CSE 341 — Haskell Mini-Exercises # 3

These are questions for discussion in class. (You don't need to hand in anything.) The solutions are on the class web page.

1. Suppose that we have the following definition of the `fix` function in Haskell:

   ```
   fix f = f (fix f)
   ```

   What is the type of each of the following expressions? (Or it might give an error.)

   ```
   fix
   fix (*0)
   fix (+1)
   fix ord  -- ord is a function that takes a character and returns its ascii code
   fix (const 10)
   fix id   -- id is the identity function
   fix error
   ```

2. What is the value returned by evaluating each of these expressions in Haskell? If they get into an infinite recursion, what *is* the fixed point of the function, if it has one? (There might be zero, or many.)

   ```
   fix (*0)
   fix (+1)
   fix ord  -- ord is a function that takes a character and returns its ascii code
   fix (const 10)
   fix id   -- id is the identity function
   fix error
   ```

3. Define a recursive function `range` in Haskell that takes an integer `n` and returns a list of integers from `n` down to 0. For example `range 5` returns `[5,4,3,2,1]`.

4. Define a non-recursive version of `range` that uses the `fix` function to remove the recursion.

5. Define the `range` function in Racket.

6. Define a non-recursive version of `range` in Racket using the Y combinator.

7. Consider the factorial example in the notes `fixedpoint.hs`. In preparation for defining a non-recursive version of factorial using `fix` function we turned the factorial function into a parameter of a lambda. We then gave it a name:

   ```
   improver = (\f -> (\n -> if n==0 then 1 else n*f (n-1)))
   ```

   Then we could define factorial by taking the fixed point of this function:

   ```
   fact3 = fix improver
   ```

   Suppose we define a function `pathetic_factorial` that can compute the factorial of 0, 1, 2, 3, or 4:

```
pathetic_factorial n = case n of
  0 -> 1
  1 -> 1
  2 -> 2
  3 -> 6
  4 -> 24
  _ -> error "sorry, I don't know how to compute that"
```

What is the result of evaluating these expressions?

```
pathetic_factorial 0
pathetic_factorial 1
pathetic_factorial 2
pathetic_factorial 3
pathetic_factorial 4
pathetic_factorial 5
pathetic_factorial 6
```

Now let's try feeding this function to `improver`. What is the result of evaluating these expressions?

```
improver pathetic_factorial 0
improver pathetic_factorial 1
improver pathetic_factorial 2
improver pathetic_factorial 3
improver pathetic_factorial 4
improver pathetic_factorial 5
improver pathetic_factorial 6
```