Imperial College of Science, Technology and Medicine Computer Science (CS) / Software Engineering (SE) Department of Computing	University of London BEng and MEng Examinations Part I Integrated Laboratory Course
Laboratory work is a continuously assessed part of the examinations and is a required part of the degree assessment. Laboratory work must be handed in for marking by the due date. Late submissions may not be marked.	

Exercise: 2	Working: Individual
Title: Recursion in Haskell	
Issue date: 13th October 2003	Due date: 20th October 2003
System: Linux	Language: Haskell

Aim

- To provide practice at desigining simple recursive functions in Haskell
- To experiment with local (nested) definitions and Haskell's scope rules

The Problem

You should write the following functions in a script **Recursion.hs**.

- **sine1, sine2** each of which computes an approximation to the sine of a given number using two different methods
- **pie1, pie2** each of which computes an approximation to π again using two different methods equation solver.
- **isPrime** which returns **True** if a given number is prime; **False** otherwise
- **nextPrime** which finds the smallest prime larger than a given non-negative integer.

Submit by Monday 20th October 2003

What To Do

- Your script should be called **Recursion.hs**.

Approximating $sin(x)$ for a given x

- Write a function **sine1** declared as follows:

  ```
  sine1 :: Double -> Integer -> Double
  sine1 x n = ...
  ```

  that will compute an approximation to the sine of a given number. The function should use recursion to encode the following definition of $sin(x)$:

  $$sin(x) \;=\; x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

  The number of terms that should be used is given by the second argument of **sine1** such that for example, **sine1 2 1** gives 2, **sine1 2 2** gives $2 - 2^3/3!$ and so on. Q: What should the base case be? To get this right look carefully at the precondition. Note: the function is defined in terms of the factorial function $n!$ which can be defined in Haskell as follows:

  ```
  fact :: Int -> Int
  -- pre: n>=0
  fact n
    = if n == 0 then 1 else n * fact ( n - 1 )
  ```

  This works fine, but only up to integer values of 16. To see this try typing **fact 16** and then **fact 17** at the Haskell prompt. Q: What is going on? When using this function to define **sine1** you have to be aware that **n** in **in sine1** can be at most 8 to avoid the problem. A quick fix is to change the type **Int** to **Integer** which handles arbitrary-precision integer arithmetic—you might like to try this if you have time. You may use either **Int** or **Integer** types in your submission.

- Now write a function **sine2** with the same type:

  ```
  sine2 :: Double -> Integer -> Double
  sine2 x n = ...
  ```

  that again approximates the sine of a given number, this time using the definition:

  $$sin(x) \;=\; x \prod_{i=1}^{\infty} \left(1 - \frac{x^2}{i^2\pi^2}\right)$$

  The second argument of **sine2** represents the number of terms in the product that should be used. What is the base case now, i.e. what should happen if **n** is 0?

- Write a function **pie1** declared as:

  ```
  -- pre: n>=0
  pie1 :: Int -> Double
  pie1 n = ...
  ```

that uses the following equivalence to approximate π:

$$\frac{\pi^2}{8} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + ...$$

Use a local function to compute the (truncated) series. Hint: Note that you need to sum the terms of a series and then scale the result to estimate π. Use a local function to compute the series.

- (Harder) Now define another function **pie2**:

  ```
  -- pre: n>=0
  pie2 :: Int -> Double
  pie2 n = ...
  ```

  that again approximates π, this time using the formula:

  $$\frac{2}{\pi} = \sqrt{\frac{1}{2}}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}}...$$

  A naive way to calculate the term on the right is to compute each of the sub-terms $\sqrt{\frac{1}{2}}$, $\sqrt{\frac{1}{2}}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}$ etc. separately and then multiply them together. A much better way is to notice that each sub-term can be used to define the next, i.e. without having to recompute it. An extra 0.5 marks is reserved for this 'smarter' solution.

Generating prime numbers

- Write a function **isPrime** declared as follows:

  ```
  isPrime :: Int -> Bool
  isPrime n = ...
  ```

  that returns **True** if a given non-negative integer is prime; **False** otherwise. A non-negative integer is *prime* if and only if it is at least 2 and has no divisors except *one* and *itself*. (Put another way: a prime is an integer $n > 1$ with exactly two factors.) For odd numbers greater than 2 the easy way to define the function is to check whether it is divisible by any of the numbers 3, 5, 7... Q: what's the largest number you need to check?

- Write a function **nextPrime** declared as follows:

  ```
  nextPrime :: Int -> Int
  nextPrime n = ...
  ```

Reminder: Haskell has two built-in functions for integer division and remainder, called *div* and *mod* respectively. Thus:

$$\begin{aligned} 7 \text{ `}div\text{` } 3 &= 2 \\ 7 \text{ `}mod\text{` } 3 &= 1 \end{aligned}$$

If **m** can be divided exactly by **n** then m ‘*mod*‘ $n = 0$.
You may find these operators useful in determining if an integer is prime.

- When you have **nextPrime** working, try this at the Haskell prompt:

  iterate nextPrime 2

  Sit back and watch the infinite list of primes unfold! (Type control-C to interrupt.)

  The built-in Haskell function **iterate** is a *higher-order* function producing a *list*. You will learn more about lists and higher-order functions later in your course.

Unassessed

- If you make **isPrime** a local funcion (i.e. part of a where clause within nextPrime) can you simplify your (now local) definition of isPrime in any way?

Submission

- As always **test your program against a wide range of inputs**. Make sure that you test your programs as you write them.
- Make sure that your script is in the file **Recursion.hs**, and that your functions are called **sine1**, **sine2**, **pie1**, **pie2** and **nextPrime**, then submit your program by typing

  submit 2

  at your Unix prompt.

Assessment

```
sine1                               1.0
sine2                               1.0
pie1                                1.0
pie2                                1.0
nextPrime                           1.0

Design, Style and Readability       5.0

Total                              10.0
```