Homework 5: Even More Recursion
Due: 11:59 PM, Oct 10, 2018

Contents

1 Sum, Product, and More! 2
2 Flip and Flipper 3
3 Set? 4
4 Set-Equal? 5
5 Analysis 5

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:

- sum-product.rkt
- flip.rkt
- flipper.rkt
- set.rkt
- set-analysis.txt
- set-equal.rkt
- set-equal-analysis.txt
- analysis.txt

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 re-submit your homework, you must re-submit all files. If you choose to also store these files on department machines, all your solution files should reside in your `~/course/cs0170/homeworks/hw05` directory.

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, this is only if the procedure is recursive.
- 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.

Practice

Problems

1. Sum, Product, and More!

Task: Write a procedure `sum3` that takes in three integers, `x`, `y`, and `z` and produces the sum of those three numbers. You may only use the `+` procedure with 2 arguments, but can use nested expressions. Feel free to skip the design recipe (I/O contract, recursion diagrams, test cases).

Task: Write a procedure `product3` that takes in three integers, `x`, `y`, and `z` and produces the product of those three numbers. You may only use the `*` procedure with 2 arguments, but can use nested expressions. Feel free to skip the design recipe (I/O contract, recursion diagrams, test cases).

Task: Write a procedure `sum-or-product` that takes in a symbol `s` that is either `add` or `multiply`, as well as three integers, `x`, `y`, and `z`. The procedure should output the sum of those integers if `s` is equal to `add`, and the product of those integers if `s` is equal to `multiply`. 
Note: You may not use any helper procedures, or call sum3 or product3. You may only reference \( x, y, \) and \( z \) once in the body of your procedure.

Task: Write a general implementation of this, fold3, which takes in a procedure proc, as well as three integers, \( x, y, \) and \( z \). When given \( + \) as proc, it should behave like sum3. When given \( \ast \) as proc, it should behave like product3. It should be generalizable and work for other procedures like max.

Examples:

\[
\begin{align*}
(fold3 + 2 3 4) & \Rightarrow 9 \\
(fold3 * 2 3 4) & \Rightarrow 24 \\
(fold3 \text{ max } 2 5 4) & \Rightarrow 5
\end{align*}
\]

Task: In addition to your regular exhaustive testing of fold3, specifically test fold3 with max and quotient as the input procedure.

2 Flip and Flipper

You were just hired to help determine who murdered Mr. Boddy at Tudor Mansion. The suspects are Miss Scarlet, Professor Plum, Mrs. Peacock, Reverend Mr. Green, Colonel Mustard, Mrs. White, and Dr. Orchid. Your job is to make it easier for investigators to sort through and identify various lists containing the suspects and their respective last known locations. To do this, you will need to write some procedures that manipulate lists of paired data effectively.

Task: Write a procedure flip that takes as input a list of pairs (i.e., two-element lists) of symbols and outputs a new list of pairs, where the pairs are in the same order as the corresponding pairs in the input list, but each pair is in reverse order.

The reverse of a pair contains the same elements as the original, but they appear in the opposite order. For example, the reverse of the pair \((\text{quote (a b)})\) is \((\text{quote (b a)})\).

Examples:

\[
\begin{align*}
(flip (quote ((Wadsworth Billiards) (Yvette Kitchen) (Plum Conservatory)))) & \Rightarrow ((\text{Billiards Wadsworth}) (\text{Kitchen Yvette}) (\text{Conservatory Plum})) \\
(flip (quote ((Peacock Ballroom) (Boddy Study)))) & \Rightarrow ((\text{Ballroom Peacock}) (\text{Study Boddy}))
\end{align*}
\]

Task: Write a procedure flipper that takes as input a list of symbols and outputs a new list in which the first input element becomes the second output element and the second input element the first output element, the third input element becomes the fourth output element and the fourth input element the third output element, and so on. If the input list has no other element with which to flip the last element, then it should remain in its initial position.
That is, the first element becomes the second element and the second element the first, the third element becomes the fourth element and the fourth the third, and so on. If there is no other element with which to flip the last element, then it should remain in its initial position.

Examples:

```racket
(flipper (quote (Monopoly Clue)))
=> (Clue Monopoly)

(flipper (quote (Monopoly Clue Scrabble Risk Catan)))
=> (Clue Monopoly Risk Scrabble Catan)
```

Sets: An Introduction

When dealing with lists of suspects, locations, and evidence, it is essential that all of the evidence is distinct and contains no duplicates. A set is a collection of distinct elements, such that there are no repeated elements, making it the perfect tool for helping investigators to determine who killed Mr. Boddy! In the next two problems, we will be representing sets by lists with no duplicates. We will only consider sets of atomic data (that is, lists of atomic data with no duplicates), as this is data for which member? is guaranteed to work.

