

Exercise 4: Great Times with Recursion

EECS 111, Winter 2017

Due Friday, Feb 10th by 11:59pm

Introduction

In this assignment, you will practice writing recursive functions using **ordinary recursion** (i.e. without accumulators), and **iterative recursion** (i.e. with accumulators).

For each problem, complete the following process:

1. Write the **problem statement** for the function.

```
;; compute the factorial of the given number
```

2. Write the **type signature** for the function.

```
;; fact : number -> number
```

3. Determine the **base case** for your recursion. This is the simplest case, in which the answer is trivial.

```
;; (fact 0) => 1
```

4. Write a **test case** (using `check-expect`) for the base case.

```
(check-expect (fact 0) 1)
```

5. Write multiple **test cases** for the recursive case. Remember to test for all *edge cases*, or situations where your function might break.

```
;; using Racket's * function to avoid having to do math
(check-expect (fact 1) 1)
(check-expect (fact 5)
              (* 5 4 3 2 1))
```

6. Finally, **write the code itself**. Remember that although the specifics will vary, you generally want to have a pattern like this:

```
(if <BaseCase>          ; (= n 0)
    <BaseCaseResult>   ; 1
    <RecursiveStep>)  ; (* n (fact (- n 1)))
```

If you have multiple cases, remember that the `<RecursiveStep>` can be any Racket expression, including another `if` condition. Alternatively, if your function is more complicated, you may want to use `cond` instead:

```
(cond
  [<BaseCase> <BaseCaseResult>] ; [(= n 0) 1]
  [<AnotherCase> <AnotherCaseResult>]
  ;; ...
  [<else <RecursiveStep>]]      ; [else (* n (fact (- n 1)))]
```

where the `<RecursiveStep>` often takes the form

```
(<Combiner> <FirstPartOfData>
  (<RecursiveCall> <RestOfData>))
```

Whether you use `if` or `cond` is purely a matter of style, but using `cond` can be easier to read for multiple levels of logical branching.

VERY VERY IMPORTANT: Unless otherwise stated, you may *not* use `map`, `filter`, `foldl`, `foldr`, `apply`, or `build-list`. You will not receive credit for solutions using these functions (again, unless explicitly stated otherwise).

Part 1: Ordinary Recursion

For this section, use **ordinary recursion** (without an accumulator) for all your functions. This is the simplest form of recursion, and the first version you learned in class.

Question 1: multiply-list

Write a recursive procedure, `multiply-list`, to compute the product of a list of numbers.

```
;; multiply-list : (listof number) -> number
```

For example, `(multiply-list (list 1 2 3 4))` should return 24.

- **Be sure to follow the steps listed in the introduction.** You must include your own test cases, signatures, and purpose statements.
- Note that the product of the empty list is defined to be 1, not 0.¹

Question 2: my-iterated-overlay

It's baaaaack! :^) Write a recursive function, `my-iterated-overlay`, that behaves exactly the same way as `iterated-overlay`.

```
;; my-iterated-overlay : (number -> image), number -> image
```

Here are some pointers and hints:

- You will need to return a blank image if the number of iterations is zero. You can do this by returning `empty-image` (no parentheses needed – it's a constant, not a function).
- Remember that calling `(iterated-overlay proc 5)` calls `proc` with the sequence 0, 1, 2, 3, 4. More generally, calling `(iterated-overlay proc count)` will generate the sequence 0, 1, ..., `count - 1`.
- Remember that `(overlay a b)` puts `a` on top of `b`, and `(overlay b a)` puts `b` on top of `a`. You need to make sure the iterations stack appropriately. For instance, if `gen` is a generator function, the result of calling `(my-iterated-overlay gen 5)` should place `(proc 0)` on top, then `(proc 1)`, and so on, with `(proc 4)` at the bottom.

Calling your function like this:

```
(my-iterated-overlay (lambda (n) (square (* n 10)
                                         "solid"
                                         ;; make each iteration progressively lighter
```

¹If you're into math, that's because it's usually most useful to define the sum or product of an empty list as being the "identity element" for the operation you're performing (adding or multiplying). The identity element is the number you can add/multiply another number by without changing the other number. So 0 is the identity element of + because $x = x + 0$. But 1 is the identity element of multiplication because $x = x * 1$. Wow I just really love the properties of real numbers

```

                    (color (* n 50) 0 0)))
  5) ; number of iterations

```

should return a result like this:



- Remember, you cannot use `foldl`, `foldr`, `apply`, etc.

Question 3: iterated-any

Now you will *abstract* your code from `iterated-overlay` just like we did in Exercise 2. Write a recursive function, `iterated-any`, that takes an arbitrary *combiner* (a function with the signature `image, image -> image`), a generator, and a number of iterations, and combines the results using the specified combiner.

```

;; iterated-any : (image, image -> image), (number -> image), number -> image
;;
;;
;;
                    <Combiner>          <Generator>      <Count>

```

You can test your function against your implementation of `my-iterated-overlay` from the previous question.

