Homework Assignment 1

Functional and Logic Programming, 2024

Due date: Thursday, April 18th, 2023 (18/04/2024)

Bureaucracy

- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW1_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW1_<id>.zip` if submitting alone.
- The zip file should contain **a single file** named `HW1.hs`!
- Make sure your submission compiles successfully. Submissions which do not compile will receive a 0 grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW1.hs`.
- You may submit the assignment after the due date even without approval (e.g., excluding reserve duty, serious illness, or other cases covered by the student administration).
  - You will be penalized for **5 points for every late day**.
  - The **maximum** extension allowed by this is **3 days**.
- If you don't know how to implement some function, **do not remove it**! Use `undefined` in the implementation, otherwise your entire submission will fail compilation, which will also result in a 0 grade.
  - This is especially true to for the bonus section!

General notes

- The instructions for this exercise are split between this file and `HW1.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints. The Haskell file details all the required functionality for this assignment.
- You may not modify the **`import`** statement at the top of the file, nor add new **`import`**s.

  - N.B. HLS has the habit of adding **unnecessary imports** when it doesn't recognize an identifier, **so please double check this before submitting**!
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also supports module lookups, e.g., `Prelude.not`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
    * And in some cases their definition may not be entire clear just yet!

- The exercises and sections are defined in a linear fashion. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.
  - Do not be alarmed by the large amount of functions! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.
  - In general, you may define as many helper functions as you wish.
- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Although it is not required in the homework assignments, non-elegant code *will* be penalized in the test, so this is a good exercise. HLS and hlint can be very helpful in this.
  - Do note that in some cases, hlint may suggest functions which are not imported!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: Warm-up

This section includes a bunch of simple utility higher-order functions . All of these should be self-evident from the signature alone, and in general are "impossible" to implement wrongly.
Hint: All functions can be implemented in a single (short) line!

Section 2: Basic integer functions

In this section, you will implement a few functions on `Integer`s. Since you cannot use lists or `String`s in this assignment, you will be making heavy use of manual recursion.
General hint: in Haskell, `div` is used for integer division, and `mod` is used for computing modulo (`(/)` is "usual" division).

```
42 `div` 10
4
42 `mod` 10
2
42 / 10
4.4
```

- `countDigits` counts the number of digits in an integer. Negative numbers have the same number of digits as their positive counterparts.

```
countDigits 0
1
countDigits 1024
4
countDigits (-42)
2
```

`toBinary` and `fromBinary` are a pair of functions for converting numbers to and from "pseudo-binary" representation. You can assume the input to `fromBinary` is valid, i.e., it is an `Integer` made up of only 1s and 0s.

```
toBinary 0
0
toBinary 1
1
toBinary 42
101010
toBinary (-10)
-1010

fromBinary 0
0
fromBinary 1
1
fromBinary 101010
42
fromBinary (-1010)
-10
```

Hint: `toBinary . fromBinary` and `fromBinary . toBinary` should both be equivalent to `id`!

```
fromBinary . toBinary $ 42
42
toBinary . fromBinary $ 101010
101010
```

- `isAbundant` checks if a number is abundant or not, i.e., the sum of its proper divisors is greater than the number itself.

```
isAbundant 12 -- (1 + 2 + 3 + 4 + 6 = 16)
True
isAbundant 15 -- (1 + 3 + 5 = 9)
False
isAbundant 18 -- (1 + 2 + 3 + 6 + 9 = 21)
True
isAbundant (-12) -- Only positive numbers are considered abundant
False
```

- `roateDigits` rotates the digits of a number to the left. Negative numbers rotate to the right.

```
rotateDigits 1
1
rotateDigits 1234
2341
rotateDigits (-1234)
-4123
rotateDigits 102
21
rotateDigits (-102)
-210
```

Section 3: Generators

We can define a (possibly infinite) generator as a triplet of two functions for generating elements and checking if we should continue generating, and an initial seed. For example, the following generates all the positive integers (the initial value is not considered part of the generated set).

```
type Generator a = (a -> a, a -> Bool, a)
positives :: Generator Integer = ((+ 1), const True, 0)
```

In this section you will implement a few utility functions for working with such generators.

