COMP 210, FALL 2000
Lecture 24: Termination Conditions

Reminders:

1. Homework is due **today** in class. The next homework will be available Tuesday, due some time in the future.

Review

1. We talked about the changes to our methodology that are necessitated by generative recursion. In particular, we need to develop two kinds of examples and test cases now–-those that test for end conditions [such as (qsort empty)] and those that illustrate the workings of our proposed solution technique –- or algorithm.
2. We then turned to one final example of generative recursion, and I picked the wrong test data to illustrate it. This dug me into a hole from which I could not recover, so we ended class early. Today, we will revisit that example.

Another 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.

a) **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

b) **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) … )
```

c) **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.

d) **The Generative Template**

Last class, we derived a template for programs that involve generative recursion.

```
(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)) …
    )]))
```

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?”

5. **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)] ))
```

6. **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.

This behavior is bad enough that we need to add a step to the design methodology that specifically addresses it. We must write out a *termination argument.*

The termination argument should explain why the divide and conquer approach must eventually produce a trivial case of the problem. You should write it in your code as a comment –- right after the test cases and examples. For example, with Sierpinski, we might have written something such as:

> **Termination:** At each step, sierpinski partitions the input triangle into three triangles whose sides are strictly smaller than those of the original. The trivial problem test checks whether the sides are smaller than an externally supplied threshold. Since the sides grow smaller on each recursive call, they must eventually be shorter than the threshold value. When that happens the trivial problem test will succeed and the recursion will terminate.

[Note that this argument provides a clue to one possible problem in the use of sierpinski–-if we invoke sierpinski with a negative threshold, the recursion will not terminate!]

So what went wrong with hi-lo?

We could check to see if the answer is either hi or lo. This would have caught our problem case.

```
;; hi-lo: int int -> int
;; Purpose: given low & high, return the hidden number in [low, high]
(define (hi-lo lo hi)
  (cond  [(symbol=? (guess lo) 'equal)  lo]
         [(symbol=? (guess hi) 'equal) hi]
         [else
                (local [(define  mid   (/  (+ lo hi) 2))
                          (define answer  (guess mid))]
                       (cond
```

```
      [(symbol=?  answer  'equal)    mid]
      [(symbol=?  answer  'higher)  (hi-lo  (truncate mid) hi)]
      [(symbol=?  answer  'lower)
        (hi-lo  lo (truncate mid))])))  ] ))
```

With this fix, the termination argument becomes

> **Termination:** At each step, hi-lo checks whether the number to guess is one of the endpoints of the range. If this is not the case, then the number must be strictly between the endpoints. If the number is strictly between the endpoints, then the recursive call to hi-lo must be over a strictly smaller range. As the range gets strictly smaller on each call, it must either reach a point where it finds the number or reach a case where the endpoints differ by one. In this latter case, the number to guess must be one of the endpoints and the test for endpoints will solve it.

Another way to fix it: Notice that the answer, 3, showed up relatively early in the computation as a boundary point. Our check of **mid** did not catch it, because mid had a non-integer value. We passed the truncated value to the recursive calls, but tested the non-truncated value. This leads to a simple fix

```
;; hi-lo: int int -> int
;; Purpose: given low & high, return the hidden number in [low, high]
(define (hi-lo lo hi)
  (local [(define  mid   (truncate (/  (+ lo hi) 2)))
          (define answer  (guess mid))]
         (cond
           [(symbol=?  answer  'equal)    mid]
           [(symbol=?  answer  'higher)  (hi-lo  mid hi)]
           [(symbol=?  answer  'lower)   (hi-lo  lo mid)])))
```

Here, the termination argument is similar.

Try it on (0 15) to find 3

Try it on (0 2) to find 0

Try it on (0 2) to find 2 –- ***ooops***…

The problem here is that the midpoint can never equal the original upper bound. We can fix this in one of two ways. We can test for the original upper bound as a special case–-it happens once per binary search rather than repeatedly. We put the current version of hi-lo into a local and wrap around it a check for the upper bound and a recursive call.

The termination condition for this version becomes:

> **Termination:** On each recursive call to hi-lo-helper, lo and hi are either one of the endpoints of the original interval, or some value computed as a midpoint of an interval. The program explicitly checks every value computed as a midpoint against the guess, along with the upper bound of the original interval. The program will terminate if the hidden number is one of these value. (Thus, we

must show that the sequence of midpoints, along with the original upper bound, can become any number in the original interval.)

On each recursive call, the range between hi and lo becomes smaller, by roughly one-half. The smallest case that we reach is when hi = lo + 1. At that point, hi cannot be the hidden number–-either it is the original upper bound and was explicitly checked, or it was computed as a midpoint, and explicitly checked. The midpoint of [lo,lo+1] will become lo, so lo will be explicitly checked.

At each recursive call, we now, constructively, that the hidden number lies between lo and hi. As long as the original interval contains the hidden number, this binary search will find it and terminate.

Another way to fix **hi-lo** is to leave the midpoint out of the new intervals. (After all, it has been checked and found uninteresting.)

```
;; hi-lo: int int -> int
;; Purpose: given low & high, return the hidden number in [low, high]
(define (hi-lo lo hi)
  (local [(define  mid   (truncate (/  (+ lo hi) 2)))
          (define answer  (guess mid))]
         (cond
           [(symbol=?  answer  'equal)    mid]
           [(symbol=?  answer  'higher)  (hi-lo  (add1 mid)hi)]
           [(symbol=?  answer  'lower)   (hi-lo  lo (sub1 mid))])))
```

This version of hi-lo has a much simpler termination condition:

> **Termination**: The range between lo and hi gets strictly smaller on every recursive call. In the extreme case, the interval becomes a single number (hi=lo=midpoint) and the algorithm terminates because guess returns 'equal.

The simplicity of this termination condition argues for using this solution to the problem. Thinking about termination, and writing out termination conditions, led to a much simpler, cleaner termination condition. In turn, that condition generates the simplest solution (which also turns out to do less work because it shrinks the interval more quickly).

In some very real sense, the extent to which you do this kind of structured thinking about termination and correctness determines whether you are a recreational programmer–-someone who hacks together something and checks it on a few simple examples–-or a professional programmer who writes reliable, robust applications.

Note: Many students are uneasy with the creative aspects of developing programs based on generative recursion. If you don't see how to divide up the problem, you cannot write a generative recursive solution. Generative recursive problems arise in many contexts. Some of them have obvious divide & conquer solutions, such as binary search in hi-lo. Others require a true insight, as in C.A.R. Hoare's QuickSort algorithm—the insight there might be termed genius. As you gain practice with this sort of solution, you will discover that it can be both fun and challenging.

How often will you need to write programs that use generative recursion? Most problems have structural recursive solutions. Consider sorting. QuickSort is probably the fastest general purpose sorting algorithm. However, on your homework you wrote two different sorts that were based on structural approaches (insertion sort and merge sort). As a rule of thumb, look for the structural recursion solution first. If they prove to be excessively slow or clumsy, think about the generative recursion solution. If the structural solution works, and is sufficiently fast, be thankful and content yourself with the fact that it took much less time to develop.