Functional Programming
Practical 1: Factoring Numbers

Deadline: Week 4

This practical requires only a small amount of actual programming. Its main purpose is to give you a chance to get to know the Haskell environment. However, we would like a brief report on it, in order to evaluate your progress. The report should consist of your well-commented Haskell script, plus your test and experiment results, and answers to each of the exercises below. Your work should be completed, and signed-off by a demonstrator, by the end of your practical session in Week 4.

Copy the file `pa1.lhs` from the course web page at:
`https://www.cs.ox.ac.uk/teaching/materials16-17/fp/`
or from the course practicals directory `/usr/local/practicals/fp/prac1` into your own directory. Answer the exercises below within this script; label each answer clearly, to help the marker.

Factoring numbers

Given any two numbers, we may by a simple and infallible process obtain their product, but it is quite another matter when a large number is given to determine its factors. Can the reader say what two numbers multiplied together will produce the number 8616460799? I think it is unlikely that anyone but myself will ever know.

W. S. Jevons, The Principles of Science, Macmillan, 1874.

This practical is concerned with a simple problem whose solution appears extremely difficult: factoring numbers efficiently. In fact, many of today's measures to protect confidential data and communication depend on the belief that there are no efficient methods for factoring numbers. Of course, there are algorithms that are simple to state, and we will be considering some below, but the problem is that none of them are guaranteed to work within a reasonable time bound. Today's fastest supercomputers and the best algorithms are unable routinely to factor 200 digit numbers. Here is a simple method for finding the smallest prime factor of a positive integer:

```
> factor :: Integer -> (Integer,Integer)
> factor n = factorFrom 2 n

> factorFrom :: Integer -> Integer -> (Integer,Integer)
> factorFrom m n
>   | r == 0     = (m,q)
>   | n <= m*m   = (n,1)
>   | otherwise = factorFrom (m+1) n
>   where (q,r) = divMod n m
```

The expression `factorFrom m n` returns a pair of integers (a, b), where $a \times b = n$ and a is the least such number with $m \leq a \leq \sqrt{n}$, if such an a exists, otherwise $a = n$. The value of `divMod n m` is a pair (q, r), where q is the quotient of n divided by m (i.e. the largest number of times that m can be subtracted from n), and r is the remainder. Thus $q = \lfloor n/m \rfloor$, where $\lfloor x \rfloor$ is the largest integer no bigger than x, and $r = n - q \times m$. The function `divMod` is provided automatically as a library function in the Haskell Standard Prelude (a library module of standard datatypes, classes, and functions which is imported by default in any Haskell module,
`http://hackage.haskell.org/package/base-4.7.0.1/docs/Prelude.html`).

Exercise 1. Explain why the test `n <= m*m` is used to terminate the search rather than the more obvious `n == m`. Can the first two lines in the definition of `factorFrom` be interchanged? Approximately how many recursive calls are needed to evaluate `factor n` in the worst case (as a function of n)? ♢

Exercise 2. *Without* trying it on the computer, determine the values of `factor 0` and `factor 1`. ♢

Exercise 3. You will find the functions given above, and others, in the script file `pa1.lhs`. Run GHCi (the `ghci` command) and check your answer to Exercise 2. ♢

The above definition of `factorFrom` is not quite how the algorithm is usually coded in textbooks. Instead, the following version is given:

```
> factorFrom1 :: Integer -> Integer -> (Integer,Integer)
> factorFrom1 m n
>    | r == 0      = (m,q)
>    | q <= m      = (n,1)
>    | otherwise   = factorFrom1 (m+1) n
>    where (q,r) = divMod n m
```

Exercise 4. Explain informally why this variation defines the same function, and why it is more efficient. ♢

Exercise 5. The real inefficiency with either of these simple methods is that all numbers greater than 1 are used as trial divisors. It would be better if, for example, 2 was treated as a special case, and the odd numbers 3, 5, 7, . . . were used as trial divisors. Code this version (call it `factor2`) and add it to your copy of the script file. You will need a subsidiary function analogous to `factorFrom1` (call it `factorFrom2`).
How much more efficient would you expect this version to be? ♢

Exercise 6. Test your function, `factorFrom2`, and paste some of the test results into your file as part of your final report. (*This is something you should do implicitly for every exercise that involves writing code from now on.*) ♢

