Assignment 3

(due 18 December 2015, 23:59)

Advanced Functional Programming 2015

1 Too lazy to find other puzzles... **`rally.hs`, `dice.hs`, 5 + 5 points**

In the lectures we mostly saw how to use GHCi, the interpreter built on top of the Glasgow Haskell Compiler. However, as we also saw, GHC can produce executable files. One just needs to define a `main` function with type `IO ()` which will be the entry point of the program as shown below.

```
$ cat hello.hs
main :: IO ()
main = putStrLn "Hello, World!"
$ ghc hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...
$ ./hello
Hello, World!
```

Task

Write two Haskell programs that produce executables that solve the Rally and Dice problems from the previous assignments. The programs should read data from **standard input** and return their results on **standard output**:

```
$ ghc rally.hs
[1 of 1] Compiling Main             ( rally.hs, rally.o )
Linking rally ...
$ cat rally.txt | ./rally
5
3
5
$ ghc dice.hs
[1 of 1] Compiling Main             ( dice.hs, dice.o )
Linking dice ...
$ cat dice.txt | ./dice
2
3
-1
```

The format of `rally.txt` and `dice.txt` is specified on the next page. You can download these sample files from the course's page. Notice that in the sample runs above, these files are piped to the standard input.

Format of input

Rally

The first line of input contains an integer C, specifying the number of test cases that follow ($1 \leq C \leq 10$). Each test case starts with a line containing the two integers A and B separated by a single space, indicating the maximum values of acceleration and braking.
The next line describes the track. It is given by pairs of integers N V indicating a section of N units with speed limit V. The end of the track is indicated by a 0 0 pair.
For the values of the instance data, the limits are the same as in the first assignment.

rally.txt

```
3
30 10
10 100 5 70 3 40 6 100 0 0
40 50
15 100 0 0
40 20
1 50 1 40 1 30 1 20 1 10 1 20 1 30 1 40 1 50 0 0
```

Dice

The first line of input contains an integer C, specifying the number of test cases that follow ($1 \leq C \leq 10$). Each test case starts with a line containing the three integers N, E and D separated by a single space, indicating the number of Nodes and Edges of the graph and the number of Dice in the dice list. The next line contains E node pairs, describing the graph. Each pair indicates an edge between the respective nodes. The last line contains D numbers, which are the values of the dice that you have available.

dice.txt

```
3
3 4 2
1 2 2 1 2 3 3 2
3 5
4 3 1
1 2 2 3 3 4
1
3 2 3
1 2 2 3
4 2 6
```

Limits

There will be at most 30 nodes in the graph and at most 30 dice in the dice list.

Output

For each instance, your programs should print the answer on a new line, as shown on the previous page. The requested answers are the same as in the original assignments: for `rally` instances, the answer is the number of moves needed to *cross* (not just reach!) the finish line and for `dice` instances it is the number of moves needed to reach the winning node or `-1` if this is not possible.

2 Your turn to be lazy... `my_lazy.hs`, **4 points**

Using as input `in` the infinite list of `primes = [2,3,5,7,..]` we can create a new infinite list `out` as output in the following way:

- Start by taking the first element of the `in` list:

  `out = [2]`

- Append the next element of the `in` list and a new copy of `out`:

  `out = [2] ++ [3] ++ [2] = [2,3,2]`

- Repeat:

  `out = [2,3,2] ++ [5] ++ [2,3,2] = [2,3,2,5,2,3,2]`

- Repeat:

  `out = [2,3,2,5,2,3,2] ++ [7] ++ [2,3,2,5,2,3,2] = [2,3,2,5,2,3,2,7,2,3,2,5,2,3,2]`

- ...

Task

Define the function `lazy` which takes two `Integer` indices `from` and `to` and an infinite list `in` and calculates the sum of the elements of the `out` list from index `from` to index `to`.

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load my_lazy.hs
[1 of 1] Compiling Main          ( my_lazy.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t lazy
lazy :: Integer -> Integer -> [Integer] -> Integer
*Main> lazy 1 4 [1..]
7
*Main> lazy 5 26 [1..]
42
*Main> lazy 42 42 [2,4..]
4
*Main> lazy 1000 2000 primes
3681
```

3 Type Classes

showme.hs, 2 points

GHC can employ the C++ preprocessor if given the command-line argument `-cpp`. This allows the use of preprocessor commands like `#ifdef` and `#include`. The following code is therefore acceptable as a Haskell program:

vector.hs

```
module Vector where

type Vector   = [Integer]
data Expr     = V Vector
              | VO VectorOp Expr Expr
              | SO ScalarOp IntExpr Expr
data IntExpr  = I Integer
              | NO NormOp Expr
data VectorOp = Add | Sub | Dot
data ScalarOp = Mul | Div
data NormOp   = NormOne
              | NormInf

