Homework 3: Recursion
Due: 11:59 PM, Sep 25, 2018

Contents

1 Factorial 3
2 Fibonacci 4
3 Odds Only 5
4 Increment All 6
5 Frequency 6
6 Sublist 7
   6.1 Take .................................................. 7
   6.2 Drop .................................................. 7
   6.3 Putting it Together .................................. 8

Objectives

By the end of this homework, you will be able to:

1. Visualize recursive calls in diagram form
2. Use recursion to filter out a list
3. Use recursion to apply the same procedure to every element of a list

How to Hand In

To hand in your solutions to these problems, you must store them in appropriately-named files. Each should be named for the corresponding problem, as follows:

- factorial-table.txt
- fibonacci-table.txt
- fibonacci-diagram.png (or .pdf, .jpg, .gif)
- odds-only.rkt
For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension .rkt.

For this and every future assignment, you should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

```
README.txt:
B01234567
There's nothing to say except that I'm turning in some files plus this README the way the instructions say that I should.
```

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you choose to also store these files on department machines, all your solution files should reside in your ~/course/cs0170/homeworks/hw03 directory.

### Design Recipe

In last week’s homework, we introduced you to the Design Recipe. For each procedure you wrote, you wrote a procedure specification, listing the input and output of a procedure.

This week, we will add two other pieces to the Design Recipe — recursion diagrams and test cases.

Whenever we ask you to “write a procedure,” you are expected to carry out each and every step of the design recipe. That is, you are to:

- Write a procedure specification (input/output)
- Write recursion diagrams for each possible behavior, including recursive input/output. Note, if the procedure is not recursive, you may leave this out.
- Write test cases, testing both base and recursive cases.
- Write the procedure

Additionally, you are to follow the Racket/Scheme style guide on our website.

You should not test undefined behavior, nor should your goal be to write as many tests as possible. Instead, you should write high quality tests that differentiate good and bad implementations. To accomplish this, you should test both base cases and recursive cases, as well as any special cases you can think of.

To receive full credit, you must also define all procedures to take in arguments in the same order as described in the problem.

We’ve included an example below:
;; input: alon, a list of numbers
;; output: the sum of all numbers in alon

;; recursion diagrams
;; input: (quote (5))
;;   recursive input: (quote ())
;;   recursive output: 0
;; output: 5

;; input: (quote (1 2 3))
;;   recursive input: (quote (2 3))
;;   recursive output: 5
;; output: 6

(define sum
  (lambda (alon)
    (cond
      ((empty? alon) 0)
      (#true (+ (first alon) (sum (rest alon)))))))

;; test cases
(check-expect (sum empty) 0)
(check-expect (sum (quote (1))) 1)
(check-expect (sum (quote (5 4 3))) 12)

Problems

1 Factorial

In class, we introduced harmonic numbers and showed we can calculate the $n^{th}$ harmonic number using recursion. In this problem, we’ll explore the $n^{th}$ factorial. As you may recall from a mathematics class, we write the $n^{th}$ factorial as $n!$

It’s equal to:

$$F(n) = \begin{cases} 
1 & \text{if } n = 0 \\
\times F(n-1) & \text{if } n > 0 
\end{cases}$$

As you can see, we have both a base case and a recursive case. To implement this in Scheme/Racket, we would create a procedure like this:

;; input: n, a non-negative integer
;; output the n`th factorial

;; recursion diagrams
;; input: n = 1
;;   recursive input: 0
;;   recursive output: 1
;; output: n = 1

;; input: n = 4
Task: In a text file called factorial-table.txt, construct a table like the following, filling in the values for each factorial in the below table:

<table>
<thead>
<tr>
<th>n</th>
<th>F(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>

In the next problem, we ask you to draw another type of diagram where each invocation is a node (circle) in a graph with an arrow going from one invocation to its recursive invocation.

To demonstrate, we’ve included a diagram for our factorial procedure.
2 Fibonacci

In the harmonic number example presented in class and the factorial problem in the previous question, we’ve written recursive procedures with one recursive call. That’s not always the case. Next, we’re going to look at Fibonacci numbers. The \( n \)th Fibonacci number is defined as:

\[
F(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F(n-1) + F(n-2) & \text{if } n > 1
\end{cases}
\]

