Homework 1: Scheme

Accept this assignment on GitHub Classroom here: https://classroom.github.com/a/FzI06rAW.

This assignment provides an introduction to list processing, functional programming, and Scheme. Implement, comment, and test the following functions in scheme. Scheme includes some of the constructs below, but you **may not** use these built-in implementations. Instead, you must implement them yourself. Make sure all functions are thoroughly tested.

You must implement the functions as specified. You may write other helper functions and define test data in your file, but you may not change the functions' names or their signatures. If you do, the autograder will not award points.

Test your code! We have given you unit tests in the file `tests.rkt`. As with all assignment, you can run them by typing `make test` on Linux. Do not just blindly run the tests until you pass them all—look at the tests and add your own. Unit tests almost never provide 100% test coverage, and that is certainly the case for the tests we provide.

You can run the tests using Racket's testing GUI by opening `run-tests-gui.rkt` in DrRacket and hitting the "Run" button. This will give you a nice breakdown of exactly which tests passed and which failed.

Also note:

- You must use proper Scheme style. We use proper Scheme style in class. You may also refer to this Scheme style guide for general advice, but to be clear, it is *not* the case that anything goes!
- If a problem asks you to implement a *recursive* solution, your solution **must be recursive** or you will receive no credit. Remember, a recursive function is a function that calls itself.
- If a problem asks you to write a *pure* function, you **must not** use `set!` or any form of mutable state in your solution. If you do, you will receive no credit.
- No input error-checking is required; you may assume input obeys the constraints stated in the problem description. For example, if the problem says an argument is a positive integer, you do not have to handle negative arguments.
- I have annotated each problem with the number of lines of code in the body of my solution. No problem requires very much code. You *do not* need to match my line count in order to receive full credit.

You should complete the homework by modifying the files `README.md` and `scheme.rkt` that we provide as part of the assignment. You do not need to create or modify any other files for this submission.

Make sure you push your changes to GitHub and **submit your work to GradeScope**.

This assignment is worth 100 points. There are 100+6 possible points.

Notes on style

The code we see in class adheres to standard Scheme style. In particular, the following are considered **bad** style:

1. Placing parentheses on lines by themselves.
2. Using `cond` when there are only two cases (use `if`)

You may lose one or more points for *each* problem exhibiting poor style.

If you have questions about style, please ask!

1 Problem 1: `(falling x n)` (20 points)

Define a **pure**, **recursive** function `falling` that implements the falling factorial function, $x^{\underline{n}}$ (pronounced "x to the falling n"), defined as

$$x^{\underline{n}} = \overbrace{x(x-1)\ldots(x-n+1)}^{n \text{ factors}}, \quad \text{integer } n \geq 0$$

Notice the straight line under the n; this indicates that the n factors descend stepwise. By convention, the product of zero factors is 1, so $x^{\underline{0}} = 1$. You may assume that x and n are both integers and that $n \geq 0$. It is possible for x to be zero or negative.

The following facts may help you test your implementation:

1. $n! = n^{\underline{n}}$
2. $x^{\underline{n}} = x!/(x-n)!$

You can read more about the falling factorial on Wikipedia. Unlike Wikipedia, we use D. E. Knuth's notation for the falling factorial; see the book *Concrete Mathematics* for more.

Your solution *must not* use the factorial function to compute the falling factorial function; instead, your implementation should be $\mathcal{O}(n)$. How could we write a test that, if it passed, gave us confidence that `falling` was not implemented using the factorial function? Hint: for what value(s) of n is the falling factorial easy to compute, but for which an implementation using factorial would take a long time to complete?

Examples:

```
(falling 10 1) => 10
(falling 10 10) => 3628800
```

My solution: 3 lines

2 Problem 2: `(S n k)` (20 points)

This problem provides practice translating a (mathematical) recurrence relation into executable code. You **do not** need to understand Stirling numbers (of any kind) to complete the problem. Just implement the recurrence relation as a recursive Scheme function!

Define a **pure**, **recursive** function S that computes the Stirling numbers of the second kind. The Stirling numbers of the second kind, written $S(n,k)$ or $\left\{ {n \atop k} \right\}$ and pronounced "n subset k", count the number of ways to partition a set of n distinct items into k non-empty partitions. One way to compute the Stirling numbers of the second kind is by using the following recurrence relation:

$$\begin{aligned}
\left\{ {n \atop n} \right\} &= 1, && \text{for } n \geq 0 \\
\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} &= 0, && \text{for } n > 0 \\
\left\{ {n \atop k} \right\} &= k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}, && \text{for } n > 0
\end{aligned}$$

The following facts may help you test your implementation:

$$\sum_{k=0}^{n} \left\{ {n \atop k} \right\} x^{\underline{k}} = x^n$$

$$\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$$

For the curious: You can read more about the Stirling numbers of the second kind on Wikipedia. The Stirling numbers of the second kind are sequence A008277 in the OEIS.

