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: 4	Working: Individual
Title: L-Systems in Haskell	
Issue date: 27th Oct 2003	Due date: 3rd Nov 2003
System: Linux	Language: Haskell

Aim

- To provide further experience with list handling and tuples in Haskell with emphasis on higher-order functions.
- To introduce the role of stacks and state in recursive calculations.
- To introduce turtle graphics and L-Systems.

The Problem

You should write and submit the following functions in a script **LSystems.hs**. The functions when combined allow the user to construct the graphically interpretable objects known as L-Systems, starting with an input string and a set of rules and finishing with a list of lines to be drawn in a graphics window.

As in the previous exercise two small graphics modules **ICGraphics.hs** and **GraphicsUtils** which provide the function **drawLines** have been included so that you can draw out the object produced by your program. A skeleton program **LSystems.hs** containing some initial base values and rewrite rules (see section "Rewrite Rules") has been provided. You should copy the files **LSystems.hs** and **ICGraphics.hs** using the command:

```
exercise 4
```

You do not need to understand how this function is implemented and you can treat it as a *black box* which provides a graphical function, although the text is available for study if you want.

To make the functions more readable you should use the type synonyms given below in the "What To Do" section.

- **angle**, **base**, **rules** Three simple functions used for extracting fields needed for constructing LSystems from the **system** table.
- **lookupChar** An auxiliary function (used by **expandOne**) which finds a character in a lookup table which associates characters and strings and returns the associated string.
- **expandOne** This uses **lookupChar** to return a string formed by replacing each character in a given string with the string associated with it in a lookup table.
- **expand** This produces an L-System in the form of a string by applying **expandOne** a specified number of times on a given initial string and lookup table.
- **move** An auxiliary function used by **trace** to calculate the new position of a turtle given a move instruction. The move instruction can be either to turn left or right or to move forward in the present direction. The current position is an *(x, y)* co-ordinate. The result is a new position expressed as an *(x, y)* co-ordinate.
- **trace** This takes a string of *turtle* commands and converts them into a list of lines that can be subsequently drawn on the screen by **drawLines**. (A string could be produced for example by **expand**, **expandOne** and **mapper** discussed in detail later on.) A line is expressed as a pair of *(x, y)* co-ordinates. It uses the auxiliary function **move** to process each individual character.

Submit by Monday 3rd Nov 2003

Background

L-Systems

L-systems are named after the biologist Aristid Lindenmayer. They work by repeatedly replacing each item in a sequence according to a fixed set of rewrite *rules*. Starting from an initial base string (a seed) the sequence grows in size as the rewrites are applied over and over. By careful design of the rewrite rules and by interpreting the items in the sequence as movements of 'pen on paper' (often implemented in hardware by a *turtle* - a device on wheels with a pen underneath) a sequence can be used to construct pictures of strange and beautiful mathematical objects. A drawing function has been provided for you so that you can display these pictures on your computer screen.

A modest extension to the very simple systems introduced here enables elaborate branching structures to be described. An optional unassessed exercise invites you to expand the program so that it can draw complex plant-like structures. This may involve arbitrary extensions to the basic program and/or drawing function provided.

There are many sources of information on L-systems on the web. See, for example, **http://pages.cpsc.ucalgary.ca/~jacob/lsys.html**

Turtle Graphics

A graphics *turtle* can be thought of as a simple robot which moves around on a large sheet of paper according to a set of basic commands. The robot has a pen built into it which, in

general, can be raised or lowered onto the paper. If the robot moves whilst the pen is 'down' the movement will trace a line on the paper. This exercise uses a simple imaginary robot turtle whose pen is always in the 'down' position and which moves around using just three commands encoded as characters:

'F' Moves the turtle forward a fixed distance in the direction in which the turtle is facing.

'L' Rotates the robot by a given angle anticlockwise (i.e. to the left) on the spot.

'R' Rotates the robot by a given angle clockwise (i.e. to the right) on the spot.

The movement distance will be fixed here arbitrarily at 1 unit; the angle of rotation will be a property of a particular graphical interpretation of an L-System.

