Homework Assignment 1

Functional and Logic Programming, 2024

Due date: Tuesday, April 22nd, 2025 (22/04/2025)

Bureaucracy

- The staff member in charge of this assignment is Gal Lalouche (gal.lalouche@post.runi.ac.il).
- 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.
  - You do not need special permission to submit alone.
- The zip file should contain **a single, top-level file (no folders!)** named `HW1.hs`!
- Make sure your submission compiles successfully. Submissions which do not compile will not receive a 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 no 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 and examples. 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 ask HLS or Hoogle.
  - Hoogle also supports module lookup, 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.
  - If you're feeling fancy, you might even implement some functions using later ones!
- 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 *strictly required* in the homework assignments, elegant code *will* receive a higher-grade 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.

A note on testing

- The suggested way of testing yourselves is either via `GHCi` or using `-- >>>` comments, as taught in the tutorials.
- Although we do not cover unit tests in class, adventurous students can use the internet or even a generative AI to generate a template file for writing Haskell tests.
  - Personally, we recommend looking into HUnit (see tutorial here), or a wrapper for HUnit called Tasty.
  - N.B. A **very** important part of programming in general is writing your own unit tests, so try not to offload *too* much work to your friendly neighborhood chatbot.

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, `mod` is used for computing modulo (`(/)` is "usual" division), and `(^)` is used for exponentiation.

```
42 `div` 10
4
42 `mod` 10
2
42 / 10
4.5
2 ^ 10
1024
```

- `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
```

- `sumDigits` sums the digits of a number. Negative numbers are considered to have the same digits as their positive counterparts.

```
sumDigits 0
0
sumDigits 1024
7
sumDigits $ -42
6
```

  Challenge: Can you implement both `countDigits` and `sumDigits` using the same helper function? Such a helper function could be useful elsewhere in the assignment!

- `reverseDigits` reverses the digits of a number. Negative numbers remain negative after reversing.

```
reverseDigits 1234
4321
reverseDigits $ -42
-24
reverseDigits 120
21
```

- The Collatz function is defined thus:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

It is conjectured that every positive number will eventually reach 1 after applying the Collatz function repeatedly. The length of the sequence from a number to 1 is called the **Collatz sequence length**. You can assume the input will always be a positive integer.

```
collatzLength 1
0
collatzLength 2
1
collatzLength 3
7
collatzLength 4
2
collatzLength 1024
10
collatzLength 1025
36
```

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.

```
type Generator a = (a -> a, a -> Bool , a)
```

For example, the following generates all the positive integers (the initial value is **not** considered part of the generated sequence).

```
positives :: Generator Integer
positives = ((+ 1), const True , 0)
```

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

3.1 Basic functionality

- `nthGen` returns the n^{th} element generated by a generator. If n is larger than the length of the list, return the last element. You can assume $n \geq 0$.

```
nthGen 0 positives
1
nthGen 2 positives
3
nthGen (-1) positives
0
nthGen 42 ((+1), (< 10), 0)
10
```

- `hasNext` checks if there is another element to generate.

```
hasNext ((+ 1), (<= 1), 0)
True
hasNext ((+ 1), (<= 0), 0)
True
hasNext ((+ 1), (<= 0), 1)
False
```

- `nextGen` returns generator after applying the function. It ignores the continue check, but it doesn't have to be total if the generator has stopped (as in `nextGen emptyGen` in the next section).

```
thd3 $ nextGen ((+ 1), (< 0), 0)
1
```

- `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
```

- Since `lengthGen` would run forever for infinite generators—In other words, it is not **total**—we can use `hasLengthOfAtLeast` if we want to verify the length of a generator is at least some value.

```
hasLengthOfAtLeast 10 ((+ 1), (< 10), 0)
True
hasLengthOfAtLeast 10 ((+ 1), (< 9), 0)
False
hasLengthOfAtLeast 42 positives
True
```

3.2 Generating `Generators`

While we can create generators by explicitly defining the tuple, we can also create generators using helper functions

- `constGen` creates a generator that always generates the same value.

```
nthGen 42 $ constGen "foobar"
"foobar"
```

- `foreverGen` creates a generator that generates an infinite generator from some seed and a function. For example, an alternative definition of `positives` using `foreverGen` would be:

```
positives = foreverGen (+ 1) 0
```

- `emptyGen` Creates an empty generator, i.e., one whose length is always zero

```
lengthGen (emptyGen :: Generator Int)
0
lengthGen (emptyGen :: Generator (Int, Int))
0
lengthGen (emptyGen :: Generator (Generator Int))
0
```

  Hint: Note that it does not accept a seed, and yet the function is polymorphic for all `a`! How can this be implemented?

