Advanced Programming
3rd Exercise: Functional Programming

Programming Group
guido.salvaneschi@unisg.ch

University of St.Gallen

1 Goals

In this exercise, you'll practice functional programming in Haskell and problem solving with immutability, function composition, higher-order functions, lazy evaluation, and type classes.

2 Functional Programming in Haskell

Haskell is a purely functional statically-typed language, which emphasizes immutability and declarativity.

Info: Functional Programming describes program as composition of mathematical functions, avoiding state and mutable data. Some other example languages are Haskell, Lisp, OCaml, F#, and others.

2.1 Functional Programming

Let's get familiar with reasoning in functional languages. Consider the Sieve of Eratosthenes, which is an efficient way for computing a sequence of prime numbers. Starting with a list of all the numbers from 2 to N it does the following:

1. It selects the first untagged number,
2. It traverses the list of numbers and removes all the numbers that are a multiple of it,
3. It tags that number as a prime.
4. Then, it returns to the first step until there are no more untagged numbers.

For example, suppose we want to find all the prime numbers between 2 and 10. These are the steps the algorithm takes:

1. Mark 2 as prime: $\dot{2}, 3, 4, 5, 6, 7, 8, 9, 10$
2. Delete all multiples of 2: $\dot{2}, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}$
3. Mark 3 as prime: $\dot{2}, \dot{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}$
4. Delete all multiples of 3: $\dot{2}, \dot{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}$
5. Mark 5 as prime: $\dot{2}, \dot{3}, \cancel{4}, \dot{5}, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}$
6. Delete all multiples of 5: $\dot{2}, \dot{3}, \cancel{4}, \dot{5}, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}$
7. Mark 7 as prime: $\dot{2}, \dot{3}, \cancel{4}, \dot{5}, \cancel{6}, \dot{7}, \cancel{8}, \cancel{9}, \cancel{10}$
8. Delete all multiples of 7: $\dot{2}, \dot{3}, \cancel{4}, \dot{5}, \cancel{6}, \dot{7}, \cancel{8}, \cancel{9}, \cancel{10}$

We conclude that the primes are 2, 3, 5, and 7.

Info: To run Haskell code, one simple way is to use the GHCi REPL (Read-Eval-Print Loop). You can enter it by running the command `ghci` in your terminal. Then, you can use the following commands from inside the REPL:

1. '<your haskell code>': evaluate your Haskell code
2. ':load file.hs' or ':l file.hs': import Haskell definitions from a file
3. ':reload' or ':r': reload all imported files
4. ':browse': show all imported definitions
5. ':type expr' or ':t expr': show the type of an expression
6. ':quit' or ':q': exit the REPL

Task 1: The Sieve of Eratosthenes

Let's write a program `primesBefore :: Integer -> [Integer]` that implements the Sieve of Eratosthenes to find the primes up to some number.

2.2 Lazy Evaluation

Haskell uses lazy evaluation, meaning that expressions are not evaluated until their results are needed. This allows for the creation of infinite data structures and can lead to performance improvements by avoiding unnecessary computations.

Task 2: Creating Infinite Lists

How can you define `nats :: [Integer]`, the infinite list containing all the natural numbers?

If we try to print the entirety of `nats` our program will go on indefinetely. However, we can still use `nats` productively by querying it incrementally, thanks to lazy evaluation.

Task 3: Querying Infinite Lists

In Haskell, `take` and `skip` are two important functions that take the first n items of a list, or skip the first n items of a list respectively. Implement the functions:

```
take' :: Integer -> [a] -> [a]
skip' :: Integer -> [a] -> [a]
```

Task 4: The Lazy Sieve of Eratosthenes

Implement `primes :: [Integer]` by modifying your implementation of the Sieve of Eratosthenes to return the list of *all* primes.

Task 5: The Mersenne Primes

The Mersenne primes are special primes that take the shape of $2^n - 1$ for some n. Implement the `mersenne :: [Integer]` function of the Mersenne primes. Then, implement a function to return the first five Mersenne primes with signature `first5Mersenne :: [Integer]`.

2.3 Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return functions as results. A common example is the `map` function, which transforms the element of a list.

Task 6: Map

Implement the `mapList :: (a -> b) -> [a] -> [b]` function that takes a function and a list, and applies that function to each element of the list, returning a new list with the results.

In the lecture, we've also seen `filter`, `foldl`, and `foldr` as common examples of higher-order functions and fundamental operations for manipulating lists.

Task 7: Generality of Folding

`foldr` is sufficient to implement all other operations on lists. Prove that claim by using `foldr` to implement `map'`, `filter'`, and `foldl'`.

Task 8: Functional List Manipulation (Optional)

Below is an imperative loop in a C-like language that returns the index of the first element of a list that satisfies some given predicate:

```
int findFirst(predicate, list) {
  int index = 0;
  foreach (x: list) {
    if (predicate(x)) {
      return index;
    }
    index += 1;
  }
  return -1;
}
```

Convert this program into the functional style using only `map`, `filter`, and `foldr`.

2.4 Data Types and Type Classes

Working with built-in data types is often not enough to solve real-world problems. Haskell allows us to define our own data types and type classes to model the domain of our application and its constraints.

Task 9: Custom Data Types

Algebraic Data Types (ADT) allow us to easily represent "syntax trees" of grammars. Given the following grammar, implement its corresponding data type:

```
Program ::= Instruction ";" Program
          | ""

Instruction ::= "push" Integer
              | "pop"
              | "dup"
              | "swap"
              | "apply" Op

Op ::= "add" | "sub" | "mul" | "div"
```

Task 10: Built-In Type-Classes

We wish to pretty-print the programs of the previous grammar. Haskell provides a built-in typeclass `Show` for converting values of some type into a string. Implement the `Show` typeclass for each of the previous data types.

Task 11: Custom Type-Classes

In this grammar, we are describing the programs in some limited stack-based instruction-set. Naturally, some of our data types describes computation over a stack. Describe that property with a typeclass:

1. What would you call that typeclass?
2. What would be its methods?
3. Which data types should implement it?

Task 12: Executing Programs

Implement the typeclass you defined on the data types that you identified. Test your implementation by running the following programs:

1. ```
   Input:  push 1; push 2; dup; apply mul; swap; apply div; pop
   Result: Just [4]
   ```
2. ```
   Input:  push 1; push 3; dup; apply mul; swap; apply div; push 4; pop
   Result: Nothing (error)
   ```

Task 13: Optimizing Programs (Optional)

Users often write inefficient programs. Optimizers are software tools that automatically spot useless computations and eliminate them, making the program more efficient. Here are some examples of optimizations:

1. A `push` followed by a `pop` is useless.
2. Two consecutive `swap` instructions are useless.
3. A `push n; push m; swap` can be optimized to `push m; push n`.
4. Two consecutive `push n; push n` can be optimized to `push n; dup`.

Write a function that takes a program and optimizes the aforementioned four cases. Hint: add the `deriving Eq` to the end of the data type definitions of `Program`, `Instruction`, and `Op` to be able to compare them. Test your implementation by optimizing the following program:

```
Input:  push 1; push 10; swap; swap; push 20; push 10; swap; pop
Result: push 1; push 10; push 10
```