Rewrite Rules

In this exercise each turtle command will be a character ('F', 'L' or 'R') and a sequence of commands will be a string. An L-System again works with strings but the characters within the strings (we'll call them 'symbols') can be arbitrary. A rewrite rule is a mapping from characters to strings and each will be represented as a **(Char, String)** pair. A set of rules will be represented as a table (a list of such pairs) thus:

```
type Rule =  ( Char, String )
type Rules = [ Rule ]
```

An L-System consists of a base string (or axiom), a set of rewrite rules and (when interpreted graphically) an associated rotation angle used to drive the turtle. In principle any characters can appear in a string, although before interpreting a string as a turtle command they must be mapped to the characters 'F', 'R' or 'L' in some way. This is done using **mapper** (see below).

To help you get started eight L-Systems have been prepared for you. Each system is stored as a triple, (angle, base, rules). Each function takes an integer i as a parameter and returns the angle/base/rules value for the i^{th} L-system, $1 \leq i \leq 8$. They are available in the skeleton program **LSystems.hs**.

```
-- Cross
system 1  = ( 90, "M-M-M-M", [ ( 'M', "M-M+M+MM-M-M+M" ),
                               ( '+', "+" ),
                               ( '-', "-" )  ] )
-- Triangle
system 2  = ( 90, "-M",      [ ( 'M', "M+M-M-M+M" ),
                               ( '+', "+" ),
                               ( '-', "-" )  ] )

-- Arrowhead
system 3  = ( 60, "N",       [ ( 'M', "N+M+N" ),
                               ( 'N', "M-N-M" ),
                               ( '+', "+" ),
                               ( '-', "-" )  ] )
```

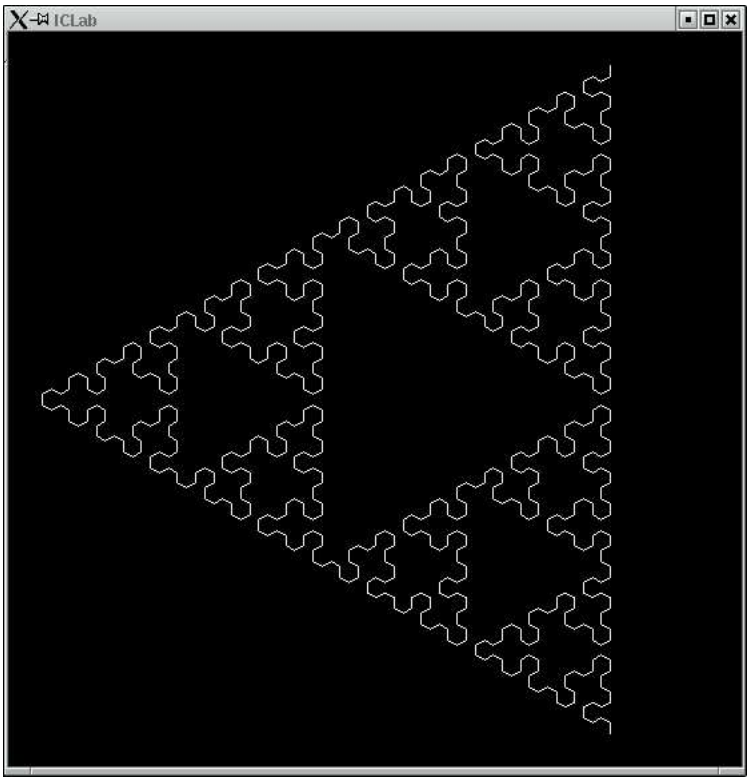
```
-- Peano-Gosper
system 4  = ( 60, "M",       [ ( 'M', "M+N++N-M--MM-N+" ),
                               ( 'N', "-M+NN++N+M--M-N" ),
                               ( '+', "+" ),
                               ( '-', "-" )  ] )

-- Dragon
system 5 = ( 45, "MX",       [ ( 'M', "A" ),
                               ( 'X', "+MX--MY+" ),
                               ( 'Y', "-MX++MY-" ),
                               ( 'A', "A" ),
                               ( '+', "+" ),
                               ( '-', "-" ) ] )

-- Snowflake
system 6 = ( 60, "M--M--M",  [ ( 'M', "M+M--M+M" ),
                               ( '+', "+" ),
                               ( '-', "-" ) ] )

-- Tree
system 7 = ( 45, "M",        [ ( 'M', "N[-M][+M][NM]"),
                               ( 'N', "NN"),
                               ( '[', "[" ),
                               ( ']', "]" ),
                               ( '+', "+" ),
                               ( '-', "-" ) ] )

-- Bush
system 8  = ( 22.5, "X",     [ ( 'X', "M-[[X]+X]+M[+MX]-X" ),
                               ( 'M', "MM" ),
                               ( '[', "[" ),
                               ( ']', "]" ),
                               ( '+', "+" ),
                               ( '-', "-" ) ] )
```