- `integers` generates all integers, **except 0**. There is no order requirement; rather, the only requirement is that every integer should appear at some **finite** index, i.e., you cannot generate all the positives "and then" all the negatives.

```
anyGen (== 42) integers
True
anyGen (== (-42)) integers
True
anyGen (== 0) integers -- Would run forever!
```

3.3 Interacting with `Generators`

- `sumGen` returns the sum of all the elements generated (again, not including the seed). If the generator is infinite, the function would never halt.

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

- `anyGen` checks if any element generated satisfies a predicate. If the generator is infinite, this function would only halt if the predicate is true for some generated value.

```
anyGen (> 0) ((+ 1), (<= 1), 0)
True
anyGen (const True) ((+ 1), (< 1), 1)
False
anyGen (const True) ((+ 1), (< 10), 0)
True
anyGen (> 42) positives
True
```

- `andAlso` Adds another check to the generator. The new generator will stop generating if **either** of the two checks is false.

```
lengthGen $ andAlso (< 10) positives
10
lengthGen $ andAlso (> 20) $ andAlso (< 10) positives
0
lengthGen $ andAlso (< 20) $ andAlso (< 10) positives
10
```

3.4 Bonus (15 points)

`divisors` generates all the positive divisors of a number smaller than the number itself, in increasing order. For example, `generators 6` should generate 1, 2, 3. A negative number has the same divisors as its positive counterpart. `divisors n` would be empty if and only if $|n| \leq 1$.

- Reminder: If you don't implement this function, implement it using `undefined`!
- Fun fact! We can check if a number is a perfect number using the following one-liner:

```
isPerfect n = n == sumGen (divisors n)
```

General notes and hints

- Generators are a special case of a **combinator library**, i.e., it provides a small set of general and useful functions we can use to build more complicated applications on top of. We will see more of these in the future.
- Make sure you check edge cases, such as empty generators, ignoring the initial element, and short-circuiting for infinite generators.

- We haven't reached lists yet, but if you want to verify the elements generated by a given `Generator`, you can use the following pair of functions, e.g., in `GHCi`:

```
-- Don't worry about this implementation right now!
toList :: Generator a -> [a]
toList gen = if hasNext gen then go $ nextGen gen else []
 where
  go g = currentGen g : if hasNext g then go $ nextGen g else []
toListLimited :: Int -> Generator a -> [a]
toListLimited n = take n . toList
-- Example usage
toListLimited 10 $ positives
[1,2,3,4,5,6,7,8,9,10]
```

Section 4: Number properties

In this section you will implement a few functions for working with prime numbers. Hint: You can use the previous functions in many of these!

- `isPrime` checks if a number is prime or not. Reminder: 2 is the smallest prime number.

```
isPrime 1
False
isPrime 2
True
isPrime 29
True
isPrime $ -2
False
```

- `nextPrime` returns the first prime number greater than the given number.

```
nextPrime $ -42
2
nextPrime 1
2
nextPrime 2
3
nextPrime 100
101
```

- `primes` generates all the prime numbers.

```
nthGen 0 primes
2
nthGen 100 primes
541
```

- `isHappy` checks if a number is a decimal happy number. Negative numbers can also be happy.

```
isHappy 7
True
isHappy 42
False
isHappy 130
True
isHappy $ -130
True
```

- `isArmstrong` checks if a number is a decimal Armstrong number. There are no negative Armstrong numbers.

```
isArmstrong 0
True
isArmstrong 1
True
isArmstrong 42
False
isArmstrong 153
True
```

- `isPalindromicPrime` checks if a number is a decimal palindromic prime, i.e., a prime number that is also a palindrome.

```
isPalindromicPrime 2
True
isPalindromicPrime 11
True
isPalindromicPrime 13
False
isPalindromicPrime 101
True
```