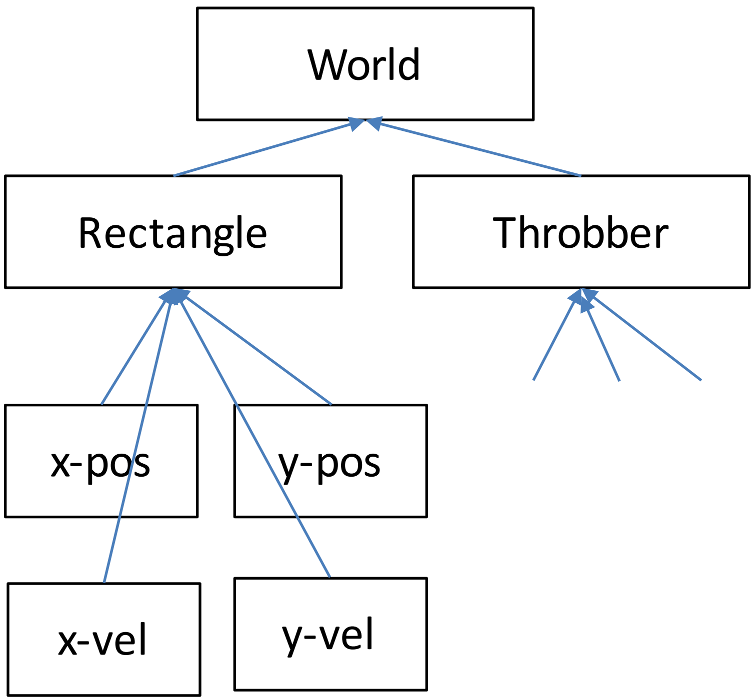
The Design Recipe
1. Information Analysis and Data Design
2. Contract and Purpose Statement
3. Examples
4. Design Strategy
5. Function Definition
6. Tests