As a simple example, suppose we work with L-system 3. The base value is the string "N". After the first rewrite the 'N' will be replaced by the string "M-N-M", using the rewrite rules obtained by calling **rules 3**. If the string is rewritten again, each 'N' will be replaced likewise, each 'M' will be replaced by the string "N+M+N" and the '+'s and '-'s will be replaced by themselves. Thus, after a second application of the rules, the string becomes "N+M+N-M-N-M-N+M+N", and so on.

Notice that there are *different* rules for rewriting 'N's and 'M's, but both will be ultimately interpreted as 'move' commands for the turtle. The reason for having multiple move commands is thus not to control the turtle in more elaborate ways, but to enable more interesting rewrite systems to be expressed.

If you apply the rules for L-System 3 a total of six times the command string has 1457 characters! If the 'M's and 'N's in this string are both mapped to turtle command 'F' and '+' and '-' to 'L' and 'R' respectively (i.e. using **mapper**) the turtle will trace out the following picture:

L-System 3 after six rewritings



What To Do

Note that credit will be awarded for the appropriate use of Haskell's prelude functions and higher order functions in particular.

- Firstly define three functions **angle**, **base** and **rules**:

```
angle :: Int -> Float
-- Extract angle from system
base  :: Int -> String
-- Extract base from system
rules :: Int -> Rules
-- Extract rules from system
```

- Now define a function **lookupChar**:

```
--
-- Look up command character in rule table
-- pre: the character has a binding in the table
--
lookupChar :: Char -> Rules -> String
```

For example:

```
lookupChar 'M' ( rules 4 )
"M+N++N-M--MM-N+"
lookupChar '+' ( rules 2 )
"+"
```

- Now use **lookupChar** to write a function **expandOne**:

```
--
-- Expand command string s once using rule table r
--
expandOne :: Rules ->  String  -> String
```

For example:

```
expandOne ( rules 2 ) ( base 2 )
"-M+M-M-M+M"
```

- Write a function **expand**:

```
--
-- Expand command string s n times using rule table r
--
expand :: Rules -> String -> Int -> String
```

For example:

```
expand ( rules 3 ) ( base 3 ) 2
"N+M+N-M-N-M-N+M+N"
```

- Write a function **move**:

```
--
-- Move turtle: 'F' moves distance 1, 'L', 'R' rotate left and right
-- by given angle
--
move :: Char -> TurtleState -> Float -> TurtleState
```

The state of the turtle is given by its current (x, y) co-ordinate and its orientation (the orientation is measured in degrees, anticlockwise from the positive x-axis). **Note:** you must declare these type synonyms and headers exactly as we give them. The turtle state is given thus:

```
type Vertex = ( Float, Float )

type TurtleState = ( Vertex, Float )
```

For example:

```
move 'L' ( ( 100, 100 ), 90 ) 90
( ( 100.0, 100.0 ), 180.0 )

move 'F' ( ( 50, 50 ), 60 ) 60
( ( 50.5, 50.866 ), 60.0 )

move 'F' ( ( -25, 180 ), 180 ) 45
( ( -26.0, 180.0 ), 180.0 )
```

- Write a function **trace**:

```
--
-- Trace lines drawn by turtle using given colour following commands in cs
-- assuming given angle of rotation
--
-- Includes the stack - used for branching systems
--
trace :: String -> Float -> Color -> [ ColouredLine ]
```

where **ColouredLine** defines the start and end points of a line in the Euclidean plane, along with its colour and thickness:

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

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 $\theta = 90$ initially, L=add to θ, R=subtract from θ. Initially the turtle is at $(0, 0)$ and angle $\theta = 90$.

