Section Notes 4

CS51—Spring 2009

Week of February 23, 2009

Outline

1. Higher order functions
   (a) lambda!
   (b) Passing to functions
   (c) Returning from functions
   (d) Understanding evaluation for lambda
2. Games!
   (a) Game trees
   (b) Strategies
   (c) Game values
3. Graphs
   (a) Probabilitic transitions
   (b) Random walks

1 Goals for today:

At the end of today's section, you should be able to do the following:

1. Write inline lambda functions for use in filter and map and fold.
2. Manually evaluate a game tree and find the optimal strategy.
3. Manually simulate a random walk on a graph
4. Write a function that returns a strategy for a number picking game.

This week's section will focus on higher order functions and games. As we started to see in class last week, and as we'll keep seeing for the rest of the semester, higher order functions are incredibly useful tool that help make our code more concise and more elegant. Playing games includes very natural applications of higher order functions, because a strategy in a game is simply a function that determines what to do given some game state.

2 Higher order functions

Defining a lambda:

```
(lambda (x) (string-append x " -- ARRR!"))
```

What does a lambda expression evaluate to?

Passing it to a function:

```
(map (lambda (x) (string-append x " -- ARRR!")) my-speech)
```

Q1: How can we write a single call that will append "– ARRR!" to the end of every string in my-speech, and then concatenate all of them together.

Using a lambda to *partially apply* a predicate:

```
(filter (lambda (x) (clique? graph x)) possible-triangles)
```

Q2: Write `dup-dup`, which takes a list and duplicates each element: `(dup-dup '(1 2 3))` should evaluate to `'(1 1 2 2 3 3)`.

Q3: Write `(num-occurs el lst)` using foldr

Q4: Challenge! Write num-occurs-deep using foldr. You can have your function return 1 if el = lst.

Making a function that returns a function:

```
(define (suffixize suffix)
    (lambda (text)
       (map (lambda (s) (string-append s suffix)) text)))
```

And using it:

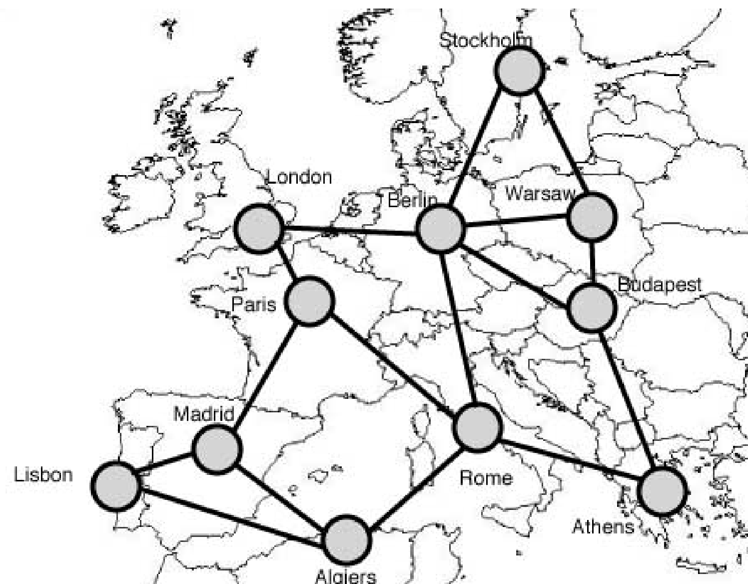
```
((suffixize "- HAR HAR") my-speech)
```

What if I didn't want to have a separate define for suffixize? Here's a way to suffixize a whole bunch of speeches all at once.

```
(map (lambda (speech)
        (map (lambda (s) (string-append s "- HAR HAR")) speech))
 my-speeches)
```

Confusing, eh? To figure out what's going on, start from the inside and work out. Note the helpful variable names.

3 Where in the world?



Europe! Adventures! Strategery!

3.1 Exploring a graph

Our hero, Carmen, starts out in Rome, and wants to go explore Europe by bus and airplane. What are some ways she might go about it?

One way to approach this problem is to come up with an *exploration strategy*: given a location, decide where to go next. What would some possible strategies be?

- Pick a city where you haven't been to yet...
- Pick a city with the cheapest bus/plane ticket...
- If there's a city with really fun museums nearby, go there. Otherwise explore somewhere new.
- She doesn't like being too predictable. Pick randomly!