As you can see, we have two base cases, where both return a constant. Our recursive case (when \( n > 1 \)) has two recursive calls. To implement this in Scheme, we would create a procedure like this:

```scheme
;; input: n, a non-negative integer
;; output the n\textsuperscript{th} fibonacci number
;;; recursion diagrams
;;; input: 2
;;; recursive input: 1
;;; recursive output: 1
;;; recursive input: 0
;;; recursive output: 0
;;; output: 1

;;; input: 3
;;; recursive input: 2
;;; recursive output: 1
;;; recursive input: 1
;;; recursive output: 1
;;; output: 2

(define fib
  (lambda (n)
    (cond
      ((= n 0) 0)
      ((= n 1) 1)
      ((> n 1) (+ (fib (- n 1)) (fib (- n 2)))))))
```

**Task:** In a text file called fibonacci-table.txt, construct a table like the following, filling in the values for each fibonacci number in the below table:

<table>
<thead>
<tr>
<th>n</th>
<th>F(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>
Task: Draw another type of diagram where each invocation is a node (circle) in a graph with an arrow going from one invocation to its recursive calls(s). Scan, take a photo of, or draw on the computer the full diagram. Name it fibonacci-diagram.png.

Hint: Here’s a diagram to get you started!

3 Odds Only

We can use builtins that return booleans (these are called *predicates*) to filter lists by removing elements for which the predicate returns *false*.

Task: Write a procedure, `odds-only`, that consumes (i.e., takes as input) a list of integers, `aloi`, and produces (i.e., outputs) another list of integers that contains only the odd numbers in `aloi`, in their input order. Use the predicate `odd?` to determine whether the number is odd or not.

Examples:

```
(odds-only (quote (1 2 3))) => (1 3)
(odds-only (quote (2 18))) => ()
```

Note: Be sure to include recursion diagram(s) (input, recursive input, recursive output, and output) as part of the design recipe. You should include two recursive diagrams that model two distinct cases. Be thoughtful in your choice of inputs so that the two recursive diagrams show distinct behavior.

4 Increment All

Task: Write a procedure, `increment-all`, that takes as input a list of integers, `aloi`, and outputs a new list where each element is the result of adding one to the corresponding number in the old list.

Examples:

```
(increment-all empty)
```
=> ()

(increment-all (quote (1)))
=> (2)

(increment-all (quote 1 2 5))
=> (2 3 6)

(increment-all (quote (-1 5 -3)))
=> (0 6 -2)

Note: Be sure to include recursion diagram(s) (input, recursive input, recursive output, and output) as part of the design recipe. You should include two recursive diagrams that model two distinct cases. Be thoughtful in your choice of inputs so that the two recursive diagrams show distinct behavior.

5 Frequency

Task: Write a procedure, `freq`, that takes as input a list, `L`, a data object `x`, and outputs the number of times that `x` occurs in `L`.

Examples:

(freq (quote ()) 7)
=> 0

(freq (quote (#false #true #false)) #true)
=> 1

(freq (quote (1 2 1 5)) 1)
=> 2

Note: Be sure to include recursion diagram(s) (input, recursive input, recursive output, and output) as part of the design recipe. You should include two recursive diagrams that model two distinct cases. Be thoughtful in your choice of inputs so that the two recursive diagrams show distinct behavior.

6 Sublist

The next problem will be fully coded on the next homework assignment. For this week, we’re asking you to get started by writing the design recipe.

6.1 Take

Task: Write the design recipe — including procedure specification, test cases and recursive diagrams(s) — for a procedure `take` that takes in a non-negative number `n` and a list `L`, and outputs a list consisting of the first `n` elements of `L`. 
Note: You may assume that the input n will be between 0 and the length of L. (inclusive)

Examples:

(take 3 (quote (1 2 3 4)))
=> (1 2 3)

(take 0 (quote (1 2 3 4)))
=> ()

6.2 Drop

Task: Write the design recipe — including procedure specification, test cases and recursive diagrams(s) — for a procedure drop that takes in a non-negative number n and a list L, and outputs a list consisting of all but the first n elements of L.

Note: You may assume that the input n will be between 0 and the length of L. (inclusive)

Examples:

(drop 2 (quote (1 2 3 4)))
=> (3 4)

(drop 0 (quote (1 2 3 4)))
=> (1 2 3 4)

6.3 Putting it Together

Task: Write the design recipe — including procedure specification and test cases — for a procedure sublist that takes in a list L, a non-negative number start, and a non-negative number len. The return value should be the list of elements in L, starting with the one whose index is start and going for len elements.

Note: The index of the first element of a list is 0. Also note that this procedure is not recursive itself, and as such, no recursive examples are required.

Note: As this procedure itself is not recursive, you need not submit a recursive diagram.

Examples:

(sublist (quote (1 2 3 4 5)) 2 3)
=> (3 4 5)

(sublist (quote (1 2 3 4 5)) 1 2)
=> (2 3)

(sublist (quote (1 2 3 4 5)) 0 3)
=> (1 2 3)
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback