L-Systems in Haskell

COMP40009 – Computing Practical 1

26th – 30th October 2020

Aims

- To provide experience of working with higher-order functions.
- To introduce the role of stacks and state in recursive computation.
- To introduce turtle graphics and L-Systems.

Introduction

An interesting challenge in the games and cinematic effects industries is that of generating and rendering visually realistic scenes containing organic structures like plants, bushes and trees.

One approach is to have a database of predefined objects that can be looked up and placed into a scene on request. However, if the same structure is used repeatedly then the rendered scene tends to looks very artificial – in nature not all trees are alike!

An alternative is to use a program to generate artificial plant-like structures from scratch using rules for generating trunks, branches, flowers etc. as the program executes. With a little additional effort, these rules can be applied probabilistically so that no two structures end up looking the same.

A popular way of defining plant-like structures is to use *Lindenmayer Systems*, or *L-Systems* for short. L-Systems are named after the biologist Aristid Lindenmayer who was interested in describing the apparently fractal nature of plant growth using simple rewriting rules. L-Systems can be used to define some of the well-known fractals you might have come across already.

In this exercise, we are going to work with very simple deterministic two-dimensional structures, but it should be easy to see how the process might be generalised, for example to build probabilistically-generated three-dimensional objects.

How L-Systems work

L-Systems work by repeatedly replacing the items in a list according to a fixed set of rewrite rules. Starting from an initial "seed" list, or *axiom*, the list grows in size as the rewrites are repeatedly applied. In this exercise the items will be characters, so the rewrite problem becomes one of expanding an axiom string into an output string respresenting the final structure. To render the structure the characters in the output string are mapped to a sequence of *commands* for a *turtle* - an imaginary device on wheels with a pen underneath that draws as it moves.

Turtle graphics

A graphics *turtle* can be thought of as a simple robot which moves around on a 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 an attribute of the L-System used to generate the command string.

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 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 list of such pairs (a table), thus:

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

An L-System consists of an axiom string, a list of rewrite rules and an associated rotation angle used to drive the turtle. In principle any characters can appear in an axiom string and rewrite rules. However, in order to drive the turtle the characters in the final output string must be mapped to the command characters 'F', `'R'` or `'L'` in some way (see later).

To help you get started a number of L-Systems have been defined for you in the skeleton file for this exercise, `LSystems.hs`. Each is stored as a triple of the form (angle, axiom, rules):

```
type LSystem = (Angle, Axiom, Rules)

cross, ..., arrowHead, ... :: LSystem

cross
  = (90,
     "M-M-M-M",
     [('M', "M-M+M+MM-M-M+M"),
      ('+', "+"),
      ('-', "-")
     ]
    )

...

arrowHead
  = (60,
     "N",
     [('M', "N+M+N"),
      ('N', "M-N-M"),
      ('+', "+"),
      ('-', "-")
     ]
    )

...
```

As a simple example, suppose we work with the `arrowHead` L-system. The axiom 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 arrowHead`. 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 `arrowHead` 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 the turtle will trace out the picture shown in Figure 1.

Bracketed L-Systems

In the above example the string contains just the characters `'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 `bush` and `tree`.

Bracketed systems present an interesting problem because the commands between a `'['` and its matching `']'`, corresponds to a *branch* in the structure. To render this, the turtle will have to draw the bracketed term as if it were a separate L-System. When we reach the matching `']'` we will have to restore the turtle state to where it was before it hit the `'['`. We thus need a mechanism for "remembering" the turtle state throughout the execution of the bracketed commands, so that we can process the commands after the `']'` starting from the same state. There are two ways to do it; both involve writing a helper function.

Method 1: By recursion

Think about the problem: when we hit a `'['` we have no idea where the matching `']'` is. Therefore, we cannot immediately determine the list of commands that follow the matching `']'`. One way to fix this is to get the helper function to return not only a list of coloured lines, corresponding to

the trace, but also any remaining commands that have yet to be processed. When you hit a `']'` command you return an empty trace (nothing more to do) *and* the list of commands that follow the `']'`. When you hit a `'['` you first produce the trace for the commands up to the matching `']'`. Then you process the remaining commands, which you now know as they are returned along with the trace. The final trace is obtained by appending the two together.

Note that nested bracketed terms fall out naturally by recursion.

Method 2: Using an explicit stack

The second solution remembers the turtle state *explicitly* at each `'['` command by carrying around a stack of turtle states – an additional argument to the helper function. When you process a `'['` command you *push* the current turtle state onto the stack. When you process a `']'` command you need to process the remaining commands (after the `']'`) but using the same turtle state that you had when you hit the matching `'['` command. But this turtle state is precisely what now sits on top of the stack! Therefore, all you need to do is *pop* this state from the stack and continue processing the remaining commands.

Notice again that nested bracketed command sequences present no problems: each time you come across a `'['` you push the current turtle state. If there was already a state on top of the stack that state will now sit one element lower. So, it will take two `']'` commands before that state is restored – exactly what you want.

This solution avoids the need to produce and dismantle pairs (Method 1). However, it requires an extra parameter to be carried with each call to the helper function.

Note that stacks are easily implemented in Haskell using 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 ("popped") by pattern matching, e.g. `f (top : stack) = ...`.

Important: Stacks are arguably the most important data structures in computer science, precisely because they provide a way of traversing rich branching structures in a sequential manner. You should implement both of the above methods as it will help you to understand the role of stacks in the implementation of recursive functions.

Getting started

As per the previous exercise, you will use the `git` version control system to get the repository with the skeleton files for this exercise and its (incomplete) test suite. You can get your repository with the following (remember to replace the *username* with your own username).

```
git clone https://gitlab.doc.ic.ac.uk/lab2021_autumn/haskelllsystems_username.git
```

What to do

The graphics module `IC.Graphics` has been included so that you can draw out the object produced by your program (using `drawLines`). A helper function, `drawLSystem1` (or `drawLSystem2`), is given that uses `drawLines` internally.[1]

You do not need to understand how the function `drawLines` is implemented. You can treat it as a *black box*, although the source is available for you to look at (and modify, even) if you are interested.

- Define three functions `angle`, `axiom` and `rules` for extracting the angle, axiom and rules components of an `LSystem`.

  ```
  angle :: LSystem -> Float
  axiom :: LSystem -> String
  rules :: LSystem -> Rules
  ```

- Define a function `lookupChar :: Char -> Rules -> String` that will look up a character in a list of expansion rules, each of which (a `Rule`) associates a character with a string. The function should return the associated string. A precondition is that a binding exists for the character in the expansion rules. For example:

  ```
  lookupChar 'M' (rules peanoGosper)
  "M+N++N-M--MM-N+"
  lookupChar '+' (rules triangle)
  "+"
  ```

[1] On some systems you may get an error that crashes `ghci` when using `drawLSystem1` or `drawLSystem2`. In this case, you should start `ghci` with the extra argument `-lglut`, i.e. `$ ghci -lglut LSystems.hs`. If you receive an error about glut missing, go to the terminal and type `$ cabal update` and then `$ cabal install glut` or `$ cabal install GLUT --reinstall`. If you receive any "Could not find module" errors, reinstall Haskell using: `$ sudo apt-get install haskell-platform`.

- Using `lookupChar` define a function `expandOne :: String -> Rules -> String` that will return the string formed by replacing each character in the given argument string with the string associated with that character in the given list of expansion rules. For example:

  ```
  expandOne (axiom triangle) (rules triangle)
  "-M+M-M-M+M"
  ```

- Define a function `expand :: String -> Int -> Rules -> String` that will invoke `expandOne` a specified number of times on a given initial string (axiom) using the given list of expansion rules. For example:

  ```
  expand (axiom arrowHead) 2 (rules arrowHead)
  "N+M+N-M-N-M-N+M+N"
  ```

- Define a function `move :: Command -> Angle -> TurtleState -> TurtleState` that will calculate the new position of a turtle given a move command (synonymous with `[Char]`). The move command can be either to turn left (`'L'`) or right (`'R'`) or to move forward (`'F'`) in the present direction.

  The angle is the one associated with the L-System that was used to generate the list of commands.

  The state of the turtle is given by its current (x, y) co-ordinate and its orientation, which is measured in degrees, anticlockwise from the positive x-axis.

  The type synonyms referred to above are defined in the template as follows:

  ```
  type Vertex
    = (Float, Float)

  type TurtleState
    = (Vertex, Float)

  type Command
    = Char

  type Commands
    = [Command]
  ```

For example:

```
*LSystems> move 'L' 90 ((100, 100), 90)
((100.0,100.0),180.0)

*LSystems> move 'F' 60 ((50, 50), 60)
((50.5,50.866024),60.0)

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

- Using `move`, define **two** trace functions: `trace1, trace2 :: Commands -> Angle -> Colour -> [ColouredLine]` that implement the two turtle tracing methods described above. They each take a string of *turtle* commands and convert them into a list of lines drawn with the specified colour that can be subsequently drawn on the screen by `drawLines`. The angle is the one associated with the L-System that was used to generate the list of commands – it is required when invoking `move`.

  The type `ColouredLine` defines the start and end points of a line in the Euclidean plane, along with its colour:

  ```
  type ColouredLine
    = (Vertex, Vertex, Colour)
  ```

  Note: If you are facing angle θ degrees anticlockwise from 3 o'clock, then a movement of length 1 would correspond to a displacement of $(cos\frac{\pi\theta}{180}, sin\frac{\pi\theta}{180})$ in (x, y) terms. `L` means add 90 (degrees) to θ and `R` means subtract 90 from θ. Initially the turtle is at $(0, 0)$ and has angle $\theta = 90$.

  In each case you need to keep track of the state of the turtle as it is moved and rotated; you will therefore need a helper function.

  For Method 1 you need to keep track of both the generated trace and the remaining command sequence. It's a little fiddly, as each call to the helper function now returns a *pair* of items, so you'll need to pattern match on the result each time in order to dismantle the pair. If you get stuck here you might like to get Method 2 working first.

  Recall that turtles have a single 'move forward' command (`'F'`) whilst the strings generated by the rewrite rules here use two characters `'M'`

and 'N' to mean the same thing. Also, the characters '+' and '-' are used to mean 'rotate left' and 'rotate right' respectively whereas the corresponding turtle commands are 'L' and 'R'. Before a string is passed to `trace`, therefore, the characters must be mapped into turtle commands. Fortunately, you have just built such a function which can perform such a mapping, i.e. `expandOne`! To use this to perform the mapping we need to define a new `Rules` table that maps characters to commands, e.g. 'M' and 'N' to 'F', '+' to 'L' etc., and then call `expandOne` accordingly. To save you time, we have provided this in the skeleton – it's called `commandMap` for obvious reasons. For example:

```
*LSystems> let (a, ax, rs) = arrowHead

*LSystems> let s = expand ax 3 rs

*LSystems> s
"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"

*LSystems> expandOne s commandMap
"FRFRFLFLFLFLFRFRFRFLFLFRFRFRFRFLFLFRFRFRFLFLFLFLFRFRF"

*LSystems> let (a, ax, rs) = triangle

*LSystems> trace1 (expandOne (expand ax 1 rs) commandMap) a blue
[((0.0,0.0),(1.0,0.0),(0.0,0.0,1.0)),((1.0,0.0),(0.99999994,1.0),
(0.0,0.0,1.0)),((0.99999994,1.0),(2.0,1.0),(0.0,0.0,1.0)),
((2.0,1.0),(2.0,0.0),(0.0,0.0,1.0)),((2.0,0.0),(3.0,0.0),
(0.0,0.0,1.0))]
```

Note the use of a `let` here to bind identifiers. GHCi remembers these up until next file (re-)load, or until they are redefined by a subsequent `let`. Instead of defining the components of `arrowhead` and `triangle` using a `let` we could, of course, have used the functions `angle`, `axiom` and `rules`.

To save typing in long expressions involving `expandOne` the template contains the following function:

```
expandLSystem :: LSystem -> Int -> String
expandLSystem (_, axiom, rs) n
  = expandOne (expand axiom n rs) commandMap
```

This takes the numeric index of a predefined L-System and the number of expansions required and returns the final sequence of turtle commands. You might find this useful.

Further extensions to basic system

Once you have completed the core part of the exercise you may like to extend your system to give you more practice and to produce prettier pictures! You may like to do some additional research for more ideas. Try the following:

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

- Apply *probabilistic* rewriting. In probabilistic (or *(stochastic)* L-Systems, there may be more than one way to rewrite the same character, 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:

```
probCross
  = [('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).

- Extend your system to 3D. If you choose to do this you'll need a way of rendering the resulting 3D structure. However, this should be easy as you've just finished studying vector algebra! Simply translate the structure so that it lines up with the z axis, and starting at the plane $z = k$ for some $k > 0$. Then place the viewer at the origin and use line/plane intersection (or just simple scaling) to find where the points in the structure intersect some viewing plane $z = k'$, where presumably $0 \leq k' \leq k$. The points in the structure are the end points of each line traced by the turtle.

Submission

As with all previous exercises, you will need to use the commands `git add`, `git commit` and `git push` to send your work to the GitLab server. Then, as always, log into the LabTS server, `https://teaching.doc.ic.ac.uk/labts`, click through to your `HaskellLSystems` exercise `https://gitlab.doc.ic.ac.uk/lab2021_autumn/haskelllsystems_username` and request an auto-test of your submission.

IMPORTANT: Make sure that you submit the correct commit to CATe – you can do this by checking that the key submitted to CATe matches the `Commit Hash` of your commit on LabTS/GitLab.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```

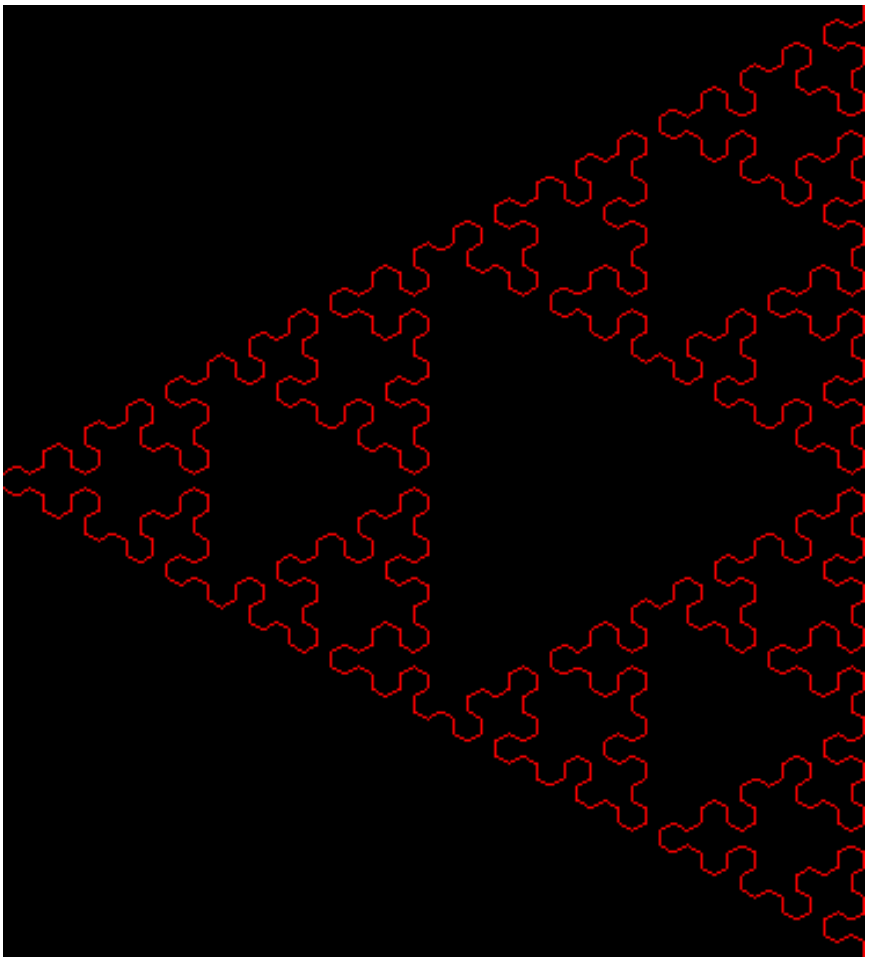

Figure 1: `arrowHead` L-System after six rewritings, rendered using the call `drawLSystem1 arrowHead 6 red` or the call `drawLSystem2 arrowHead 6 red`