Hints for Exercise 4: Recursion

EECS 111, Winter 2017

Due Wednesday, Jan 8th by 11:59pm

Question 1: `multiply-list`

For many list problems, this one included, the base case is when the list is `empty`, which you can check using `(empty? <List>)`.

Remember that you can divide a list into a head and tail using `first` and `rest`, respectively. `first` will return the first element, and `rest` returns a new, shorter list. So for instance,

```
(define lon (list 1 2 3))  ; `lon` = "list of numbers"
(first lon)  ; 1
(rest lon)  ; (list 2 3)

(first (list 1))  ; 1
(rest (list 1))  ; empty
```

Look back at the slides for definitions of recursive procedures if you're having trouble thinking of how to approach this problem.

Question 2: `iterated-overlay`

Recall that if `some-generator` is a function `number -> image`, then

```
(iterated-overlay some-generator 3)
```

is equivalent to saying

```
(overlay (some-generator 0)
         (some-generator 1)
         (some-generator 2))
```

which, in turn, is equivalent to saying

```
(overlay (some-generator 0)
         (overlay (some-generator 1)
                  (some-generator 2)))
```

This is a bit hard to read with the nesting, but all we're doing here is saying

```
(overlay (some-generator 0)  ; top image
         <BottomImage>)

;; where <BottomImage>, in turn, is
(overlay (some-generator 1)  ; top image
         (some-generator 2))  ; bottom image
```

Question 3: `iterated-any`

This is just an abstraction of `iterated-overlay`. It might be helpful to go back to Exercise 2, and think about the process you used to abstract between the last two bullseye functions. You should follow the same workflow here.

Going from `iterated-overlay` to `iterated-any` is a lot like going from `count-odd?` to `count` in this week's tutorial.

Recall that `iterated-overlay`, `iterated-beside`, `iterated-above`, etc. all perform the same general task: they call a generator some number of times, and then *combine* the resulting images into a single return value. The name of each function specifies the *combiner* used:

```
;; square-gen : number -> image
(define (square-gen n)
  (square (* n 10)
          "solid"
          (color (* n 50) 0 0)))

(check-expect (iterated-beside square-gen 3)
              ;; equivalent to using `beside` to combine
              ;; all of the iteration results
              (beside (square-gen 0)
                      (square-gen 1)
                      (square-gen 2)))
```

Questions 4 through 6: `count-tree`, `sum-tree`, `max-tree`

All three of these functions have very similar structures. The data definition for a `ListTree` provides hints regarding the structure of the function. Once again:

```
;; A ListTree is one of:
;; - <Number>            <-- this is your base case
;; - (list <ListTree>)   <-- this is your recursive case
```

Remember that you can use the predicates `number?` and `list?` to check if something is a number or a list, respectively.

It might help to first write each of these functions for a single-layer list (i.e. no nested list hell), just to make sure you understand the general concept of recursively counting, summing, computing the max, etc.

Namely, suppose we are writing a recursive function `<DataType> -> <ReturnType>`. Recall that the general pattern for a recursive step looks like this:

```
;; <Combiner> is any function of the form
;; <ReturnType>, <ReturnType> -> <ReturnType>
(<Combiner> ...something with the first part of <DataType>...
            ...recursive call with the rest of <DataType>...)
```

Since we're working primarily with numbers here, some combiners to consider are the arithmetic operations `+`, `*`, `-`, etc. as well as the function `max`. Look up the documentation for `max` if you don't know what it does.

Question 7: `depth-tree`

This one is hard! Previous questions have asked you to implement functions using *either* `map`, `foldr`, etc. *or* recursion. It's true that `map` and co. are implemented using recursion under the hood (they *abstract* over the process of manually recurring on a list). However, this question is designed to show you that higher-order

functions and recursion can work together – in fact, many types of problems like this one are much easier if you use both.

To break this question down, it's crucial to understand (1) how the problem decomposes, and (2) how each of the higher-order functions is used.

Let's start with decomposing the problem. Returning to our trusty data definition for `ListTree`:

```
;; A ListTree is one of:
;; - <Number>            <-- this is your base case
;; - (list <ListTree>)   <-- this is your recursive case
```

So for any given `ListTree`, we have the following cases:

- It's just a single number. (What should we return?)
- It's a list of numbers. (What should we return?)
- It's a list of (a list of numbers)...

Now things start to get tricky. But whenever you see a recursive data structure like this, it's a huge sign you should be using recursion to solve the problem, so **there's probably going to be a recursive call in here somewhere**.

The main difference is that before, "paring down" our data divided things neatly into two parts.

- For a number, there's `n` and `(- n 1)`.
- For a one-dimensional list, there's `(first l)` and `(rest l)`.

But for a `TreeList`, we can't just break things up this way. Consider the following example:

```
(define my-tree
  (list (list 1 2)
        (list 1 (list 2))))
```

Right off the bat, since `my-tree` is a list, we know the depth is going to be at least 1. The question is, how do we account for the children?

- `(first my-tree)` gives us `(list 1 2)`, which itself has a depth of 1.
- `(rest my-tree)` gives us `(list 1 (list 2))`, which itself has a depth of 2.

