COMP 210, FALL 2000
Lecture 26: Accumulators, part I

Reminders:

1. Missionaries & Cannibals is on the web. Due 12 April 2000. Go to lab today; it relates to solving this homework problem, as well as debugging.
2. Exam 2 is graded. Range was 33 to 100; average was 78.6 with a standard deviation of 14. (More information was shown in class. That's the bonus for attending.)

Review

1. We developed an information structure to represent the route system of Jet Set Airlines and write a program that would determine if a route exists from some starting city to a final destination. (**find-flights**). The code was rather involved–-a program that extracts a list of destinations from the route map data structure and another that processes a list of cities and recursively invokes find-flights on every city in the list.

   This exemplies a solution technique called backtracking.

Generalizing Find-Flights

The program **find-flights** works 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 …)

Why 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 –-

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 **new-routes**, 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.

There is one distasteful aspect of the way that we used **visited** to fix **find-flights**. To cure what was, in essence, a design flaw in the program, we changed its contract by adding an extra parameter. Can we avoid this problem? Certainly. We can make the new version of find-flights be a helper function with a new name, such as **fixed-find-flights**, and rewrite **find-flights** so that is simply invokes **fixed-find-flights** with the extra argument. This arrangement has the advantage that it allows us to ensure the correct initial value for the parameter **visited**.

This is a great example of a place where we can use local to hide the entire mess. We can rewrite **find-flights** so that it defines **fixed-find-flights**, **direct-cities**, and **find-flights-for-list** inside a local. What parameters can we ellide at that point? (At least the route map and the destination city!)

Have a good break.