Homework 4: More Recursion
Due: 11:59 PM, Oct 2, 2018

Contents

1 Member 2
2 Append 3
3 Pair Count 3
4 Sublist 4
   4.1 Take 4
   4.2 Drop 4
   4.3 Putting it Together 5

Introduction

In this homework, you will continue to learn how to use recursion to solve different problems. As with the previous homework, and for all future homeworks, you are expected to follow our Design Recipe.

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, recursive, and any special cases.
- Write the procedure

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.
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:

- README.txt
- member.rkt
- member-analysis.txt
- append.rkt
- append-analysis.txt
- pair-count.rkt
- sublist.rkt (which includes take, drop, and sublist)

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/hw04 directory.

Problems

1 Member

Task: Write a procedure, my-member?, that takes as input a symbol and a list L of symbols, and outputs #true if the input symbol is an element of the list, and #false otherwise.

Examples:

(my-member? (quote foo) empty)
=> #false

(my-member? (quote foo) (quote (foo)))
=> #true

(my-member? (quote fifteen) (quote (seventeen eight two thirty)))
=> #false
Note: Needless to say, you are not allowed to use the built-in member? procedure in your implementation or testing of my-member?. You are, however, welcome to use member? for all subsequent problems on this and future assignments.

Task: Write a recurrence relation for the number of operations executed by my-member?.

Here’s some help getting started:

“We define the function $f$ as follows. Let $f(n)$ be the maximum number of operations executed by my-member? on an input list of length $n$.”

2 Append

The cons procedure gives us a way of creating a list by attaching a single element to the beginning of another list. But what if we want to form a list from two preexisting lists?

Task: Write a procedure, my-append, that takes as input two lists of integers and outputs another list of integers that contains all the elements of the first list in their input order, followed by all the elements of the second list in their input order.

Examples:

```
(my-append (quote (1 2 3)) (quote (4 5 6)))  
=> (1 2 3 4 5 6)

(my-append (quote ()) (quote (0 0 0)))  
=> (0 0 0)

(my-append (quote ()) (quote ()))  
=> ()
```

Note: You are not allowed to use the built-in append procedure in your implementation or testing of my-append. You are, however, welcome to use append for all subsequent problems on this and future assignments.

Task: Write a recurrence relation for the number of operations executed by my-append. There are different ways to write recurrence relations. The traditional approach would be to take $n$ to be the total input size, i.e. in this case the sum of the lengths. We take take a slightly simpler approach. Here’s some help getting started:

“We define the function $f$ as follows. Let $f(n)$ be the maximum number of operations executed by my-append applied to any pair of lists where the first list has length $n$.”

3 Pair Count

Monopoly Man and Pringles Face play a prank on their brother by going through his precious sock collection! You are tasked with helping him find all of his matching socks.

Monopoly Man and Pringles Face have conveniently created two vertical lines of equal length of socks across the beach. Their brother is only able to compare two socks at a time—one from the left line
with the corresponding sock from the right line. Also, the moment he finds a pair of non-matching
socks, he will be so upset that he won’t be able to continue counting. Help him find out how many
pairs of matching socks there are before he finds an unmatching pair or runs out of socks!

**Task:** Write a procedure, `pair-count`, that takes as input two lists of integers of equal length and
returns the maximum number of consecutive matching pairs (a matching pair is a pair of elements,
one from each list, that have the same position in the list and have the same value).

**Note:** The inputs list must be of equal length, but can be empty.

Examples:

```scheme
(pair-count (quote (5 4 2 1 4)) (quote (5 4 3 1 4)))
=> 2

(pair-count (quote (0 0 1)) (quote (0 0 1)))
=> 3

(pair-count (quote (1 0 1)) (quote (0 0 1)))
=> 0

(pair-count (quote ()) (quote ()))
=> 0
```

## 4 Sublist

Last week, you began writing the design recipe for three procedures: `take`, `drop`, and `sublist`. This week, you will build on that work and implement all three procedures.

Be sure to save all three procedures in a file called `sublist.rkt`, and include a design recipe.

### 4.1 Take

**Task:** Write 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`. Be sure to include the design recipe you wrote last week.

**Note:** You may assume that the input `n` will in the range `[0, (length L)]`. That is, `0 ≤ n ≤ (length L)`.

Examples:

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

(take 0 (quote (1 2 3 4)))
=> ()
```
4.2 Drop

**Task:** Write 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`. Be sure to include the design recipe you wrote last week.

**Note:** You may assume that the input `n` will in the range `[0, (length L)]`. That is, `0 ≤ n ≤ (length L)`.

**Examples:**

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

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

4.3 Putting it Together

**Task:** Write 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 position in the list is `start` and going for `len` elements.

**Note:** For this problem, the position of the first element of a list is 0. You may assume all input is valid. That is, `start` will be a valid index, and there will be at least `len` items at/after the item at index `start`.

**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/csci10170/feedback](http://cs.brown.edu/courses/csci10170/feedback)