COMP 210, FALL 2000
Lecture 24: Termination Conditions

Reminders:
1. Exam is due today at 5pm. Homework is due Monday.

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 developed a program **hi-lo** that implemented binary sort to discover the number hidden by a function **guess**. We began testing it and discovered some problems in the initial formulation of **hi-lo**. (See slide from Lecture 23). Today, we will revisit **hi-lo**.

Back to Hi-lo

When class ended, we had reached the point where our program looked like:

```
;; hi-lo:  number number → number
;; Purpose: consumes the endpoints of an interval and finds the number
;;          hidden by guess. Uses a strategy called binary search to make
;;          this efficient.
(define (hi-lo   lo hi)
  (local [ (define midpoint  (truncate (/ (+ 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)] ))
```

We discovered the need for the **truncate** by getting in trouble with **guess** =3 on [0 15].

This created a situation where the code did not naturally terminate. In common parlance, this is often called and infinite loop. [Of course, no loop has every been infinite; they've all terminated somehow, except the ones still running.]

What went wrong? Our technique for generating new problems generated a case where the inputs violated an implicit, albeit unstated, assumption of the

design–-that the boundaries of the interval, as well as the hidden number, were natural numbers. (That is, they were counting numbers greater than or equal to zero.) We should have stated that in the contract for **hi-lo**. An accurate contract would have led us in the right direction to avoid this mistake.

What about the problem instance **guess** = 2 on [0 2].

(hi-lo 0 2) midpoint = 1, guess = 'higher
  (hi-lo 1 2) midpoint = 1, guess = 'higher
    (hi-lo 1 2) midpoint = 1, guess = 'higher
      (hi-lo 1 2) midpoint = 1, guess = 'higher **… oops, we did it again!**

What went wrong this time? 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 easy to create programs that recur on the same problem. This almost always produces a program that does not terminate–-or recurs forever.

This behavior is bad enough that we need to add a step to the design methodology that specifically addresses it. For any program that uses generative recursion, we must write out an explicit *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? The midpoint can never reach the upper bound of the interval. (Without the **truncate**, it would asymptotically approach the upper bound, but it still would not reach it.) With the **truncate**, it can only get within one of the upper bound. Notice that it can reach the lower bound (eventually).

How can we fix this? We could check to see if the answer is hi. 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 hi) 'equal) hi]
          [else
                (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)])) ] ))
```

With this fix, the termination argument 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.

Another Kind of Problem–-a graph problem

JetSet Air (remember them from lecture six?) was so successful with their computerized system for keeping maintenance records that they want to develop a similar system to manage information about the various routes that they fly. For each city that JetSet serves, it needs, at a minimum, a way of determining the cities that a passenger can reach with a direct flight. That is, for a city *a*, what cities are destinations on flights originating from *a*? Since COMP 210 did such a good job on maintenance, they've asked us to design the data structures and to develop the programs. How will we represent this information?

A city is a symbol.

```
;; The information for a city can be represented as a structure
;;  (make-city-info name dests)
;; where c is a city (symbol) and dests is a list of symbole
(define-struct city (name dests))

;; A route-map is a list of city-info

(define routes
  (list (make-city-info  'Houston   (list 'Dallas 'NewOrleans))
        (make-city-info 'Dallas   (list  'LittleRock  'Memphis))
        (make-city-info  'NewOrleans (list 'Memphis))
        (make-city-info 'Memphis  (list 'Nashville)) ))
```

As a first program, we need a program **find-flights** that consumes a route-map and returns a sequence of cities (not necessarily the shortest sequence) by which we can fly from a starting city to a final city. If no such sequence exists, the program should return **false** (?)

```
;; find-flights: city city route-map → (list of city) or false
;; Purpose: create a path of flights from start to finish or return false
(define (find-flights start finish  rm) …)
```

Examples:

(find-flights ‘Houston ‘Houston routes)
= (list ‘Houston)

(find-flights ‘Houston ‘Dallas)
= (list ‘Houston ‘Dallas)

(find-flights ‘Dallas ‘Nashville)
= (list ‘Dallas ‘LittleRock ‘Memphis ‘Nashville)

How would we write find-flights? If there is a direct flight from start to finish, the route is trivial, as is the program. All we need to do is to walk the list-of-city in the city-info structure for **start** and find **finish**. What if there is no direct flight? The list-of-city in the city-info structure for start gives us all the cities that we can reach in one flight (one hop). We can look through the city-info for the final city–-trying to find a two-hop solution. If that fails, we can look through those two-hop cities for a three-hop flight, and …

Based on this description, we should be able to write **find-flight**. Is this a problem for structural recursion, or for generative recursion? Hint: the answer is **generative**. If there is no direct flight between start and finish, we generate new problems–-flying from the cities that are reachable to finish. These new problems are based on our understanding of how to search for a path through the route map. (To be sure, they rely on information in the route map, but we don’t run over the whole route map in some structurally determined order. Instead, we search outward from start, looking for finish.)

Since the program needs generative recursion, we need to answer the questions that derive from the generative recursion template.

- ⇨ What is the trivial case? When start = finish.
- ⇨ What is the solution to the trivial case? A list containing start.
- ⇨ How do we generate new problems?

  Find all the cities that are destinations from start, and look for a route from one of those cities to finish. (Recur on same route map and finish, new start).
- ⇨ How do we combine the solutions?

  If we find at least one route, we keep one and add the starting city to that route. Otherwise, we return false.

Let’s fill in the code…