- `nullGen` checks if no elements are generated.

```
nullGen ((+ 1) , (< 0), 0)
True
nullGen ((+ 1) , (<= 0), 0)
False
nullGen ((+ 1) , (<= 1), 0)
False
nullGen positives
False
```

- `lastGen` returns the final element generated by the generator, or the initial element if the generator is empty.

```
lastGen ((* 2), (< 1), 1)
1
lastGen ((* 2), (<= 1), 1)
2
lastGen ((* 2), (< 1000), 1)
1024
```

- `lengthGen` returns the number of elements generated.

```
lengthGen ((+ 1), (< 0), 0)
0
lengthGen ((+ 1), (<= 0), 0)
1
lengthGen ((+ 1), (< 10), 0) -- The generator is equivalent to [1,2,...,10]
10
```

- `sumGen` returns the sum of all the elements generated by the generator.

```
sumGen ((+ 1), (< 10), 0)
55
sumGen ((+ 1), (< 10), 1)
54
sumGen ((+ 1), (< 10), 10)
0
```

Next we'll take a look at a set of predicate functions for checking properties of the entire generated values against some predicate.

- `allGen`, `anyGen`, and `noneGen` check if all, any, or no elements respectively satisfy some predicate. For infinite generators, in some cases you can still return an answer ("short-circuit"), e.g., when you find a counter-example. Reminder: the initial element does not count for these tests!

```
positivesUpTo10 :: Generator Integer = ((+ 1), (< 10), 0)
allGen (< 10) positives
False
allGen (> 0) positivesUpTo10
True
anyGen (< 10) positives
True
anyGen (<= 0) positivesUpTo10
False
noneGen (< 10) positives
False
noneGen (<= 0) positivesUpTo10
True
```

- Lastly, `countGen` counts all the elements that satisfy some predicate.

```
countGen even positivesUpTo10
5
countGen even ((+ 1), (< 10), 1)
5
countGen even ((+ 1), (< 9), 1)
4
```

Hints:

- Make sure you check edge cases, such as empty generators, ignoring the initial element, and short-circuiting for infinite generators.
- You can assume that the input the functions like `length` and `count` will never be an infinite generator.
- Many of these functions can be implemented in terms of each other. However, take special care around infinite and finite generators, as some functions will never halt for infinite generators!

Section 4: Prime numbers

In this section you will implement a few functions for working with prime numbers.

- `isPrime` checks if a number is prime or not. Reminder: all numbers smaller than 2 are not prime.

```
isPrime 1
False
isPrime 2
True
isPrime 29
True
isPrime (-2)
False
```

- `isSemiprime` checks if a number is a semiprime or not, i.e., a natural number that is the product of **exactly two** prime numbers.

```
isSemiprime 2
False
isSemiprime 10
True
isSemiprime 77
True
isSemiprime (-10)
False
```

- The Goldbach conjecture states that every even number greater than 2 can be expressed as the sum of two prime numbers. `goldbachPair` returns a pair of prime numbers that sum up to the given number. If there are two such possible pairs, return the one where the **left number is smaller**. You can assume the input to this function will always be an even number greater than 2.[1]

```
goldbachPair 4
(2, 2)
goldbachPair 10
(3, 7)
goldbachPair 100
(3, 97)
```

- `goldbachPair'` is a variant of `goldbachPair` that returns a pair of prime numbers that sum up to the given number, but such that the product of the two primes is the **largest** of all possible pairs. If there are two such possible pairs, return the one where the left number is **greater**. You can assume the input to this function will always be an even number greater than 2.

```
goldbachPair' 4
(2, 2)
goldbachPair' 10
(5, 5)
goldbachPair' 100
(53, 47)
```

[1]On the off-chance you found a counter example to the Goldbach conjecture, please let us know!

Hint: You can use the functions from the previous sections to help you implement some of these tasks!

Section 5: Bonus (10 points)

`isCircularPrime` checks if a number is a circular prime or not, i.e., all the rotations of the number are prime.

```
isCircularPrime 5
True
isCircularPrime 17
True
isCircularPrime 103
False
isCircularPrime 193
False
isCircularPrime 197
True
isCircularPrime 199
True
```