Homework 3: Racket Practice and Formal Semantics

Due September 27th at 10pm

Part 1: Formal semantics (50 points)

1. division (10 points)

A. Write the big step semantics for the division operator.

B. Test out your semantics by tracing the derivation of at least 3 examples mathematically, and comparing your result with what the Racket interpreter produces.

Report any issues with your big step semantics, and how you would fix them.

2. desugaring (10 points)

In class, we ran into an issue with our initial big step semantics for addition: in Racket, the addition operator can take any number of arguments. One solution is to treat the n-parameter version of the addition operator as **syntactic sugar** and rewrite it as a combination of additions with two arguments each. I presented the following rewrite rule for string-concatenation:

Version 1: (string-append (string-append e1 e2) e3)

But we could have written it like:

Version 2: (string-append e1 (string-append e2 e3))

Racket does, in fact, take the desugaring approach. Can you find evidence about which version of the desugaring rule Racket actually uses?

Hint: this is a case where side effects are useful!

3. conditionals (30 points)

A. Write the big step semantics for **if**.

B. Test out your semantics by tracing the derivation of at least 5 examples mathematically, and comparing your result with what the Racket interpreter produces.

Report any issues with your big step semantics, and how you would fix them.

Part 2: Racket practice (50 points)

4. tail-recursive product of digits (10 points)

Write a tail-recursive function that takes a number and returns the product of its digits. Call it **digits-product**.

Your solution does not need to handle negative numbers or decimals.

For example:

> (digits-product 123)

6

5. is-sorted? (10 points)

Write a function called **is-sorted?** that checks whether a list is sorted. It should return true if the list is a sorted list of strings or a sorted list of numbers, but false if the elements are not in order or if the list contains heterogenous datatypes.

For example:

> (is-sorted? (list 1 2 3))

#t

> (is-sorted? (list 1 "cat" 3))

#f

6. Merge sort (30 points)

Implement merge sort in Racket. You may not use any built-in sorting functions, but you may find the built-in functions **floor** and **take** helpful. **Floor** rounds a number down to the nearest integer, and **take** returns a portion of a list.

You will probably want to define a couple of helper functions, which you can either define above your main merge-sort function, or within the body using **letrec**.

Hint: you may find it easier to first implement a version of merge sort that always partitions a single element apart from the rest of the list, and then rewrite your function to divide the list in half once you have the rest working.

The algorithm for merge-sort is sketched below.

Split the list in half.
Keep dividing each sublist into smaller and smaller lists until each list consists of a single element. Those lists are sorted.
Merge pairs of lists back together, making sure that each merge operation returns a sorted list.

Extra Credit

1. desugaring

Does the desugaring approach to n-parameter addition work for functions in general? Prove or disprove that an n-parameter function may always be rewritten as a sequence of single-parameter functions.

Consider only programs that terminate without errors.

You may give a mathematical proof, a counter-example, or a procedure for rewriting an arbitrary multi-parameter Racket function as a single-parameter Racket function.