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)))

```

Namely, passing in `overlay` as our combiner:

```
(iterated-any overlay <Generator> <Count>)
```

should be equivalent to just calling `my-iterated-overlay`:

```
(my-iterated-overlay <Generator> <Count>)
```

which, in turn, should be equivalent to calling the original:

```
(iterated-overlay <Generator> <Count>)
```

The same holds for `iterated-beside`, `iterated-above`, etc.

Interlude: ListTree

The next few questions will deal with trees of numbers.

A `ListTree` is a recursive data structure with the following definition:

```
;; A ListTree is one of:  
;; - <Integer>  
;; - (list <ListTree>)
```

Note that unlike lists you've seen before, we have made the (somewhat arbitrary) stipulation that a `ListTree` is never `empty`. The base case is when the `ListTree` is just a plain old `number`.

Here are some examples of `ListTree`:

- 1
- (list 1 2)
- (list (list 1)
2
(list 1 4 5))
- (list 1
(list 2 1)
1
1)
- (list 1
(list 2
(list 3 3)
(list 2
(list 4 5))))

Question 4: count-tree

Write a recursive function, `count-tree`, that *counts* the numbers in a `ListTree`.

```
;; count-tree : ListTree -> number
```

- If a number appears twice, count both occurrences. That is,
(count-tree (list (list 1)
2
(list 1 4 7)))
;; => 5
- Remember that a `ListTree` can be an ordinary integer too, so
(count-tree (list 1))
;; => 1

(count-tree 1)
;; => 1

Question 5: sum-tree

Write a recursive function, `sum-tree`, that *sums* the numbers in a `ListTree`.

```
;; sum-tree : ListTree -> number
```

For instance,

```
(sum-tree (list (list 1)
                2
                (list 1 4 7)))
;; => 15
```

Remember to test for edge cases.

Question 6: max-tree

Write a recursive function, `max-tree`, that *computes the largest* number in a `ListTree`.

```
;; max-tree : ListTree -> number
```

For instance,

```
(max-tree (list (list 1)
                7
                (list 1 4 2)))
;; => 7
```

You may find it helpful to use the function `max`, which takes two or more numbers and returns the largest.

Again, remember to test for edge cases. `ListTrees` may contain *any integer*.

Question 7: depth-tree

Write a recursive function, `depth-tree`, that returns the number of levels of nesting in a `ListTree`.

```
;; depth-tree : ListTree -> number
```

For instance,

```
(depth-tree 1)
;; => 0, it's not even a list
```

```
(depth-tree (list 1 2))
;; => 1
```

```
(depth-tree (list (list 1)
                  7
                  (list 1 4 2)))
;; => 2
```

Whereas the previous three functions you wrote were very similar, `depth-tree` uses a different pattern, so you can't reuse your code as easily.

For this question, you MAY use `foldl`, `foldr`, `map`, and `filter`, but you don't need to.

This one is hard! Think carefully and walk through some cases on paper if you're having trouble.

Part 2: Iterative Recursion

To wrap up the assignment, you'll tweak a few of your functions from Part 1 to use **iterative recursion**. Recall that iterative recursion differs from ordinary recursion by the use of an **accumulator** argument, which represents the *partial result* at the current point in time.

Question 8: multiply-list/iter

Rewrite `multiply-list` as an iterative recursive function, `multiply-list/iter`. The functions should behave identically and have the same signature, but `multiply-list/iter` must use an accumulator,

```
;; multiply-list/iter : (listof number) -> number
```

Note that the iterative version uses the same signature as the ordinary recursive version. However, you *must* use an accumulator argument, or you will lose credit for this function.

So, you will need to define a separate function that uses an accumulator, and have `multiply-list/iter` call that function. You can do this using an entirely separate helper function, or a local expression. See the slides or reading for details.

Question 9: iterated-overlay/iter

Rewrite `my-iterated-overlay` as an iterative recursive function, `my-iterated-overlay/iter`. I'm really sorry the naming ended up being so unfortunate, it really wasn't my intention. Once again, this function should behave identically to `my-iterated-overlay` and have the same signature, but it must use an accumulator.

```
;; iterated-overlay/iter : (number -> image), number -> image
```

Question 10: iterated-any/iter

Same deal here. Rewrite `iterated-any` to use an accumulator.

```
;; iterated-any : (image, image -> image), (number -> image), number -> image  
;;               <Combiner>           <Generator>      <Count>
```

Turning It In

Make sure you've followed the process outlined in the introduction for every function, and that you've thoroughly tested your functions for all possible edge cases.

Double check that you are not using `map`, `filter`, `foldl`, `foldr`, `build-list`, `apply`, or the original `cs111/iterated` library in any of your functions except for Question 7, `depth-tree`.

When you're happy with what you've got, upload everything to the handin server as usual.