To make this work you need to keep track of the state of the turtle as it is moved and rotated. You will therefore need a helper function (either a new function at the top level, or a function defined within a **where**) which takes the current turtle state as a parameter and which recurses with the appropriately modified state.

Recall that turtles have a single 'move forward' command ('F') whilst the strings generated by the rewrite rules here use two symbols 'M' and 'N' to mean the same thing, namely forwards. Also, the symbols '+' and '-' are used to mean 'move left' and 'move right' respectively whereas the corresponding turtle commands are 'L' and 'R'. Before a

string is passed to **trace**, therefore, the symbols must be mapped into turtle commands. Fortunately, you have just built such a function which can perform the mapping–it is called **expandOne**! To use this to perform the mapping we need to define a (new) object of type **Rules** (call it **mapper**, say) that maps 'M' and 'N' to command 'F' etc and call **expandOne** accordingly. To save you time, we have provided this **mapper** function in the skeleton. For example:

```
expand ( rules 3 ) ( base 3 ) 3
"M-N-M+N+M+N+M-N-M-N+M+N-M-N-M-N+M+N-M-N-M+N+M+N+M-N-M"

expandOne mapper ( expand ( rules 3 ) ( base 3 ) 3 )
"FRFRFLFLFLFLFRFRFRFLFLFRFRFRFRFLFLFRFRFRFLFLFLFLFRFRF"

trace ( expandOne mapper ( expand ( rules 2 ) ( base 2 ) 1 ) ) ( angle 2 )
Blue
[ ( ( 0.0, 0.0 ), (1.0, 0.0 ), Blue, 1 ), ( ( 1.0, 0.0 ) , ( 1.0, 1.0 ),
Blue, 1 ), ( ( 1.0, 1.0 ), ( 2.0, 1.0 ), Blue, 1 ), ( ( 2.0, 1.0 ),
(2.0, 0.0 ), Blue, 1 ), ( ( 2.0, 0.0 ), ( 3.0, 0.0 ), Blue, 1 ) ]
```

Note you could alternatively have written:

```
 let (a,b,rs) = system 2 in trace (expandOne mapper
(expand rs b 2)) a Blue
```

To save typing in these long expressions you may find it convenient to define a function

```
lSystem :: Int -> Int -> String
```

which takes the numeric index of a predefined L-System and the number of expansions required and returns the final sequence of turtle commands. The above call to **trace** could then be replaced by the call **lSystem 2 1**, for example

- In the examples so far the string contains just the symbols 'M', 'N', '+', '-' and the final sequence of turtle commands generated from the final string defines a continuous *linear* path, i.e. with no branches. You can build more elaborate branching structures using *bracketed* L-Systems as in systems 7 and 8. A bracketed L-System becomes the new turtle state.

  Stacks are easily implemented in Haskell - they are isomorphic to lists. An item can be placed on top of the stack ("pushed") using the ':' operator. An item can be removed from the top of the stack either by pattern matching, e.g. f (top : rest) = ... or (less pleasing) using **head** and **tail**. So, what do you need to push to and pop from the stack? The answer is always 'just enough "state' so that the computation can be resumed from where it left off'. In this case we need to save the state of the turtle when a '[' is reached and process the string of commands that follow the '['. These commands may take the turtle off to an arbitrary place on the imaginary paper. No matter! The important thing is that when we finally hit the closing ']' later in the string of commands, the turtle state is guaranteed to be at the top of the stack, provided you've implemented the code