Everything starts from the Design Recipe

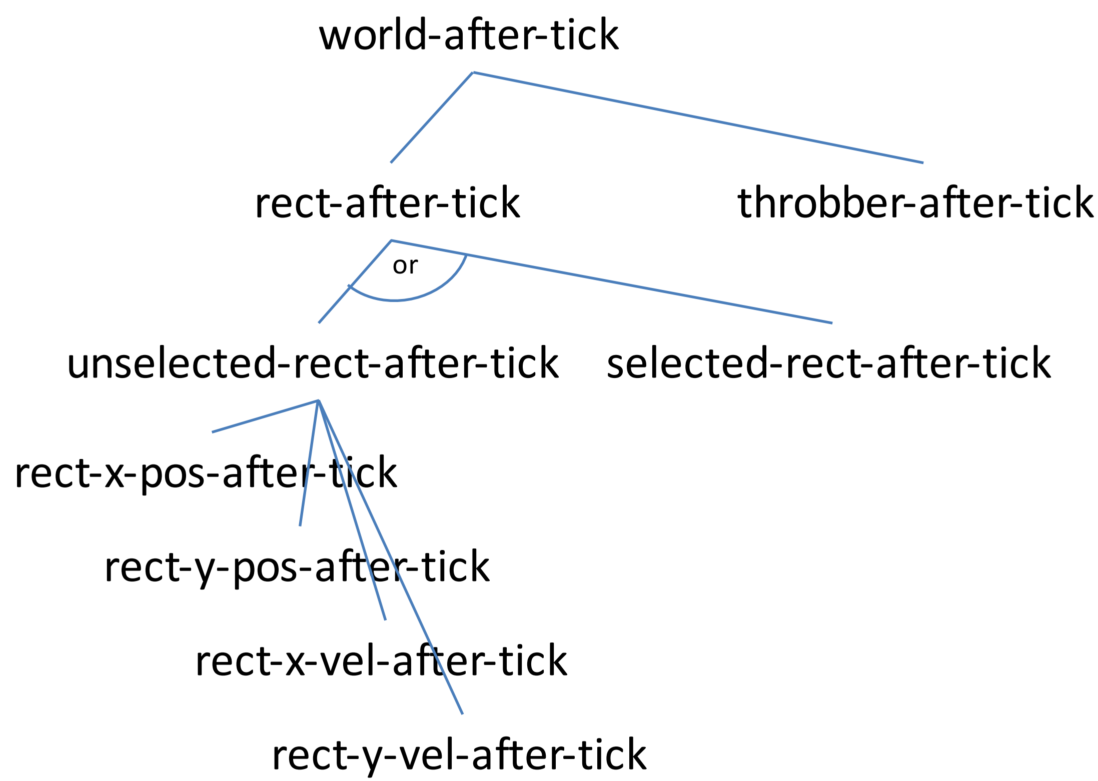
Design functions systematically

- Follow the recipe!
- The structure of the data tells you the structure of the program.
  - Or at least gives you good hints!
  - Data Definition → Template → Code
  - The data definitions structure your wishlist, too.
- Examples make you clarify your thinking
  - Be sure to cover corner cases

The Structure of the Program Follows the Structure of the Data



A Portion of the Data Tree



A Portion of the Program Tree
(your wishtree)

Maybe this won't work out in every detail, but it gives you a plan!

The Recursion Recipe

Recursion and Self-Reference
Represent arbitrary-sized information using a *self-referential* (or *recursive*) data definition.
Self-reference in the data definition leads to self-reference in the template
Self-reference in the template leads to self-reference in the code.

Typical Program Design Strategies

Design Strategies
1. Combine simpler functions
2. Use template for <data def> on <value>
3. Divide into cases on <condition>
4. Use HOF <mapfn> on <value>
5. Call a more general function
6. General Recursion
7. {Initialize|Update} sta

If you were tweeting out a description of how your function works, what would you say?

Choosing a Design Strategy

- If there are independent/sequential pieces, then combine the simpler functions.
- Is your problem a special case of another problem that might be easier to solve? If so, solve the more general problem, and then use generalization.

e.g. number-list => number-list-from, mark-depth => mark-depth-from, 8-queens => n-queens

- Otherwise, find one or more simpler instances of same problem:
  - Is the input a list? If so, consider using a HOF.
  - Is the simpler instance a substructure of the original? If so, use the template.
  - Otherwise, use general recursion.

You've been doing this all term, so you probably know this. But it's worth writing down anyway.

Using a higher-order function

- one of the inputs is a list of values
- you need to treat all the values in the list the same way and combine them the same way.
- if your function doesn't look at all the elements of the list, then probably an HOF is not suitable.
- look at the types to help choose the right HOF.
- you can write special-purpose HOFs for other kinds of tree-structured data

Using a template

- inputs are always *structured* (enumeration, compound or mixed) *data*;
- the function's organization is based on the *data definition* for one (or more) of the function's parameters
- one function per interconnected data definition
- *recursions in the functions follow recursions in the data definitions.*
- are some of the decisions or transformations complicated? Then introduce helper functions
  - There's a reason for that ugly little thing– document it and test it.

General Recursion

- Inputs encode problems from a *class of problems*
- Recursion solves *a related problem* from the same class ("*subgoal*" or "*subproblem*")
  - requires ad hoc insight to find a useful subproblem.
- Termination argument is required:
  - *how are each of the subproblems easier* than the original problem?
  - formulate this as a *halting measure*.

General Recursion vs. Structural Decomposition

- Structural decomposition is a special case of general recursion: it's a standard recipe for finding subproblems that are guaranteed to be easier, because a field is always smaller than the structure it's contained in.
- How to tell the difference between structural and general recursion:
  - In the definition of function **`f`** :

    `(... (f (rest lst)))` is structural decomposition

    - we're calling **f** on a substructure of **lst**

    `(... (f (... (rest lst)))` is general recursion

    - we're calling f on something that depends on **(rest lst)**, but it's not **(rest lst)** itself.

Invariants (1)

- Your function may need to rely on information that is not under its control
  - eg: an inventory has at most one entry for any ISBN
  - eg: the rectangle is unselected
  - eg: k = (length lst)
  - eg: u = (z+1)^2
- Record this assumption as an invariant (WHERE clause).

Invariants (2)

- If your contract is **f: Something -> ??**, then your function has to give the right answer for every possible **Something**.
- An invariant (**WHERE** clause) limits the function’s responsibility.
- If you have a **WHERE** clause, the function is only responsible for giving the right answer for inputs that satisfy the invariant.
- **f**’s caller is responsible for making sure that the invariant is satisfied.

Summary

- We've reviewed the big take-away points from the first half of the course.
- Next: we will move on to classes and objects.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Do Problem Set 8.