Homework 2

CS 131, Fall 2024
Carey Nachenberg

Soyeon Kim, 106292803

HASK1

```
largest :: String -> String -> String
largest str1 str2
 | length str1 > length str2 = str1
 | length str1 < length str2 = str2
 | otherwise = str1
```

HASK2

```
reflect :: Int -> Int
reflect 0 = 0
reflect num
 | num < 0 = (-1) + reflect (num+1)
 | num > 0 = 1 + reflect (num-1)
```

He needs to wrap `num+1` and `num-1` with parentheses. `reflect num+1` is same as `(reflect num) + 1`, and this will call the `reflect` function without changing the `num` value which causes the infinite recursion without reaching the base case 0.

HASK3

Without mutual recursion

```
is_odd :: Int -> Bool
-- if statement
is_odd num =
 if num == 0 then False
 else if num == 1 then True
 else is_odd (num-2)
-- guards
is_odd num -- guards
 | num == 0 = False
 | num == 1 = True
 | otherwise = is_odd (num-2)
-- pattern matching
is_odd 0 = False
is_odd 1 = True
is_odd num = is_odd (num-2)

is_even :: Int -> Bool
```

```haskell
-- if statement
is_even num =
 if num == 0 then True
 else if num == 1 then False
 else is_even (num-2)
-- guards
is_even num
 | num == 0 = True
 | num == 1 = False
 | otherwise = is_even (num-2)
-- pattern matching
is_even 0 = True
is_even 1 = False
is_even num = is_even (num-2)
```

With mutual recursion

```haskell
is_odd num
 | num == 0 = False
 | num == 1 = True
 | otherwise = is_even (num-1)
is_even num
 | num == 0 = True
 | num == 1 = False
 | otherwise = is_odd (num-1)
```

HASK4

```haskell
quad :: Double -> Double -> Double -> (Double, Double)
quad a b c
 | a == 0 || x < 0 = (0, 0)
 | otherwise = (((-b) + sqrt x) / (2*a), ((-b) - sqrt x) / (2*a))
 where
   x = b^2 - 4*a*c
```

HASK5

```haskell
sum_is_divisible :: Int -> Int -> Int -> Bool
sum_is_divisible a b c = (sum_range a b) `mod` c == 0
 where
   sum_range start end
     | start > end = 0
     | otherwise = start + (sum_range (start+1) end)
```

Using list comprehension

```haskell
sum_is_divisible :: Int -> Int -> Int -> Bool
sum_is_divisible a b c = (sum [a..b]) `mod` c == 0
```

HASK6

```
find_min :: [Int] -> Int
find_min x
 | length x == 1 = head x
 | otherwise = min (head x) (find_min (tail x))
```

Using list comprehension

```
find_min :: [Int] -> Int
find_min [x] = x
find_min (x:xs) = min x (find_min xs)
```

HASK7

Part A

```
all_factors :: Int -> [Int]
all_factors n = [x | x <- [1..n], n `mod` x == 0]
```

Part B

```
perfect_numbers :: [Int]
perfect_numbers = [n | n <- [1..], sum (init (all_factors n)) == n]
```

HASK8

```
count_occurrences :: [Int] -> [Int] -> Int
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x:xs) (y:ys)
  | x == y = count_occurrences xs ys + count_occurrences (x:xs) ys
  | otherwise = count_occurrences (x:xs) ys
```

- `count_occurrences xs ys`: continue matching the rest of a1 with the rest of a2
- `count_occurrences (x:xs) ys`: skip the current match

HASK9

Part A

```
fibonacci :: Int -> [Int]
fibonacci n = _fibonacci 1 1 n
 where
   _fibonacci :: Int -> Int -> Int -> [Int]
   _fibonacci a b count
     | count <= 0 = []
     | otherwise = a : _fibonacci b (a + b) (count - 1)
```

Part B

```haskell
rle :: [Int] -> [(Int, Int)]
rle [] = []
rle (x:xs) = _rle x 1 xs
 where
   _rle :: Int -> Int -> [Int] -> [(Int, Int)]
   _rle curr count [] = [(curr, count)]
   _rle curr count (y:ys)
     | curr == y = _rle curr (count + 1) ys
     | otherwise = (curr, count) : _rle y 1 ys
```