Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	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: 3	Working: Individual
Title: Lists and Fractals in Haskell	
Issue date: 20th October 2003	Due date: 27th October 2003
System: Linux	Language: Haskell

Aim

- To provide experience with simple list handling and tuples in Haskell.
- To use Haskell type synonyms to aid program readability.
- To introduce fractals and basic computer graphics in Haskell.
- To enable you to visualise the execution of a recursive program.

The Problem

You should write the following functions in a script **Fractals.hs** that allow you to draw the Sierpinski triangle fractal.

- **swTriangle** which given the co-ordinates of the bottom left-hand corner and the length of two of the sides returns a list of the co-ordinates of a "south west" triangle.
- **eqTriangle** which given the co-ordinates of the centre and the length of the sides and a boolean to indicate the orientation returns a list of the co-ordinates of the triangle.
- **makePolygon** which given the number of sides returns the list of co-ordinates of a regular polygon of size n.
- **colour** which associates a particular colour with a polygon.
- **sierpinski** which given the co-ordinates of the bottom left-hand corner, order and colour returns a list of coloured polygons which allows a Sierpinski triangle to be drawn.
- **makeLines** which given a ColouredPolygon will return an alternative representation as a list of lines.

Submit by Monday 27th Oct 2003

Background

Fractals

Fractals are recursive geometric structures that start small and grow arbitrarily big by repeated application of simple rules. The size and complexity of the final structure is determined by the number of times the rules are applied. The fractal structures in this exercise are built simply by drawing polygons of different size and colour using straightforward recursive algorithms. There are many other types of fractal structure; you'll be seeing a different type next week.

Graphics

This exercise is not actually about computer graphics! However, having built the representation of a fractal structure as a Haskell data structure, it is useful and fun to be able to visualise it. To this end a Haskell module called ICGraphics has been written for you to allow you to display your fractals in a graphics window. You do not need to understand how this works - you simply have to pass it a suitable data structure and it will do the rest.

To display the list of lines in a graphics window, you'll need to import the predefined module **ICGraphics**. In addition to use the **Color** (see below) enumerated type will also need to import the module **GraphicsUtils**. To do this type

```
import ICGraphics
import GraphicsUtils
```

at the top of your program. The graphics module provides two drawing functions:

```
drawPolygons :: [ ColouredPolygon ] -> ...
```

which takes a list of coloured polygons and draws them, in-filled with the specified colour, in a separate graphics window and

```
drawLines :: [ ColouredLine ] -> ...
```

which takes a list of coloured lines as an argument and draws them in the window. The functions do return results, as you would expect, but as you are going to be using these functions at the topmost level it is not important to know the result types. To understand these types fully requires a discussion of Haskell's *monadic I/O* which will only be covered in the optional "advanced" tutorials accompanying this course.

Note that the drawing functions will automatically scale the image so that it fits centrally in the window, regardless of the absolute magnitude of the lines or polygons being drawn. Coloured lines and coloured polygons, and the supporting types, are provided for you and are defined as follows:

```
type Vertex = ( Float, Float )

type ColouredLine = ( Vertex, Vertex, Color, Int )
```

```
type Polygon = [ Vertex ]

type ColouredPolygon = ( Polygon, Color )
```

A `Vertex` specifies a point in the Euclidean plane; a `ColouredLine` is a tuple comprising the start and end points of the line, its `Color`, and its thickness (which should be set to 1 in the assessed part); a `Polygon` is a list of vertices with the rule that the first and last entries should be the same (this just simplifies some of the functions you have to write); a `ColouredPolygon` is a pair comprising a `Polygon` and its `Color`.

So, what is a Color? You may have guessed from the spelling that we're going to use a predefined data type designed by an American! `Color` is a simple enumerated data type defined in the module `GraphicsUtils` as follows:

```
data Color = Black | Blue    | Green  | Cyan
           | Red   | Magenta | Yellow | White
```

Sierpinski's Triangle

The Sierpinski triangle is a simple fractal image formulated by the Polish mathematician Waclaw Sierpinski. It works by drawing a number of non-overlapping triangles of fixed size (1 in this case) at locations that are computed recursively. Figure 1 below shows the Sierpinski triangles of "orders" 0, 1 and 2 when the base triangle size is set to 1. You can see that to draw the Sierpinski triangle of order $n > 0$ anchored at (x, y) (the botom left-hand corner) you simply draw Sierpinski triangles of order $n - 1$ anchored at $(x, y), (x, y + s)$ and $(x + s, y)$ where $s = 2^{n-1}$. A Sierpinski triangle of order 0 is simply a south-west triangle whose short sides are both of length 1 (base case).