3 Set?

Task: Define a set predicate, called my-set?, that determines whether a list is a faithful representation of a set (i.e., whether all of its elements are distinct). Your procedure should be able to operate on lists of any single atomic data type (integers, booleans, symbols, etc.).

Hint: Use Racket's built-in member? procedure.

```racket
(my-set? (quote (Queen King Queen)))
=> false

(my-set? (quote (1 2 3)))
=> true

(my-set? empty)
=> true ;; vacuously -- empty has no elements, so they are all distinct
```

Let $M(n)$ be the number of elementary operations involved in evaluating (member? datum alod), where datum is any atomic value, and alod is a list of atomic data of length $n$. Then it's easy to see that

$$M(0) = D \quad \quad (1)$$
$$M(n) \leq C + M(n - 1) \quad \quad \quad \text{for } n > 0 \quad \quad (2)$$

and from the analysis we did for all recurrences like this, we see that $M(n) \leq Cn + D$ for every natural number $n$. As such, we can include that member? is a linear time procedure.
Task: Write a recurrence relation for my-set?. Using your recurrence relation, conclude whether my-set? is a linear-time procedure or a quadratic-time procedure.

Hint: Since counting operations exactly can be a pain, you may use constants to denote operation counts.

Note: You should explain how you derived your recurrence relation. That is, you should state which expressions contribute to any constant time cost, and which expressions contribute to any linear cost, etc.

Note that if an expression has at least the same order of growth as its subexpressions, you need not explain the subexpressions.

For example, when explaining (member? (car aloi) aloi) is linear time, we know that member? is linear time, and as such, we can ignore the runtime implications of evaluating (car aloi), evaluating aloi, binding the arguments, etc.

Use this to simplify your explanation.

4 Set-Equal?

Task: Define a procedure called set-equal? that determines whether two sets are equal (i.e., whether two lists, each with no duplicates, contain the same elements, regardless of order). Once again, the elements of the sets can be of any atomic type.

Note: Your procedure need only work on lists with no duplicates. It need not function correctly on arbitrary lists.

Hint: Two sets A and B are equal exactly when

- every element of A is an element of B, and
- every element of B is an element of A.

Hint: It may simplify your program a little if you write a helper procedure.

(set-equal? (quote (Monopoly Uno Clue)) (quote (Clue Monopoly Uno)))
=> true

(set-equal? (quote (Monopoly Uno)) (quote (Uno Monopoly Sorry)))
=> false

(set-equal? (quote ()) (quote ()))
=> true

Note: Remember to implement the design recipe (including test cases) for any helper procedures.

Task: Write a recurrence relation for each recursive procedure you write. Using your recurrence relation, conclude whether the procedure is a linear-time procedure or a quadratic-time procedure.

Hint: Since counting operations exactly can be a pain, you may use constants to denote operation counts.
Note: You should explain your recurrence relation. See the note in the previous problem for details on the best way to explain this.

5 Analysis

Recall the definition of big O from lecture: for mathematical functions \( f(n) \) and \( g(n) \), each with domain \( \{0, 1, 2, \ldots\} \), we say \( f(n) \) is \( O(g(n)) \) if there are constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for every value \( n \) greater than \( n_0 \).

Task: Each problem part specifies a function \( f \). Your job is to determine which is the first function \( g(n) \) in the following list for which \( f(n) \) is \( O(g(n)) \), and show that \( f(n) \) is \( O(g(n)) \) for that choice:

\[
g(n) = 1, g(n) = n, g(n) = n^2, g(n) = 2^n
\]

To show that \( f(n) \) is \( O(g(n)) \), you should tell us specific values of \( c \) and \( n_0 \), the constants in the definition of big O, and show with a little algebra that the inequality holds. To make things easier, you can make use of the following fact (outlined in lecture):

If two functions \( f_1(n) \) and \( f_2(n) \) are both \( O(g(n)) \) then the function \( f(n) \) defined by \( f(n) = f_1(n) + f_2(n) \) is also \( O(g(n)) \).

The same holds true, e.g., for the sum of three functions instead of two.

1. \( f(n) = 5 + 2n^2 + 7n \)
2. \( f(n) = (2n^2 + 1)(3n + 2) \)
3. \( f(n) = 10 \sin(n) \)
4. \( f(n) = \sqrt{n} \)
5. \( f(n) = 2n \sin(n) + \sqrt{n} \)

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](http://cs.brown.edu/courses/csci0170/feedback).