How would we write the random strategy function in Scheme? [1]

[1] Note that this is very similar to a markov chain babbler–there, we do the same kind of random walk, but on a very different graph–sequences of words instead of cities.

```
; Takes a graph and a current location, returns
; the node to go to next.  It must be a neighbor
; of location.
(define (strategy-bus graph location)
   (pick-random (graph-neighbors location)))
```

What if we could fly to any city instead of just taking a bus or train to a nearby city? Could combine strategies:

```
; Takes a graph and a current location, returns
; the node to go to next.  The node will be randomly chosen
; from all the vertices in the graph.
(define (strategy-fly graph location)
   (pick-random (graph-vertices location)))


; Takes a graph and a current location, returns
; the node to go next.  With probability .75, it will be
; a neighbor of location (going by bus/train), otherwise, a random
; node in the graph (flying).

(define (strategy-bus-and-fly graph location)
    (if (< 0.75 (random-float 1.0))
        (strategy-bus graph location)
        (strategy-fly graph location)))
```

What if we want to change the relative probabilities? Aha–higher order functions to the rescue–we can write a function that will return a strategy with the desired mix:

```
; Return a new strategy that follows s1 with probability p, and otherwise
; follows s2
(define (combine-strategies s1 s2 p)
  (lambda (graph location)
      (if (< p (random-float 1.0))
          (s1 graph location)
          (s2 graph location))))
```

To take the bus with probability 0.4, we could now do the following:

```
(combine-strategies stategy-bus strategy-fly 0.4)
```

3.2 The chase

Now imagine that Detective *Fry Tening* comes chasing after Carmen. She's not one to go off and hide, and still wants to go exploring, but she'd really like to avoid any unpleasant encounters. Now Carmen's strategy involves balancing between exploring new places, and trying to evade pursuit.

We'll model the situation sequentially–Carmen gets to move to a neighboring city, then Fry gets to move, etc. If we assume that Carmen knows where her pursuer is, what might be a good strategy for her to follow?

- If Fry Tening is nearby, head away from him. Otherwise, ignore him and just go to the place that seems most interesting.

- This sort of strategy only considers what's likely to happen at the next time step.

Now let's think about what happens if Carmen knows the official search procedures, and can predict exactly what Mr Tening will do in any situation? How can she use this to her advantage?

Well–when considering her move, she can predict what will happen next. But once she knows that, she can plan the move she could make after that, and so on. If she's trying to maximize the entertainment she gets from her travels, and she knows Fry's strategy, she can compute what her own strategy should be.

Let's think about this in code now. Fry Tening's strategy is just like Carmen's–a function which, given the graph and his current location (and maybe some info about where Carmen is), decides where to go.

```
(define (fry-tening-strategy graph fry-location carmen-location)
   (.... returns a move ....)
```

If Carmen knows this strategy, how should be able to call it in her own strategy function, so her strategy function now takes the fry-tening-stategy as a parameter:

```
(define (carmen-strategy graph
         carmen-location fry-location
         fry-tening-strategy)
   (pick the move that maximizes the total fun had, assuming that
         Fry will use fry-tening-strategy on his next move,
         that Carmen will use her best strategy after that, etc...))
```

3.3 Game trees and tree games

That's enough code for now. Let's talk about the Tree Game from PS3. We're given a tree that has numbers at the nodes, and at each step, the current player gets the value at the current node, and then gets to pick the child node to go to next. Then the other player goes.

Here's a tree of depth 2:

```
      4

   3     2
```

Q: What's the best strategy for player one?

Here's one of depth 3:

```
        1

     5     4

   6  7   1   5
```

Q: What's the best strategy for player one? What would a *greedy* strategy do?

Here's one of depth 4:

```
                1

        5               4

     6       7       1      5

   4   3   1  99   2   1   5   3
```

Q: What's the best strategy for player one? What would a *greedy* strategy do? What if you knew player two's strategy?

Here's one of depth n:

```
        4

     3      2

  1    2    3    7

        .
        .
        .
```

Q: Now what's the best strategy?

If you're interested in learning more about these kinds of things, check out CS181, and maybe economic game theory.

4 Answers

Q1. Use foldr!

```
(foldr (lambda (el r) (string-append el "--ARRR! " r)) "" my-speech)
```

Q2. Use foldr again!

```
(define (dup-dup l)
  (foldr (lambda (el r) (cons el (cons el r))) '() l))
```

Q3. num-occurs using foldr:

```
(define (num-occurs el lst)
   (foldr (lambda (x r) (+ r (if (equal? x el) 1 0))) 0 lst))
```

Q4. Magic!

```
(define (num-occurs-deep el x)
 (cond [(equal? el x) 1]
       [(pair? x) (foldr (lambda (x sum) (+ sum (num-occurs-deep el x))) 0 x)]
       [else 0]))
```