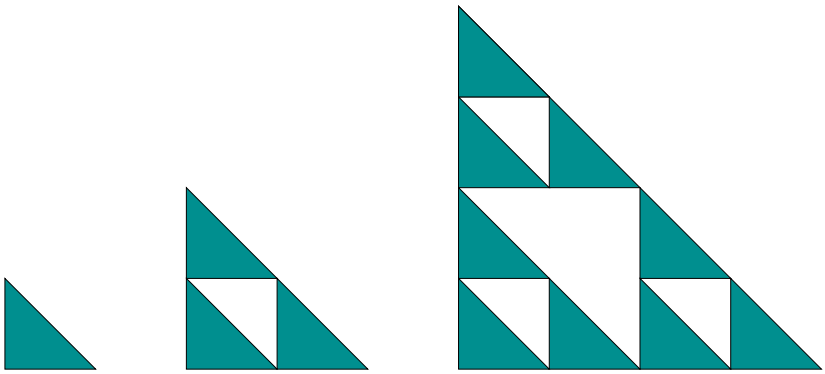

Figure 1: Sierpinski triangles of order 0, 1 and 2

What To Do

- You should copy the files **ICGraphics.hs** and skeleton file **Fractals.hs** using the command:

  ```
  exercise 3
  ```

- Write two functions for building polygon representations of both south-west right-angled triangles and equilateral triangles:

```
swTriangle :: Vertex -> Float -> Polygon
-- Returns a polygon that represents a south west triangle

eqTriangle :: Vertex -> Float -> Bool -> Polygon
-- Returns a polygon that represents an equilateral triangle
-- (boolean: True = downward-pointing, False = upward-pointing)
```

  The co-ordinates of a south-west triangle should be obvious from the way it is aligned with the x and y axes.

  For the equilateral triangle we'll assume the orientation to be a `Bool` where `True` means "point down" and `False` means "point up".

  The co-ordinates of a *downward pointing* equilateral triangle with centre (x, y) and side length s are:
  $(x - \frac{1}{2}s, y + \frac{1}{2\sqrt{3}}s)$
  $(x + \frac{1}{2}s, y + \frac{1}{2\sqrt{3}}s)$
  $(x, y - \frac{1}{\sqrt{3}}s)$

  The co-ordinates of an *upward pointing* equilateral triangle with centre (x, y) and side length s are:
  $(x - \frac{1}{2}s, y - \frac{1}{2\sqrt{3}}s)$
  $(x + \frac{1}{2}s, y - \frac{1}{2\sqrt{3}}s)$
  $(x, y + \frac{1}{\sqrt{3}}s)$

  (The unassessed part later contains a further brief discussion of how the co-ordinates of an equilateral triangle can be derived, if you are interested.)

- Write a function **makePolygon** that given the number of sides returns the co-ordinates of the regular polygon whose first (and therefore last) co-ordinate is $(0, 0)$. The polygon should traced in an anticlockwise direction, starting with $(0, 0), (1, 0)$....

```
makePolygon :: Int -> Polygon
-- Returns a polygon that represents regular polygon of size n
-- The y co-ordinates of the polygon are all non-negative, with the
-- base running from (0,0) to (1,0).
```

  Note: a movement of length 1 if you are facing angle θ degrees anticlockwise from 3 o'clock would be $(cos\frac{\pi\theta}{180}, sin\frac{\pi\theta}{180})$ in (x, y) terms, and $\theta = 0$ initially.

- Write a function **colour** that will associate a colour with a given polygon:

- ```
  colour :: Polygon -> Color -> ColouredPolygon
  -- Returns a ColouredPolygon with associated colour
  ```

  Note: you can test these functions by drawing coloured triangles using the functions provided in the module **ICGraphics**. As an example, try:
  ```

```
Main> drawPolygons [ colour ( swTriangle (0,0) 2 ) Green,
                     colour ( eqTriangle (0,0) 1 True ) Yellow ]
```

This will produce the image in a separate window to the Hugs system. To close this window and return to Hugs simply hit any key whilst the window is selected.

- Now write a function **sierpinski** that will produce a representation of a coloured Sierpinski triangle of a given position, order and colour.

