Assignment 3

Advanced Functional Programming, 2017
(Avancerad funktionell programmering, 2017)

due 28 December 2017, 23:59

1 Too lazy to find other problems... ({road,dice}.hs, 5 + 5 points)

In the lectures, we mostly saw how to use GHCi, the interactive environment of the GHC distribution. However, as we also saw, GHC also comes with a compiler that produces 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 Road and Dice problems from the two previous assignments. The programs should read data from **standard input** and return their results on **standard output** as shown below:

```
$ ghc road.hs
[1 of 1] Compiling Main             ( road.hs, road.o )
Linking road ...
$ ./road < road1.txt
2
$ ./road < road2.txt
-1
$ ghc dice.hs
[1 of 1] Compiling Main             ( dice.hs, dice.o )
Linking dice ...
$ ./dice < dice.txt
2
3
-1
```

The format of the input files is specified on the next page. You can download these three sample files from the course's page. Notice that in the sample runs above, these files are read from the standard input.

Format of input files

Road

The first line of the file contains three integers separated by spaces: N, specifying the number of lines that follow ($1 \leq N \leq 1\,000\,000$), the length L and the number X ($0 \leq X \leq L$). The rest of the input file consists of N lines, each containing two integers S_k and L_k ($0 \leq S_k < E_k \leq L$) also separated by a space. (The limits are the same as in the first assignment.) The two example input files are shown below.

road1.txt	road2.txt
4 30 6	4 30 1
1 5	1 5
11 27	11 27
2 14	2 14
18 28	18 28

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 of each test case contains D numbers, which are the values of the dice that you have available. As for limits, there will be at most 30 nodes in the graph and at most 30 dice in the dice list.

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

Output

For each test case, your programs should print each answer on a new line, as shown on the previous page. The requested answers are the same as in the previous two assignments. For `road`, the (single) number in the output is the number of days after which the largest consecutive segment of the road that is still unconstructed will not be larger than X. For each `dice` test case, the output 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]`

- ...

GHC can employ the C++ preprocessor if given the command-line argument `-cpp`. This allows the use of preprocessor commands like `#ifdef` and `#include`.

lazy_main.hs

```haskell
import Data.Numbers.Primes (primes)

import Test.HUnit ((@?=))
import Test.Framework (Test, defaultMain)
import Test.Framework.Providers.HUnit (testCase)

#include "my_lazy.hs"

test_lazy :: (Integer, Integer, [Integer], Integer) -> Test
test_lazy (from, to, list, expected) =
  testCase (show from) $
    lazy from to list @?= expected

tests = [
    (1, 4, [1..], 7),
    (5, 26, [1..], 42),
    (1000, 2000, primes, 3681),
    (1, 10000000, primes, 36746404)
  ]

main = defaultMain $ map test_lazy tests
```

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`. Here, the indexing starts from 1, as shown in the example.

Shell

```
$ runghc -cpp lazy_main.hs
1: [OK]
5: [OK]
1000: [OK]
1: [OK]

         Test Cases  Total
 Passed  4           4
 Failed  0           0
 Total   4           4
```

Your `my_lazy.hs` file will be included as shown above but with more test cases.

3 Type Classes (`showme.hs`, 2 points)

Utilizing the preprocessing in Haskell again:

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` you can have the following interaction with the interpreter:

Shell

```
$ ghci -cpp
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
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 Fulltext indexing (`indexing.hs`, 4 points)

Your task is to speed up, as much as possible, a program that performs fulltext indexing and searching of documents by using the power of multicores. The sequential program can be found in course's page, and you will need to download and unpack the set of sample documents:

```
$ wget http://community.haskell.org/~simonmar/par-tutorial-ex1.1-docs.tar.bz2
$ tar xvjf par-tutorial-ex1.1-docs.tar.bz2
```

You can try out the program like this:

```
$ ghc -O indexing.hs && ./indexing docs/*
search (^D to end): <type your search terms here>
```

For example, you could enter `parallel concurrent` and the program would list the filenames of all the documents that contain both those words. To benchmark the program, run it like this:

```
$ echo "parallel concurrent" | ./indexing docs/* +RTS -s
```

The program works by creating a mapping from words to the set of documents that contains that word.

Sample

```
$ # original indexing.hs
$ ulimit -n 4096 && ghc -O indexing.hs  && echo "keyword" | ./indexing docs/* +RTS -s
...
 Total   time    4.034s  (  4.020s elapsed)
...
$ # make indexing.hs faster
$ ghc -O indexing.hs -rtsopts -threaded
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N2 -s
...
  Total   time    3.703s  (  1.972s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N4 -s
...
  Total   time    4.384s  (  1.253s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N8 -s
...
  Total   time    5.211s  (  0.835s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N16 -s
...
  Total   time    9.992s  (  0.827s elapsed)
...
```

Grading

Your final points for this exercise is the sum of the speedup divided by the number of schedulers in each case from $[2, 4, 8, 16]$ cores measured on a machine with $>= 16$ cores. In the sample case, the total points for this exercise would be $4.02 * (1/1.972/2 + 1/1.253/4 + 1/0.835/8 + 1/0.827/16) = 2.73$.

Submission instructions

- Each student must send her/his 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 five files requested (`road.hs`, `dice.hs`, `my_lazy.hs`, `showme.hs`, and `indexing.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!