Exercise 7. It would be more efficient still to throw away multiples of 3 as well; after treating both 2 and 3 as special cases, the list 5, 7, 11, 13, 17, . . . of odd numbers increasing alternately by 2 and 4 could be used as trial divisors. Implement this idea by coding functions `factor3` and `factorFrom3`; the latter function should take an extra argument `s` which alternates in value between 2 and 4 (Hint: `6-s` does this switching). ♢

Exercise 8. The logical extension of these ideas is to use only prime numbers as trial divisors. Fine in principle, but what is the downside of such a scheme? ♢

Finding the smallest prime factor is all well and good, but we want to continue the process by finding all prime factors. To do this we will need lists, which will only be introduced properly later on in the course. Here is the algorithm:

```
> primeFactors :: Integer -> [Integer]
> primeFactors n = factorsFrom 2 n

> factorsFrom :: Integer -> Integer -> [Integer]
> factorsFrom m n =
>     if n == 1 then [] else p:factorsFrom p q
>     where (p,q) = factorFrom m n
```

The type `[Integer]` is the type of lists whose elements are integers. In the definition of `factorsFrom` the symbol `[]` denotes the empty list, while `p:ps` describes the list whose first element is `p` and whose remaining elements are those in the list `ps`.

Exercise 9. The function `primeFactors` is also provided in the script. Try the function out on some suitable examples. Rewrite the function—call it `primeFactors2`—to make use of the superior `factorFrom3` described above. ◇

In GHCi you can use the command:

```
:set +s
```

to set the system to print the approximate time and memory space used to simplify each expression.

Exercise 10. Perform some experiments to compare the speed of `primeFactors` and `primeFactors2`. Paste a small selection of the results into your file for eventual submission as part of your write-up. How long does it take to solve Jevons' problem? ◇

Optional exercises

The following questions are not compulsory, but are designed for those wanting to pursue these ideas further.
Another approach to factoring was used by Fermat in 1643. It is more suited to finding large factors than small ones.
Assume n is an odd number and that $n = u \times v$. It follows that $n = x^2 - y^2$, where $x = (u+v)/2$ and $y = (v-u)/2$ are both whole numbers (why?). Fermat's method consists of systematically searching for the smallest value of x for which there is a y such that

$$x^2 - y^2 = n \quad \text{and} \quad 0 \leq y < x\,.$$

Exercise 11. What is the smallest possible value of x, that is, the value we should begin the search with?
Suppose that at some stage the search has been narrowed to $x \geq p$ and $y \geq q$. Let $r = p^2 - q^2 - n$. If $r = 0$, then we are done. If $r < 0$, how should we change p or q? And how do we change r to maintain $r = p^2 - q^2 - n$? And what if $r > 0$?
Why is this method guaranteed to terminate for all odd n?
Design a function `search` so that `search p q r` carries out the search.
Hence design a function `fermat` for returning two factors of a given odd number. You will need the following function for computing integer square roots:

```
> isqrt :: Integer -> Integer
> isqrt = truncate . sqrt . fromInteger
```

◇

Exercise 12. Use `fermat` to solve Jevons's problem, and to find the factors of 1963272347809. ◇

Exercise 13. (Very Optional and For Amusement Only) Here is a secret message:

```
[772653220544,1915678282609,276920912360,
 1140547573312,317556866509,665998855449,
 1882188522238,653343178985,539272906038,
 1331719140974,1881623288858,1221070877456,97336]
```

It was computed by breaking the text into blocks of 5 characters, and coding each block as an integer. The resulting sequence $[x_1, x_2, \ldots, x_k]$ was then encrypted as $[y_1, y_2, \ldots, y_k]$, where $y_i = x_i^3 \bmod n$, where

```
> n = 1963272347809
```

The file `encrypt.lhs` in the practicals directory contains the code to do the encryption.
Can you decipher the secret message? Hint: try searching the web or a library to find out about RSA encryption. [Warning: this exercise is likely to be very time consuming] $\diamondsuit$

Ani Calinescu, October 2016
(with thanks to Peter Jeavons, Gavin Lowe, Richard Bird and William Stanley Jevons)