Tutorial 4: Recursion

EECS-111 Fall 2015

Problem 1

Write the **factorial** function $n!$ as a recursive procedure. The factorial of a number is the product of all the numbers from one to that number:

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times n$$

So 1! = 1, 2! = 2, 3! = 6, 4! = 24, 5! = 120, and so on.

Problem 2

Now write **factorial** as an iterative recursion.

Problem 3

Now write a recursive procedure, **reverse-list**, that takes a list as input and returns the list with its elements in reverse order. So, for example, (reverse-list (list 1 2 3)) should return (list 3 2 1).

For this problem, you'll need to use the **cons** procedure, which takes an object and a list, and returns a new list that's just like the original list, only with the object at the beginning. So (cons 1 (list 2 3)) = (list 1 2 3). And (cons 0 (cons 1 (list 2 3))) = (list 0 1 2 3). It's sort of the opposite of rest; rest shortens a list by one element, while cons lengthens it by one element. **Do not use append**.

Hint: use an iterative recursion. Your code will look like:

```
(define (reverse-list list)
  (local [(define (do-the-work  remaining reversed)
            fill this in)]
    (do-the-work  list  '())))
```

Problem 4

Write a recursive procedure **flatten**, that takes a list that may have lists inside it (and which may have lists inside them, etc.) and returns a list of all the elements of all the lists, in order, without any sublists. Here's what we mean:

- (flatten 1) should return (list 1)
- (flatten (list 1 2 3)) should return (list 1 2 3)
- (flatten (list 1 (list 2 3 (list 4)) 5)) should return (list 1 2 3 4 5)

So the definition of the flattening of a list is:

- If x isn't a list, then (flatten x) = (list x)
- If x is a list, then (flatten x) = the flattenings of all x's elements, merged into one list.

Hint: It's not the most efficient way to solve it, but you can do it in 5 lines using map, apply, and append. There's a more efficient way to do it using a procedure called append-map, but it's not available in Racket's student language :-(

Problem 5

Here's a procedure:

```
(define (filter-map predicate proc list)
  (map proc
       (filter predicate list)))
```

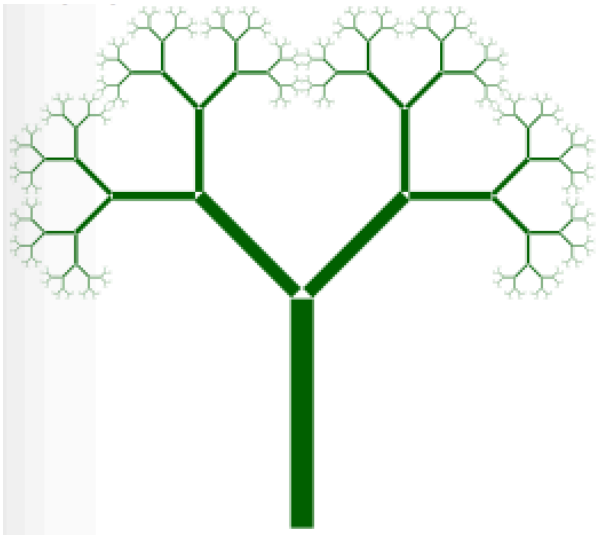
This is just like map, only it prefilters the list so that map only runs on elements of the input list for which predicate returns true. So (filter-map odd? square (list 1 2 3 4 5)) returns (list 1 9 25) because 2 and 4 get rejected by the odd? Predicate.

The code above isn't a particularly efficient way of implementing filter-map because the filter call makes a list and then map makes a new list from it. So the intermediate list is sort of a waste. Write filter-map directly as a recursion, so that there's no wasted intermediate list.

Hint: you will need to use cons. Do not use append.

Problem 6

Write a recursive procedure, **tree**, that makes a recursive image something like this:



Your image doesn't need to look exactly like this one; just experiment with making recursive pictures like this. Have fun.

Note: remember that you'll need to add **(require 2htdp/image)** to your file to get access to the graphics procedures.

Hint: the idea is that a tree is a stick with two smaller trees sticking out of it at angles. So you're going to:

- Make the stick using **rectangle**
- Recurse to make the subtrees, that is call **tree**
- Shrink the subtrees using the **scale** procedure
- **Rotate** them and add them to the stick.

So your code is going to look something like this:

(above (beside (rotate *angle* (scale *factor recur*))
(rotate –*angle* (scale *factor recur*))
(rectangle … *whatever arguments you want*…))

Pick whatever *angle* you want; I used 45 degrees. For factor, choose some number less than 1, so that the subtrees are smaller than the original stick. And for rectangle, use whatever arguments you want, but you want to be sure the rectangle is narrow and tall.

For *recur*, put in a recursive call. You'll need some way of keeping the program from recursing forever, so write your tree procedure to take a number as an argument. If the number is 0, have it return **empty-image** (that's a built-in magic variable that has a blank image in it). Then you can choose how detailed your want your tree to be by choosing whether you say (tree 2) or (tree 3) or (tree 10).

Here are a series of calls to my version of tree with arguments 1 through 10:

Feel free to experiment. You can add little circles at the ends, for example to make something that looks vaguely like leaves. There's no specific right answer we're looking for on this one.

Problem 7

Write a procedure, arg-max, that takes a function and a list and returns the element of the list that produces the highest score from the function. As a contrived example, the sine of the first few integers are:

- sin 0 = 0
- sin 1 = 0.8414709848078965
- sin 2= 0.9092974268256817
- sin 3 =0.1411200080598672

So if we were to call (arg-max sin (list 1 2 3)), we should get the result 2, because sin 2 is greater than sin 0, sin 1, or sin 3.

Hint: this is easier to write as an iterative recursion. As usual, you write two procedures, arg-max, and the procedure that does the actual work, so that your code looks something like this:

```
(define (arg-max function list)
   (local [(define (do-the-work  remaining  best-so-far  best-score-so-far)
             fill this in)]
      (do-the-work list
                   (first list)
                   (function (first list))))
```