```
sierpinski :: Vertex -> Int -> Color -> [ ColouredPolygon ]
-- Returns a list of Coloured Polygons that represent a
-- sierpinski triangle
```

Note: again you can test this function using **drawPolygons**. For example, try this:

```
Main> drawPolygons ( sierpinski (0,0) 2 Yellow )
```

This should produce the image shown on the right of Figure 1 with the triangles shown in Yellow.

- Write a function **makeLines** that will take a list of coloured polygons and produce an alternative representation as a list of coloured lines:

```
makeLines :: [ ColouredPolygon ] -> [ ColouredLine ]
-- Converts a list of ColouredPolygon into a list of ColouredLine
```

Note: you can test this using the drawLines function provided in module **ICGraphics**. For example, to get the above Sierpinski triangle of order 2, this time with the triangles drawn in outline, try:

```
Main> drawLines ( makeLines ( sierpinski (0,0) 2 Yellow ) )
```

Unassessed

As an optional unassessed exercise you can write a function that uses your **eqTriangle** function to draw snowflakes.

Snowflake

There are many snowflake fractals in the literature - you'll see another one next week. The one you'll build here is produced by drawing equilateral triangles of different sizes and colours, rather than south-west triangles of fixed size and colour. In what follows, the centre of an equilateral triangle is taken to be the point equidistant from the three vertices—you can show that this is two thirds of the way down the line connecting any vertex with the mid-point of the side opposing it (try proving it).

To draw a snowflake of order $n > 0$, centred at location (x, y) and orientation p you first draw an equilateral triangle with mid-point (x, y), sides of length $m = 3^n$ and orientation p. Here we'll assume the orientation to be a `Bool` where p =`True` means "point down" and p =`False` means "point up" and we'll start with a triangle point down (i.e. p=`True`) initially. Now you draw six snowflakes of order $n - 1$ with orientation and location as follows:

1. Point down at location $(x - r, y + d)$
2. Point down at location $(x + r, y + d)$
3. Point down at location $(x, y - 2d)$
4. Point up at location $(x - r, y - d)$
5. Point up at location $(x + r, y - d)$
6. Point up at location $(x, y + 2d)$

The offsets r and d are chosen for aesthetic reasons and here we choose them so that the smaller snowflakes are centred three quarters of the distance between (x, y) and the six points of the "Star of David" which would be formed had we drawn two triangles of size m centred at (x, y) with opposite orientation, instead of one. Because an equilateral triangle with sides of length m has height $m\sqrt{(\frac{3}{4})}$ (prove it) we therefore use $d = \frac{m\sqrt{3}}{8}$ and $r = \frac{3m}{8}$ where $m = 3^n$. Check these for yourself. To draw a snowflake of order 0 we just draw the triangle (in the base case of size $3^0 = 1$). Figure 2 below shows snowflakes of order 0 and 1. As you can see, it is easier to see what's going on if each triangle size is drawn in a different colour. This also makes the final image more attractive!

Figure 2: Snowflakes of order 0 and 1

- If you do the unassessed exercise write your code in a function that will produce a representation of a snowflake fractal of a given order and starting colour.

```
snowflake :: Vertex -> Int -> Color -> [ ColouredPolygon ]
-- Returns a list of Coloured Polygons that represent a
-- snowflake
```

Note: You need to define a helper function, **snowflake'** say, that will draw a snowflake of a given order, in a given location with a given colour and orientation. The type is identical to that of snowflake except for the orientation parameter – this tells you how to represent the top-level triangle prior to building the six sub-flakes (if required). You'll need this as different sub-flakes start off in different orientations.

- To draw the snowflake fractal you're going to draw each triangle of a given size in a different colour. To get from one colour to the next, write a function **nextColour** that will advance throught the colours in the order they are defined in the Color data type, for example `nextColour Green` should yield `Cyan`.

```
nextColour :: Color -> Color
-- Returns next colour from enumerated type, wrapping around from
-- last to first
```

Submission

- As usual use the submit command, **submit 3**, in the appropriate directory. Make sure your file and the functions within it are correctly named.

Assessment

`swTriangle`	0.5
`eqTriangle`	0.5
`makePolygon`	0.5
`colour`	0.5
`sierpinski`	1.5
`makeLines`	1.5
`Design, Style and Readability`	5.0
`Total`	10.0