COMP 210, FALL 2000
Lecture 23: More Generative Recursion

Reminders:

⇨ Homework is due **Monday** in class. It's a short assignment.

Review

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

1. Data analysis and design, including examples of the data
2. Contract, purpose, & header
3. Construct test cases for the program
4. Write the template
5. Fill in the program's body
6. 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:

1. What is a trivial instance of the problem?
2. What is the solution to a trivial instance?
3. How do we generate one or more smaller subproblems from the original problem?
4. How many subproblems should we generate?
5. Is the solution to the subproblem the solution to the original problem, or do we need to combine the solutions from several subproblems?
6. 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.)

An Example –- Binary Search

Let's write another program–-one that plays a simple "guess the number game." The program **hi-lo** consumes two numbers that it takes as the endpoints of an interval.

⇨ **Data analysis: hi-lo** works by taking a closed interval, represented by two numbers. The numbers must be integers. They have a simple, natural representation in Scheme as numbers. The hidden number will be represented with a function **guess**.

;; guess: number –> ‘higher or ‘equal or ‘lower

⇨ **Contract, Purpose & Header** for **hi-lo:**

```
;; hi-lo:  integer integer –> integer
;; Purpose: given lo & hi, return the number hidden by guess
;;  Assume that the hidden number lies in [lo, hi]
(define (hi-lo  lo  hi) … )
```

⇨ **Examples:**

If guess hides 3, then (hi-lo 0 12) ➔ 3
If guess hides 5 then (hi-lo 5 15) ➔ 5
If guess hides 10 then (hi-lo 0 10) ➔ 10

With a hidden 3, (hi-lo 0 12) should operate as

```
midpoint of 0-12 is 6, (guess 6) ➔ ‘lower
recur on [0 6]
    midpoint of 0-6 is 3
    (guess 3) ➔‘equal
    return 3
return 3
```

This strategy, called binary search, generalizes to

```
Compute midpoint of [lo hi] and guess it.
If  (guess midpoint) is ‘equal, return midpoint
If  (guess midpoint) is ‘lower, recur on (0,midpoint)
If  (guess midpoint) is ‘higher, recur on (midpoint, 0)
```

This strategy is actually a form of generative recursion. The notion of computing the midpoint and using that to narrow the search comes from the problem–-searching an interval–-rather than from the structure of the data–-a pair of points that represent the interval and a function **guess** that hides the number.

Using The Generative Template

How can we fit the **hi-lo** game into this framework?

a) *What is the trivial case?* It occurs when the (guess midpoint) returns ‘equal?

b) *The corresponding solution?* Return midpoint

c) *Generate new problems?* Pick the appropriate sub–interval (either [0 midpoint] or [midpoint hi]) and recur

d) *Generate one or more new provlems?* Generate one, because we can prove that we don't need to look at the other one. (I.e., guess "proved" that the other interval is irrelevant)

e) *Do we need to combine solutions from subproblems?* No. [This obviates the need for the final question–-"How do we combine them?"

1. **Filling in the Program's Body**

   The answers tell us a couple of things that shape the problem. First, we need to compute this midpoint. We are going to reference it several times in the code, so it might go into a local. Similarly, we need to compute (guess midpoint) and save it, since we will use it to distinguish between the trivial case and the two other cases.

   ➔ This says we need a local inside **hi-lo** that accomplishes these things for us.

```
(define (hi-lo   lo hi)
  (local [(define midpoint  (/ (+ lo hi) 2))
          (define answer      (guess midpoint))]
         (cond
            [(trivial-to-solve?  arg1  … argn)  (solve  arg1 … argn)]
            [else
                  (combine-solutions
                     … (hi-lo (generate-problem1 problem-data)) …
                     …
                     … (hi-lo (generate-problemk problem-data)) …
  )]))
```

Now we can start filling in the **cond** construct. It's going to take some major renovations.

```
(define (hi-lo   lo hi)
  (local [(define midpoint  (/ (+ lo hi) 2))
          (define answer      (guess midpoint))]
         (cond
            [(symbol=?  answer midpoint) midpoint]
            [else
              (cond
                    [(symbol=?  answer ‘higher)  (hi-lo  midpoint  hi)]
                    [(symbol=?  answer  ‘lower)  (hi-lo  lo  midpoint)] )]
                ))
```

Of course, we can simplify the cond within the cond as

```
(define (hi-lo   lo hi)
  (local [(define midpoint  (/ (+ lo hi) 2))
          (define answer      (guess midpoint))]
         (cond
            [(symbol=?  answer midpoint) midpoint]
             [(symbol=?  answer ‘higher)  (hi-lo  midpoint  hi)]
             [(symbol=?  answer  ‘lower)  (hi-lo  lo  midpoint)] ))
```

2. **Testing the Program**

Try it on our first example, guessing 3 in [0,12]. This should work.

What about guessing three in [0,15]?

➔ First recursion breaks the contract for **hi-lo** because it passes 7.5 as **hi**. Fix this by truncating the midpoint when it is passed. You can justify this by noting that the guess must be an integer and the interval between the midpoint and the truncated value cannot contain an integer.

Now try [0, 15] again with a hidden three.

```
(hi-lo  0 15)   mid = 7.5
        ⇨  (hi-lo  0 7)  mid = 3.5
        ⇨  (hi-lo  0 3)  mid = 1.5
        ⇨  (hi-lo  1 3)  mid = 2.5
        ⇨  (hi-lo  2 3) mid = 2.5
        ⇨  (hi-lo  2 3) mid = 2.5
        ⇨  (hi-lo  2 3) mid = 2.5    uh oh, things have gone awry.
```

In the common parlance, we are in an infinite loop. [Of course, no loop has every been infinite; they’ve all terminated somehow, except the ones still running.]

What went wrong? Our techique for generating new problems reached a case where it generated the same problem again, rather than generating a new problem. With structural recursion, the template guaranteed that we always recurred on a smaller problem. With generative recursion, it is much easier to write programs that recur on the same problem. This almost always produces a program that fails to terminate–-or recurs forever.

See next lecture