correctly. At this point you pop the stack and resume tracing *with the saved turtle state* and using the commands which follow the ']'. Note that it doesn't matter where the turtle was when you hit the ']'. It's like using several turtles to solve the problem—at a branch point you make a copy of the current one on the stack and send the original off to walk one of the branches. The copy on the stack is saved, ready to walk the other branch later on. The recursive nature of branching structures like bracketed L-Systems means that a branch make contain sub-branches but, again, no matter! For each '[' we push a turtle state and for each ']' we pop one off, so it doesn't matter how many sub-branches we encounter; provided the brackets match the stack is guaranteed to be restored so that our saved turtle state is the right one.

Important Aside Stacks are arguably the most important data structures in computer science precisely because they provide a way of traversing rich branching structures in a purely sequential manner. If you think about it this is exactly what programs do when they call functions. If we write a program in Haskell, Java, C... like:

```
f x y z = g1 x y + g2 x z
g1 a b = h ( a + b )
g2 a b = etc.
h c = etc.
```

f calls g1 and g2 (a branch in the call sequence!), and both g1 and g2 may call many other functions before a result can be returned. In a call to f the computer must calculate both g1 x y and g2 x z before it can return their sum. How does it remember the state of the computation prior to calling g1 so that it can resume later with g2 and then the sum? Answer: It uses a stack which remembers the values of the arguments x, y and z and where the execution of f got to prior to the call to g1. When the call to g1 completes the state (at the top of the stack) is restored and execution of f continues. Once again, it doesn't matter how many functions are called in the evaluation of g1 (and g2), so long as each function call has a matching return the stack will be restored with the correct state at the top.

Unassessed

Once you have completed the assessed part of the exercise you may like to extend your system to give you more practice and to produce prettier pictures! There is also a competition to produce the best image. You may like to do some additional research (e.g. starting at the url given earlier) for more ideas. You can see an example of what you can achieve if you look at the roses file available from the first year lab page; this was produced by making some small changes to the trace function. You might like to try the following:

- Vary the colour and thickness of the lines traced by the turtle. For example, you might build a branching L-system for modelling plant growth where the branches get thinner and change colour as you ascend the structure. This should be straightforward as **drawLines** has as an argument a **ColouredLine** which has an associated thickness and colour.

- Apply *probabilistic* rewriting. In probabilistic (or *(stochastic)* L-Systems, there may be more than one way to rewrite the same symbol, each with an associated probability. The total probability must sum to 1. This will involve extending the **Rules** to allow the associated probability to appear in the table. For example:

```
newrules 1 = [ ( 'M', 0.33, "M[+M]M[-M]M" ),
               ( 'M', 0.33, "M[+M]" ),
               ( 'M', 0.34, "M[-M]M" ),
               ( '+', 1.0,  "+" ),
               ( '-', 1.0,  "-" )  ]
```

- Add special features at the leaves of the structure, e.g. plant leaves or flowers (very ambitious, but quite possible).

The 2003 Haskell Programming Competition

- You are hereby invited to enter this year's competition, which is both optional and unassessed. There will be a 'star' prize of a bottle of Champagne (or a book token eqivalent) for the best picture generated by an L-System. You can modify the given L-System or your lab soloution code in any way you see fit. The competition is open to *groups* of students of arbitrary size (including 1) but in the case of a dead heat the smallest group wins! The competition will be judged by a panel of functional programming experts within the Department and the closing date for submissions is **Tuesday** 11^{th} **November**.

Submission

- As usual use the submit command, **submit 4**, in the appropriate directory. Make sure your file and the functions within it are correctly named. If you wish to enter the competition submit a *single* Haskell program, called **competition.hs**, (including any extended graphics functions) by typing **submit competition** and provide at the end of the program predefined functions for generating sample pictures. Please include a comment which explains how to 'run' the program to produce these sample pictures.

Assessment

```
angle, base, rule             0.5
lookupChar                    0.5
expandOne                     1.0
expand                        0.5
move                          1.0
trace                         1.5

Design, Style and Readability 5.0

Total                        10.0
```