Assignment 3

(due 23 December 2013, 23:59)

Advanced Functional Programming 2013

1 Too lazy to find another puzzle... rally.hs, dice.hs, 5 + 5 points

In the lectures we saw how to use GHCi, the interpreter built on top of the Glasgow Haskell Compiler. However, GHC can also create directly 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 executable Haskell programs that solve the Rally and Dice problems from the previous assignments. The programs should read data from **standard input** and return the result 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 and standard output!*

Format of input

Rally

The first line of input contains an integer C, giving 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, giving 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 the infinite list `in = primes = [2,3,5,7,..]` we can create a new infinite list `out` 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 use the C++ preprocessor if given the command-line argument `-cpp`. This enables the use of preprocessor commands like `#ifdef` and `#include`. The following code is therefore acceptable:

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 All languages are equal, but some... reverse_hash.erl, 4 points

You want to find the reverse image for a number of values computed by an unknown integer hashing function, which is implemented in Erlang.

For that purpose you will be given the function and a list of 2^{16} unique hash values. You know that each hash has been generated by an integer between 1 and $2^{27} - 1$.

Task

Try to find the reverse image for as many input values as possible. Read the next section to see how your program should operate.

Grading

Your solution should be scalable. Your submission will be benchmarked within a grading framework which will operate in the following way:

1. It will spawn your program in a new process and start a countdown.
2. When the countdown expires it will send a `finish_up` message to your program.
3. Your program must send a {`reply, List`} message back to the grader within 1s or be disqualified.
4. Your program may also send the {`reply, List`} message at any earlier point.

A sample grading framework is included in `reverse_grading.beam` and exports the following functions:

- `sample_fun()`: Returns a sample hashing function, which expects a value between 1 and $2^{27} - 1$ and returns a value in the same range.
- `sample_inputs(Fun)`: Given a hashing `Fun`, returns 2^{16} hash values generated by random input values from the domain $[1..2^{27} - 1]$.
- `estimate_timeout/0`: Returns an estimation (in milliseconds) of the timeout that would be used, if you were running with 1 scheduler on your current platform trying to reverse values calculated with the function returned by `sample_fun()`.
- `base_score/0`: Returns an estimation about the number of inputs from `sample_fun/0` that should be solved with 1 scheduler on your current platform to get full points.
- `sample_grade()`: Invokes the grader which will eventually spawn a new process and call your main function: `reverse_hash:solve(Fun, Inputs, P, Schedulers)`:
  - `Fun` is the hash function that you are trying to reverse (e.g. the one returned by `sample_fun/0`).
  - `Inputs` is a list of hash values that have been calculated with `Fun`.
  - `P` is the Erlang PID of the grader. After it spawns your process it will wait for a {`reply, List`} message, where `List` should be a list of 2-tuples {`Hash, ReverseImage`}, with `Hash` being one of the values in `Inputs` and `ReverseImage` a value such that `Hash = Fun(ReverseImage)`. At some point the grader will send to the spawned process a `finish_up` message and give you 1 second to reply with your list. If you fail to do so, the grader will disqualify your program.
  - `Schedulers` is the number of usable schedulers that you have available. It can be changed from the default (1 scheduler per core) by passing the `+S` flag when starting the VM.

While grading your program will be run with 1, 2, 4 and 8 schedulers on an 8 core machine. To get all 4 points you have to be able to solve with 1 scheduler at least as many inputs for the `sample_fun/0` as the `base_score/0`, which uses a simple solver. You should then be able to solve in the same time twice as many inputs with 2 schedulers, four times as many inputs with 4 and eight times as many inputs with 8. You can test your solution with `sample_grade()` before submitting.

Submission instructions

- Each student must send their **own individual submission**.
- For this assignment you must submit a single `afp13_assignment3.zip` file at the relevant section in Studentportalen.
- `afp13_assignment3.zip` should contain six files (without any directory structure):
  - The five files requested (`rally.hs`, `dice.hs`, `my_lazy.hs`, `showme.hs` and `reverse_hash.erl`) 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!