Clearly, we want to add the *larger* of these two depths. What function might we use to compute this value?

But we're not quite done! Suppose we have even more items:

```
(define my-tree
  (list (list 1 2)
        (list 1
              (list 2))
        ;; Three new friends!
        1
        (list 1
              (list 2
                    (list 3)))
        (list 1)))
```

Does your answer still work for an *arbitrary* number of list items? We need some way of retrieving the largest depth out of an *arbitrary number* of list elements. Actually, we need some way to get the depth of all those elements to begin with. This leaves us with two questions:

- Which higher order functions are good at applying some function to all the items in a list?
- which higher order functions are good at aggregating a list of results into a single result?

Appendix 1: Iterative Recursion

For instance,

```
;; fact : number -> number
;; regular factorial with ordinary recursion
(define (fact n)
  (if (= n 0)
      1
      ;; Recursive Step
      (*  ; combiner
        n           ; current data
        (- n 1))))  ; rest of our data

(fact 5)  ; call our function

;; fact/iter : number, number -> number
;; factorial using iterative recursion
(define (fact/iter n acc)  ; [1]
  (if (= n 0)
      acc  ; [2]
      ;; Recursive Step
      (fact/iter  ; [3]
       (- n 1)
       (* n acc))))

(fact/iter 5 1)  ; 1 is our starting accumulator
```

There are exactly three differences to note here, labeled as comments in the code above:

1. The addition of a **new argument** for the accumulator (the partial product, in this case). This is reflected in:
    1. An **updated function header**, `(define (fact/iter n acc) ...)`
    2. An **updated recursive call**, which now takes *two* arguments: the decremented number, and the new accumulator.
2. The base case now **returns the accumulator** rather than 1. That's because the base case signals we're done recurring, so the "partial product" is really the "complete product" at that point, and we can just return whatever value it's accrued.
3. All of our computations have moved **inside the recursive call**. To be clear,

    ```
    ;; ordinary recursion
    (*                 (<Combiner>
      n                  <Current>
      (fact (- n 1)))    (<RecursiveCall> <Rest>))

    ;; iterative recursion
    (fact/iter         (<RecursiveCall>
      (- n 1)            <Rest>
      (* n acc))         (<Combiner> <Current> <OldAcc>))
    ```

    By making our combiner work on the accumulator, we can progress directly into the next recursion, without waiting for the recursive result to finish expanding. In other words, we don't need to leave

behind an expanding trail while we wait for the base case to hit:

```
;; 5! in progress, without accumulators
(* 5
   (* 4
      (* 3  ; this trail gets big really quickly
         (fact (- 3 1)))))

;; 5! in progress, with accumulators
(fact/iter (- 3 1)
           (* 3 20))  ; 20 is the accumulator 5 * 4
```

There's one last caveat, though. Notice that instead of calling our function with

```
(fact 5)
```

we now have to call it with 1 as our starting accumulator:

```
(fact 5 1)
```

This is really annoying and also bad practice from a software design standpoint: if the starting accumulator is always going to be 1, we should just write that into our software somehow. Leaving it to user discretion only increases the risk of something going wrong (e.g. suppose someone accidentally writes 0, and now `fact/iter` always returns 0 for every input).

So, the **final step in converting a function to iterative recursion** is to provide a **wrapper** of some kind around your main recursive function. This wrapper should have the same signature as your main function, *without the accumulator*, and it should kick off the recursive process by calling your main function with the starting accumulator value.

You can accomplish this in one of two ways: by writing a separate wrapper function, or using `local` as a wrapper.

1. Writing a separate wrapper function

```
;; 1. Rename our old function to fact/iter-helper

;; fact/iter-helper : number, number -> number
;; factorial using iterative recursion
(define (fact/iter-helper n acc)
  (if (= n 0)
      acc
      ;; Recursive Step
      (fact/iter-helper
       (- n 1)
       (* n acc))))

;; 2. Make fact/iter a wrapper around the helper

;; fact/iter : number -> number
(define (fact/iter n)  ; notice acc is gone again
  (fact/iter-helper n 1))

;; Now we can use our iterative solution like before!
(fact/iter 5)
```

2. Using a `local` expression

```
;; Rename the original function to fact/iter-helper, and
;; dump it in the definitions part of a `local`.
```

```
;; fact/iter : number -> number
(define (fact/iter n)
  (local [;; fact/iter-helper : number, number -> number
          ;; factorial using iterative recursion
          (define (fact/iter-helper n acc)
            (if (= n 0)
                acc
                ;; Recursive Step
                (fact/iter-helper
                 (- n 1)
                 (* n acc))))]

    ;; Now that we've locally defined our helper, we can
    ;; just call it in the body of the `local` like normal.
    (fact/iter-helper n 1)))
```

So, to summarize, there are four steps involved with the transformation from ordinary recursion to iterative recursion:

1. Add the accumulator argument to the function definition, and update the recursive call
2. Return the accumulator in the base case
3. Restructure the recursive step so that the recursive call is on the outside. Remember to pass in a new accumulator, which should be the result of combining the current data with the old accumulator.
4. Finally, wrap everything up so the function has the same signature as before you added the new argument. This means using a `local` or writing a helper function; both are perfectly reasonable strategies.