Assignment 2

COMP321

Due Sept 27, 11:59pm on Canvas

You may submit one `\.rkt'` source code file — textual answers should be provided as in-line comments, similar to how the sample code is structured. Correctly identify which functions/lines of code are answers to which question when you write answers in comments. You can also submit textual answers as a PDF separately.

1 You're looking for closure, but it doesn't exist (20 Points)

Write a function `genFib` that returns a **closure** that, when called, returns the next fibonacci number. So the first time you call it, it should return 0, then 1, then 1 again, then 2…You will need to mutate variables within `genFib`.

```
>(define (genFib) …)
>(define nextFib (genFib))
>(nextFib)
0
>(nextFib)
1
>(nextFib)
1
>(nextFib)
2
>(nextFib)
3
```

What variables in your function definition are free variables?

2 360 No Scope (20 Points)

This question requires no code in Racket to answer, it's simply a question about scope of operators and how scope is implemented in the **Python programming language**. Answer each question succinctly (no more than 50 words).

(a) Explore the scope of the Python variable definition operator(=). You should explore its behavior in at least the following contexts: (1) the global program context, (2) within a function, and (3) within nested functions. You should also explore how multiple uses of the operator interact. Does it's scope and meaning change in these different contexts?

(b) Python's `global` declaration modifies the scope of variables within its scope.

```
def foo():
    global x
    x = "outside"
    return()
> foo()
> x
```

What does `global` do? What does it do to variables within its scope? What is its scope? How does global interact with mutation in python?

3 Its lambdas all the way down (30 Points)

Write lambda or named functions as the procedures to higher-order functions (`map`, `foldl`, `filter`...) that do the following intended operations on a list of values. Please write **at least one good test case** that illustrate that your lambda function (with a higher-order convenience function of your choice) works as intended.

Apart from the other higher order functions, you may use basic list and string functions, like `cons`, `list?`, `string=?`, `empty?`, `min`...

(a) Get only the consonants from a list of string characters (Assume each list value is a string of length 1).

(b) Write a function that returns the sum of squares of the items in a list of numbers. Call this sum-of- squares.

(c) Flatten a nested list into a flat list. For example, calling the function on '('(1) 2 3 '(4) '((5)) 6)) should return '(1 2 3 4 5 6). Obviously don't use the built-in `flatten` function.

(d) Find the largest element in a list of numbers.

(e) Given two lists as arguments, return **a list of lists**, where each nested list at index i contains the ith item of list 1 and the ith item of list 2. For example,

```
> (...lambda... '(1 2 3) '(4 5 6))
> '((1 4) (2 5) (3 6))
```

If the lists are different sizes, truncate the longer list to be the same length as the shorter one (return as soon as either list is empty).

(f) Write a function that takes a comparison operator, a value, and a list of numbers, and returns all items from a list of numbers that are satisfy the comparator with respect to the value.
For instance, if the comparison operator is <= and the value is 5, the function returns all items in the list that are less than or equal to 5.

4 Sort it out among yourselves (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 rough algorithm for merge sort is:

1. Split the list in half
2. Keep dividing each sublist into smaller lists until each list is one element.
3. Sort each list.
4. Merge pairs of lists back together, making sure that each merge operation returns a sorted list.

What is the time and space complexity of your implementation?

5 Bonus: Map men (10 Points)

Write a version of `map` using the `foldl` higher order function. Call it my-map.