**COMP 210, FALL 2000**
**Lecture 23: More Generative Recursion**

**Reminders:**
⇨ Homework is due **Friday** in class. (*mea culpa, mea culpa*)

**Review**

1. We built two programs that did not follow our by-now classic template: qsort and sierpinski. In both cases, the recursion relations in the program arose from the problem rather than from the data. We introduced the notion that this is "*generative recursion*" as opposed to the *structural recursion* that we've seen in the first *k* weeks of COMP 210.

**Generative Recusion**
What are the similarities between QuickSort and Sierpinski? Both programs seem to violate our template model. They contain a new kind of recursion that does not arise from the structure of the information that they process. Instead, the recursion occurs as some innate part of the way that the problem was defined.

- In QuickSort, the algorithm operates by creating (at each step) two smaller lists that must be sorted and then merging them together with **append**. Here, the recursion comes from some insight into sorting.
- In Sierpinski, the algorithm derives the midpoints of the current triangle's sides. Connecting these midpoints creates four smaller triangles. The program draws the inner one and recurs on each of the outer ones.

We call this style of recursion "generative recursion," since the program proceeds by generating subproblems and solving them recursively. Both QuickSort and Sierpinski use a *divide and concur* approach to problem solving. They take the problem, split it into smaller instances of the same problem & solve those problems.

QuickSort recurred on successively smaller problems until it reached the degenerate case of a single number to be sorted. (Some of its speed comes from the fact that, at each step, it pulls the pivot element out of consideration.) Sierpinski, on the other hand, recurred until it reached some resolution limit imposed by the function **too-small?**, at which point it terminated the recursion.

QuickSort solved the original problem by explicitly combining the solutions to the smaller problems; Sierpinski combined them, but in an implicit way rather than in an explicit way. The only reason that we run Sierpinski is to execute the various calls to **draw-triangle**. When **draw-triangle** executes, it changes some pixels on the screen to form lines. The lines are persistent, so the effect is a superposition of all the triangles–- achieving a visual effect that is analogous to the combination caused by the **append** in QuickSort.

**What About Our Methodology?**
Our design methodology for structural recursion should be engraved in your hearts by now. The steps are

- ⇨ Data analysis and design, including examples of the data
- ⇨ Contract, purpose, & header
- ⇨ Construct test cases for the program
- ⇨ Write the template
- ⇨ Fill in the program's body
- ⇨ Test the resulting program (against results of 3)

Do these same steps make sense for generative recursion? Most of them do. We still need to do data analysis and construct examples–-we cannot develop the program if we don't have the data definitions. Every program *needs* a contract, purpose, and header.

When we generate test cases, we need two kinds of test cases: those that test the limits or boundaries of the data. [For example, what happens on QuickSort of an empty list?] We also need examples that demonstrate how the program (or algorithm) operates. These should be similar to the worked out examples that we did on the board for Quicksort. These worked out examples help to solidify the operation of the algorithm–-the nuts and bolts of how it works.

We also need to use a template that is appropriate for generative recursion. The implementations of QuickSort and Sierpinski have some common elements. Both use a **cond** that separates the trivial (or degenerate) case from the recursive case. The recursive case decomposes the problem into smaller problems and solves them. The trivial case halts the recursion–-either because the problem is small enough to solve directly, as in QuickSort, or because there is no point in proceeding further, as in Sierpinski. Picking out these cases requires problem specific knowledge.

```
(define (gen-recur-func    problem-data)
  (cond
     [(trivial-to-solve?  arg1  … argn)  (solve  arg1 … argn)]
     [else
        (combine-solutions
           … (gen-recur-func (generate-problem1 problem-data)) …
           …
           … (gen-recur-func (generate-problemk problem-data)) …
        )]))
```

This template doesn't give us as much specific guidance as the structural recursion template, but it does lay the groundwork for writing program that used generative recursion. In the structural recursion template, we just had to fill in the missing parts. In this template, you have to replace the various parts of the template with code that implements that part of the program. Thus, in QuickSort, the function **trivial-to-solve?** became the familiar test **empty?** and the solution for that case was to return **empty**. In contrast, the trivial case in Sierpinski actually required some computation to detect–-the program must compute the distance between two of the points and compare it to some arbitrary threshold (set on the basis of the appearance of the picture on the screen). To fulfill the contract, the trivial case returned **true**.

In general, there are a series of questions that we need to ask about the problem before we can develop all the code. These questions will often lead to other, less formulaic, questions. Among the questions we should ask are:

- ⇨ What is a trivial instance of the problem?
- ⇨ What is the solution to a trivial instance?
- ⇨ How do we generate one or more smaller subproblems from the original problem?
- ⇨ How many subproblems should we generate?
- ⇨ Is the solution to the subproblem the solution to the original problem, or do we need to combine the solutions from several subproblems?
- ⇨ How do we combine the solutions from subproblems (if that is necessary)?

In the design of programs that use generative recursion, step 4 “write the template” is much more involved than it is in programs based on structural recursion. (The complex version of programs that work with multiple complex arguments began to have some analysis, but it was conceptually simpler than the process for generative recursion.)

- ⇨ At this point, I made a mistake with the example and the lecture fell apart !!!
- ⇨ Class was dismissed.