#include "showme.hs"
```

Task

Define the contents of `showme.hs` so that given the above `vector.hs` (which you can also find on the course's web page) you can have the following interaction with the interpreter:

Shell

```
$ ghci -cpp
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load vector.hs
[1 of 1] Compiling Vector           ( vector.hs, interpreted )
Ok, modules loaded: Vector.
*Vector> VO Dot (VO Add (V [1,2]) (V [3,4])) (SO Mul (NO NormOne (V [2])) (V [5,6]))
{'dot', {'add', [1,2], [3,4]}, {'mul', {'norm_one', [2]}, [5,6]}}
```

In general, given any part of an expression that is using the above constructors the interpreter should print the equivalent "pretty" version, as it appeared in the specification of the `vector_server` in the first assignment.

Hint

The type class `Show` may have something to do with this question...

4 Hacking in Haskell way

`reverse_hash_solver_(a|b).hs`, 4 points

This problem is the Haskell version of Problem 2 in Assignment 1. As there is clear distinction between concurrent and parallel programming in Haskell, you are free to choose which way you want to solve this problem. You only need to choose one way, either concurrent programming using threads (`forkIO`) or parallel programming using parallel map (`parMap`). The lectures mainly covered parallel programming in Haskell, but you can find all necessary information on concurrent programming in Haskell in the later part of the same tutorial, which is linked on the course web page.

Task

Depending on which approach you decide to take, do either (a) *or* (b) below:

(a) Implement the following function in `reverse_hash_solver_a.hs`:

```
solver_a :: (Int -> Int) -> [Int] -> MVar () -> MVar [Pair] -> Int -> IO ()
solver_a hash inputs signal box schedulers = ...
```

where its arguments are:

(a) `hash` the hashing function used
(b) `inputs` 2^{16} hash values in a list
(c) `signal` an `MVar` set by the main thread to notify the solver to send it all solved cases soon
(d) `box` an `MVar` to hold the solved cases in a list, with each element being a pair `(hash, rev_image)`
(e) `schedulers` the number of schedulers available

Since the actual data is communicated using `MVar`s, the return type is unit captured in an `IO` monad.

The main thread would fork another thread for `solver_a` to run, set the `signal` in 2 seconds, and wait for solved cases in `box` for at most 1 second. Then, the speed, measured in solved cases per time, is calculated, which is used to derive the final score. More about this in the Grading section.

(b) Implement the following function in `reverse_hash_solver_b.hs`:

```
solver_b :: (Int -> Int) -> [Int] -> Int -> [Pair]
solver_b hash inputs schedulers = ...
```

where its arguments are:

(a) `hash` the hashing function used
(b) `inputs` 2^{16} hash values in a list
(c) `schedulers` the number of schedulers available

The return value is the list of solved cases, with each element being a pair `(hash, rev_image)`.

The main thread would run `solver_b` with appropriate arguments, and measure the total amount of time it takes. Then, speed, measured in solved cases per time, is calculated, which is used to derive the final score. More about this in the Grading section.

If you need to import additional modules, put those in the `reverse_hash_imports.hs` file; otherwise leave this file empty.

Grading

The grading scheme for this task is provided in `reverse_hash.hs`. Depending the approach you take, you need to modify it to include the right file and call the corresponding score calculation function, `get_score_a` or `get_score_b`.

```
$ ghc -cpp -O2 reverse_hash.hs -threaded -rtsopts && ./reverse_hash +RTS -N8
For 1 scheduler(s) score: 1.00
For 2 scheduler(s) score: 0.91
For 4 scheduler(s) score: 0.84
For 8 scheduler(s) score: 0.78
```

While grading your program will be run with 1, 2, 4 and 8 schedulers on an (>= 8) core machine. For each case, the score is calculated as the ratio between your solution's speed (solved pairs per time) and the base speed (result from `base_speed`), and the final score for this exercise the sum of these four scores.

Submission instructions

- Each student must send their **own individual submission**.
- For this assignment you must submit a single `afp_assignment3.zip` file at the relevant section in Studentportalen.
- `afp_assignment3.zip` should contain seven files (without any directory structure):
  - The six files requested (`rally.hs`, `dice.hs`, `my_lazy.hs`, `showme.hs`, `reverse_hash_imports.hs`, and one of `reverse_hash_solver_(a|b).hs`) which should conform to the specified interfaces regarding exported functions, handling of input and format of output.
  - A text file named `README.txt` whose first line should be your name. You can include any other comments about your solutions in this file.

Have fun!