My solution: 4 lines

3 Problem 3: `(zip xs ys)` (20 points)

Define a **pure**, **recursive** function `zip` that takes two arguments, each of which is a list, and produces a list of pairs, where the n^{th} pair's `car` is the n^{th} element of `xs` and the n^{th} pair's `cdr` is the n^{th} element of `ys`. The behavior of `zip` is undefined if either `xs` or `ys` is not a list or if `xs` and `ys` are both lists but do not have the same length.

Examples:

```
(zip '(1 2 3) '(a b c)) => ((1 . a) (2 . b) (3 . c))
```

My solution: 4 lines (I did some extra error checking)

4 Problem 4: `(pow f n)` (20 points)

Write a **pure** higher-order function pow that take a function `f` and a non-negative integer n as arguments and returns the function f^n, which composes `f` with itself n times. The pow function you write implements an analogue of exponentiation, but for functions. This operation is also called *function iteration*.

In mathematical notation,

$$f^n(x) = (\underbrace{f \circ \cdots \circ f}_{n \text{ instances of } f})(x)$$

That is, f^n is the function that consists of the composition of n instances of f. For example, given the following definition:

```
(define (square x) (* x x))
```

the value of the expression `(pow square 2)` should be a procedure that is equivalent to the procedure that is value of the expression `(lambda (x) (square (square x)))`.

For any number n, $n^0 = 1$ and $n^1 = 1$. Analogously, for any function f, f^0 is the identity function and f^1 is just f. Therefore, for any argument `f`, the value of `(pow f 0)` should be the identity function, and the value of `(pow f 1)` should be the value of `f`.

My solution: 3 lines

5 Problem 5: `(argmax f xs)` (20+5 points)

Define a **pure**, **recursive** function `argmax` that takes two arguments: a function `f` that maps a value in set A to a number, and a *nonempty* list `xs` of values in set A. It returns an element `x` in `xs` for which `(f x)` is as large as possible. You may assume `f` is pure, i.e., it always produces the same result for the same argument, and that `xs` is non-empty.

In mathematical notation, `(argmax f xs)` is written (the two are equivalent)

$$\text{argmax}_{xs}\, f = \underset{x \in xs}{\text{argmax}}\, f(x)$$

Assuming `f` takes constant time, `argmax` should be $\mathcal{O}(n)$ in the length of `xs`. There are many ways to make the asymptotic complexity of `argmax` worse than $\mathcal{O}(n)$, but two common ways are computing the length of the `xs` argument in each recursive step and computing $f(x)$ more than once for each value x.

Examples:

```
(define (square x) (* x x))
(argmax square '(1 2 -10)) => -10
```

Hint: The specification of `argmax` states that `xs` is non-empty. Exploit this fact. You may want to use a `let` expression, but first try to implement your solution without `let`. Once you having a working solution that you understand, you may see an opportunity to use `let` to avoid recomputing a value.

Bonus: If you're up for a real challenge, write an implementation of `argmax` that calls `f` the fewest number of times possible. In general, `argmax` only needs to call `f` *at most once* for each item in `xs`. There is one case where `argmax` can get away without calling `f` at all. If we take advantage of the fact that `f` is pure, then for some inputs we can do even better, but accomplishing this is *quite* tricky. If you can write an `argmax` that calls `f` at most once for each item in `xs`, it is worth 5 bonus points.

My solution: 6 lines

6 Problem 6: Homework Statistics (1 point)

How long did spend on each problem? Please tell us in your `README.md`.

You may enter time using any format described here. Please enter times using the requested format, as that will make it easy for us to automatically collect.

We are happy to read additional comments you have, but **do not** put any comments on the same line as the time report. Please **do not** otherwise edit, move, or reformat the lines reserved for time statistics reports.

Here is the reporting format you should use:

```
Problem 1: 5m

Here's a description of my experience solving this problem.
```

If you did not attempt one or more problems, you can still receive full marks by telling us you spent 0 minutes on these problems. You will not receive points for this problem if you leave **any** of the time reports blank.