```
;; find-flights: city city route-map → (list of city) or false
;; Purpose: create a path of flights from start to finish or return false
(define (find-flights start finish  rm)
  (cond
    [(symbol=? start finish) (list start)]
    [(else
       (local [(define possible-route
                  (find-flights-for-list (direct-cities start rm) finish rm))]
              (cond
                     [(boolean? possible-route)  false]
                     [else  (cons start possible-route)])) ] ))


;; direct-cities: city route-map → list-of-city
;; Purpose: return a list of all cities in the route map with direct flights
from
;;       the city given as an argument
(define (direct-cities  from-city  rm)
  (local [(define  from-city-info
                (filter (lambda (c ) (symbol=? (city-info-name c) from-
city)) rm))]
          (cond
            [(empty? from-city-info)  empty]
            [else (city-info-dests (first (from-city-info))])))

;; find-flights-for-list: list-of-city city route-map → list-of-city or false
;; Purpose: finds a flight route from some city in the input list to the
destination,
;;       or returns false if no such route can be found.
(define (find-flights-for-list aloc finish rm)
  (cond
      [(empty? aloc)  false]
      [else
        (local [(define possible-route
                  (find-flights (first aloc) finish rm))]
           (cond
                [(boolean? possible-route)
                 (find-flights-for-list (rest aloc) finish rm)]
                [else  possible-route]))]))
```

How does this program work? It employs a common algorithmic technique called *backtracking*. It tries a potential solution. If that solution does not work, we go back and try another possible solution, and another, and another, until one of two things happens. Either we find a solution, or we exhaust the possibilities.

What's the termination argument for find-flights? Each recursive call looks for a route that uses fewer flights. Eventually, each path must end in the finish city, or the city has no outbound flights. [This is an oddity of the way we have formulated the route map, but bear with me for a day or two.]

This program works fine on our initial route map. (*Time permitting, work an example from page 1*.) What if we add a flight from 'Dallas to 'Houston?

```
(define new-routes
  (list (make-city-info  'Houston   (list 'Dallas 'NewOrleans))
        (make-city-info 'Dallas   (list  'Houston 'LittleRock  'Memphis))
        (make-city-info  'NewOrleans (list 'Memphis))
       (make-city-info 'Memphis  (list 'Nashville)) ))
```

What happens when we try

```
(find-flights 'Houston 'Memphis new-routes)  ?
```

Let's write down the series of calls that occur.

```
(find-flights 'Houston 'Memphis new-routes)
        (find-flights-for-list (list 'Dallas 'NewOrleans) 'Memphis new-
        routes)
        (find-flights 'Dallas 'Memphis new-routes)
        (find-flights (list 'Houston 'LittleRock 'Memphis) 'Memphis
        new-routes)
        (find-flights 'Houston 'Memphis new-routes)
        … and so on for quite a while …
```

We ended up with a non-terminating evaluation (or an infinite loop). What happened?

First, our termination condition is wrong. It assumes that the route-map has no cycles–ways that we can fly from a to b and from b to a. In our new route-map, we have a cycle ('Houston to 'Dallas and 'Dallas to 'Houston). This clearly causes a major problem with the program. Thus, our original program only works on route-maps that have no cycles (or loops, or strongly-connected components, or …)

ßWhy does it break when it confronts a cycle? Because it has no recollection as to which cities it has already tried. Each recursive call is independent of all the others. If the program is to operate correctly on route-maps (or *s*) that have cycles (called *cyclic graphs*), it will need to remember all of the cities that it has already tried (or *visited*)

One way to handle this problem is to add a new parameter to find-flights that stores the cities already visited (as a list, naturally). Then, find-flights can check the list of already visited cities to avoid redoing work (and hitting a case that causes an infinite recursion).

That modification looks like –- see the slide labelled *"With Institutional Memory"*

What should the initial argument passed to **visited** be? It ***must*** be **empty**. Passing it other values can cause the program to malfunction. For example, if you started it with a list of all the cities in the route map, it would never find any routes except the trivial ones.

Using this on routes2, we find that it terminates without hitting the infinite recursion–-our addition of memory allowed it to prune its search when it started to run over parts of the graph that it had already visited.

This also produces a much simpler termination condition. Because it knows about its own history, find-flights will only search outward from a given city once. Thus, as long as the route map is finite, the search will terminate. [Simple, beautiful termination argument!]

So what is this parameter **visited**? In COMP 210, we call this kind of parameter an accumulator. It accumulates information over the course of the computation and lets the program have access to that information. In effect, it provides the function with a record of where it has been (and, perhaps, the results of some of those earlier computations). The next several classes will look at aspects of designing programs with accumulators. You should also read the material on accumulators in the book.