COMP 210, Spring 2001
Lecture 20: Termination Conditions in Generative Recursion

Reminders:
1. Exam this evening in DH 1055 from 7 to 9:30 P.M.

Review

1. We introduced a new form of recursion call general (or generative) recursion and illustrated it with the program Quicksort. We also noted that mergesort relies on generative recursion.
2. We discussed the changes to our methodology that are necessitated by generative recursion. In particular, we need pay particular attention to producing test cases that satisy termination conditions and test cases that exercise each different form of program decomposition.

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 mergesort. Others require a true insight, as in C.A.R. Hoare's QuickSort algorithm. 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 involving inductively defined data can be solved effectively using structural recursive. Consider sorting. QuickSort is probably the fastest general purpose sorting algorithm. However, for lists of modest size, insertion sort is faster.

As a rule of thumb, look for the structural recursion solution first. If it clumsy to write or has a non-linear complexity, think about the generative recursion solution. If the structural solution works, and is sufficiently fast, be thankful and content yourself with the fact that a simple solution (which is easier to understand, to maintain, and to modify) exists.

Another Kind of General Recursion Problem–-a graph problem

JetSet Air (remember them from Lecture ??) had such a good experiene 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 has a schedule of the flights from that city to other cities. Given that schedule, it needs to be able to

determine what cities can be reached from a give city using a combination of flights.

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.

; A city-info (table of route information for a city) is a

; structure
;  (make-city-info c dests)
; where c is city and dests is a (list-of city) identifying the
; destinations reachable by flights from c
(define-struct city-info (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 start city, a finish city, and a **`route-map`** and returns a list of cities (not necessarily the shortest list) 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
;   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 routes)
= (list 'Houston 'Dallas)

  (find-flights 'Dallas 'Nashville routes)
= (list 'Dallas '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? Does structural recursion work? We can only recur on the **`route-map`** but how does a recursive call

```
  (find-flights … … (rest rm))
```

help solve the original problem? Only if **`(first rm)`** is the **`city-info`** for **`start`**. In this special case

```
  (find-flights dest finish (rest rm))
```

where **`dest`** is a city reachable by direct flight from **`start`** is a subproblem whose solution is part of a solution to the original problem. To implement this approach in general, we would need to delete the **`city-info`** for **`start`** from **`route-map`**. This problem decomposition is not structural recursion.

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.

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 a solvable subproblem, we add the starting city to that route. Otherwise, we return **`false`**. Note that this strategy returns the first path that is found, not necessarily the best one.

Let's fill in the code…

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


; direct-cities: city route-map → list-of-city
;   return a list of the cities in route map with direct flights
;   from from-city
;   assume that from-city appears as name of some city-info in rm
(define (direct-cities from-city rm)
   (cond [(empty? rm) empty)
         [(symbol=? from-city (city-info-name (first rm)))
          (city-info-dests (first rm))]
         [else (direct-cities from-city (rest rm))]))
```

How does this program work? It employs a common algorithmic technique called *depth-first-search* or *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? Assume the set of reachability paths (a list of cities where each city in the list is directly reachable from its predecessor) in **`route-map`** from any city forms a finite tree. Then the depth of this tree decreases in each set of recursive calls in **`find-flights`**. **`routes`** has the finite tree property.

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 'Dallas 'Memphis new-routes) …
= … (find-flights 'Houston 'Memphis new-routnes) …
  OOPS!
```

We produces a non-terminating computation (an infinite recursion). What happened? First, **`new-routes`** does not satisfy the finite tree property; it has a cycle, namely **`'Houston`** to **`'Dallas`** and **`'Dallas`** to **`'Houston`**). Our termination argument depended on the absence of such cycles.

Why does **`find-flights`** 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 an accumulator 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).

Another option would be to delete the **`city-info`** record for a city from the route-map after the city has been visited.

What should the initial value of the accumulator be? It ***must*** be **`empty`** since no cities have yet been visited. Let's write the code.

```
; find-route: city city route-map à (list of city) or false
;   create a list of flights from start to finish or return false
(define (find-route start finish rm)
  (local
      [; find-route-help: city (list-of city) -> list-of-city
       ; path is the list of cities already on the current path
       (define (find-route-help start path)
         (cond
           [(member start path) false]
           [(symbol=? start finish) (list start)]
           [else
             (local
                 [(define route-tail
                    (ormap
                      (lambda (city)
                        (find-route-help city (cons start path)))
                      (direct-cities start rm)))]
               (cond [route-tail (cons start route-tail)]
                     [else false]))]))]
    (find-route-help start empty)))
```

What is our new termination condition? The accumulator grows one element longer on every recursive call and contains distinct elements. Since the number of cities in the route-map is finite, no infinite recursion is possible.

Can we improve this solution? Can we construct the answer within the **`ormap`** search? How close is **`visited`** to the solution? Here is the revised code:

```
(define (find-route start finish rm)
  (local
      [; find-route-help: city (list-of city) -> list-of-city
       ; path is the list of cities already visited
       (define (find-route-help start path)
         (cond
           [(member start path) false]
           [(symbol=? start finish) (reverse (cons start path))]
           [else
            (ormap (lambda (city)
                     (find-route-help city (cons start path)))
                   (direct-cities start rm))]))]
    (find-route-help start empty)))
```

This solution is aesthetically pleasing but it has a serious flaw: the accumulator only records the cities on the current path rather than all cities that have been visited. This mechanism assures termination but it does not prevent searching portions of the route map over and over again. To avoid searching parts of the tree repetitively, we must add a second accumulator

recording all cities that have been visited and replace **ormap** by a help function that propagates this information.

```
(define (find-route start finish rm)
  (local
    [; find-route1: city (list-of city) (list-of-city) -> list-of-city
     ; path is the list of cities on the current path
     (define (find-route1 start path visited)
       (cond
         [(member start visited) false]
         [(symbol=? start finish) (reverse (cons start path))]
         [else
           (find-route-list (direct-cities start rm)
                            (cons start path)
                            (cons start visited))]))
     ; find-route-list: (list-of city) (list-of city) (list-of city) ->
     ;   (list-of city)
     (define (find-route-list cities-list path visited)
       (cond
         [(empty? cities-list) false]
         [else
          (local [(define city (first cities-list))]
           (or (find-route1 city path visited)
               (find-route-list
                 (rest cities-list) path (cons city visited))))]))]
    (find-route1 start empty empty)))
```