L-Systems in Haskell

COMP40009 – Computing Practical 1

23rd – 27th October 2023

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 constructors of a simple datatype:

`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 `Command`, and a sequence of commands is represented as a `[Command]`. On the other hand, an L-System works with strings and 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. However, as we'll see later, it will be useful to generalise this slightly to be polymorphic – remember `String` and `[Char]` are the same: a set of rules will be represented as a list as follows:

```
type Rules a = [(Char, [a])]
```

An L-System consists of an axiom string, a list of rewrite rules producing lists of characters (i.e. `Rules Char`) and an associated rotation angle used to drive the turtle. In principle any characters can appear in an axiom string and rewrite rules. However, to drive the turtle the characters in the final output string must be mapped to the commands `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 `src/Examples.hs`. Each is stored as a datatype with an angle, axiom, and rules list:

```
data LSystem = LSystem Float [Char] (Rules Char)

cross, ..., arrowHead, ... :: LSystem
cross = LSystem 90 "M-M-M-M" [ ('M', "M-M+M+MM-M-M+M")
                             , ('+', "+")
                             , ('-', "-")
                             ]
...
arrowHead = LSystem 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 commands! 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.

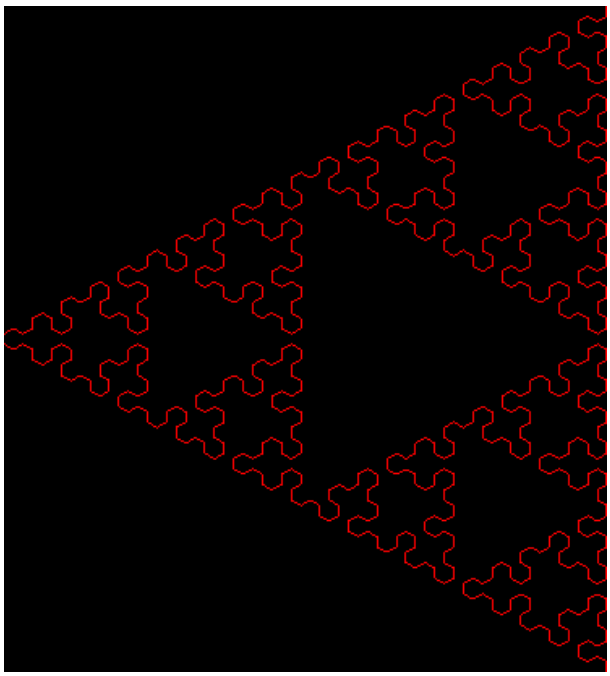

Figure 1: `arrowHead` L-System after six rewritings, rendered using the call `drawLSystem1 False arrowHead 6 Red` or the call `drawLSystem2 False arrowHead 6 Red`.

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, i.e., `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 must restore the turtle state to where it was before it hit the `'['`. We therefore 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.

This is quite tricky to do in a linearised representation like a list. Once upon a time, students were

asked to do this with, you can imagine, two extra turtle commands: `Save` and `Restore`. Instead we can simplify this problem a bit by making sure we have a representation of the turtle commands that easy encodes the *scope* of the backtracking:

```
data Command = F | L | R | B [Command]
```

The `B` constructor, standing for "bracketed", is new: it keeps a list of commands inside, which will be ran together such that when they finish the turtle state is restored to how it was on entry to the branch. However, while this makes the interpretation of the turtle straightforward, we somehow need to be able to translate a pair of `'['` and `']'` in the input into a `B` in the list of commands.

Parsing Commands

A *parser* is something that takes "unstructured" data, like text, and processes it into a more structured representation according to some *grammar* or *schema*. Since you have not met grammars yet, we won't touch on them here. Instead, it suffices to say that we need to write a function that recursively walks down an expanded L-System string and produces a list of commands as a result (in the parsing world, this is a simple *recursive-descent parser*). For this simple parser, you can think of it being modelled by the following type:

```
type Parser a = String -> (a, String)
```

This means that, given an input string, the parser can produce a value of type `a` and some *residual* input (stuff which it did not consume) alongside it. For our purposes, this will suffice: normally, you would need to consider what to do if the parser goes wrong and cannot parse a value, but here **we can assume the input is always *well-formed***. In our case, we are looking to construct a `Parser [Command]`.

In general, the parser should operate by considering each character in the input in turn, then figure out which command it should generate onto the result of parsing the rest of the input. The tricky bit is with the brackets: when we encounter a `'['` in the input, we don't know where the matching `']'` will be found – other than certainly somewhere deeper in the input. All of the commands returned from between the brackets will be in their own list, and these should go inside a `B` constructor. This means that we need to recursely apply the parser, stopping when we encounter the matching bracket, to produce `(cmds, residual)` then continue parsing `residual` afterwards to continue building the list of commands, adding `B cmds` onto the front of that result. As an example: let's parse `"N[-M][+M][NM]"` and see how it will arrive at a result:

1. First we start by consuming `'N'`, remembering we need to add `F` onto the front of the result of parsing the residual `['[', '-', 'M', ']', '[', '+', 'M', ']', '[', 'N', 'M', ']']`
2. Next, we encounter a `'['`, so we now recursively parse `['-', 'M', ']', '[', '+', 'M', ']', '[', 'N', 'M', ']']`
   (a) Read both the `'-'` and `'M'` and keep going with the remaining `[']', '[', '+', 'M', ']', '[', 'N', 'M', ']']`, remember we will need to add the `R`, and `F` later
   (b) Next we see a `']'`, so we are done reading a nested bracket pair: that means there are no more commands and the remaining input is left unconsumed. This returns `([], ['[', '+', 'M', ']', '[', 'N', 'M', ']'])`.

(c) Put the results on for the two other characters we parsed leading to a final result of `([R, F], ['[', '+', 'M', ']', '[', 'N', 'M', ']'])`

Having finished recursively parsing until the close bracket, take the result `[R, F]`, enclose it in a `B`, and continue parsing the residual input `['[', '+', 'M', ']', '[', 'N', 'M', ']']`, remembering to add `B [R, F]` to the result later.

3. Step 2 repeats twice more for the other two sets of brackets, remembering `B [L, F]` and `B [F, F]` respectively.

4. Finally, we see the empty list `[]`, so return an empty list of commands and empty residual input.

5. Add on all the found bracketed commands in reverse order, resulting in a final `([B [R, F], B [L, F], B [F, F]], [])`

That should help you understand the process of dealing with the recursive brackets. This description covers the cases for `'[':cs`, `']':cs` and `[]` specifically. Notice that the parser has two base cases, one for no input, and one for a closing bracket. If the head of the input is any other character, it will need to be converted into either a command, or nothing at all. In practice, L-Systems may require more rules than just `'M'` and `'N'`, and these rules may not map to commands at all. To help make this a bit easier, a `commandsMap` has been included to help handle all of these other cases:

```
commandMap :: Rules Command
commandMap = [ ('M', [F]), ('N', [F])
             , ('+', [L]), ('-', [R])
             , ('X', []), ('Y', []), ('A', [])
             ]
```

As examples, the `'X'`, `'Y'`, and `'A'` are all additional rules required by `bush` and `dragon`, but should not contribute to commands in the system. By leaving this `Rules Command` as an argument to the parser, this behaviour can be customised.

Executing the Turtle

After having performed parsing on the final string returned by L-System expansion, we are left with a `[Command]`. The next stage is to execute this list of commands to generate a list of coloured lines, which will then be rendered by the graphics engine. Bear in mind, however, we now need to ensure that when executing bracketed commands, the state is restored after the run. There are two methods of handling this problem:

Method 1: By recursion In this case, we are looking to process `B cmds : cmds'`, where `cmds` are the bracketed commands and `cmds'` is what should be ran next with the state put back to how it was. The beauty of recursion here is you know that if you execute `cmds` explicitly with a state `state`, when the function returns you will still have access to the exact same `state`: pass this forward to another recursive call and you're good!

Method 2: Using an explicit stack The other way of thinking about this is to track the remaining commands to execute and the state to execute it in explicitly in a stack as you execute. A stack in this instance is just implemented as a regular haskell list. In this case when you are processing `B cmds : cmds'` with state `state`, you will go ahead and execute a *single* recursive call executing `cmds` with `state`, but then explicitly push `cmds'` and `state` onto the stack to "remember" to execute them later.

In this method, there is more work to do in the base case, when the commands list is empty: when you've run out of commands to run, you need to now pop your stack and continue executing the popped commands with the popped state. Eventually, when you've run out of commands *and* the stack is empty, the function stops.

The question you might have is: "why?". Good question: for one, writing the execution function in this way will help you solidify the role of stacks in recursive functions: in the non-tail-recursive version, you are making use of an *implicit* stack called the *Call Stack* or *The Stack* (emphasis on The). This is pushed when you recurse, and popped when you finish executing: much like how you do extra work to pop the stack and carry on with "old" work in implementation with an explicit stack. It's useful to be able to relate these ideas as you will gain a better understanding of how your programs run. Secondly, this version may be more asymptotically more efficient, as you will likely implement the first method with `(++)`, which performs poorly in tree-like branching structures like this. The explicit stack version will process it linearly, as it is tail-recursive.

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/lab2324_autumn/haskelllsystems_username.git
```

Project Structure

This exercise has a more involved structure than the previous ones.

```
haskelllsystems_username
├── src
│   ├── IC
│   │   ├── Colour.hs ..... Defines a bunch of Colour values
│   │   └── Graphics.hs ..... Internal implementation of the rendering
│   ├── Examples.hs ..... Example L-Systems
│   ├── LSystems.hs ..... Where you will write the code for your solution
│   └── Renderer.hs ..... Pre-defined functions for drawing L-Systems (default module)
├── test
│   ├── IC
│   │   ├── Approx.hs ..... Testing utilities for approximate equality
│   │   └── Exact.hs ..... Testing utilities for exact equality
│   └── Tests.hs ..... A selection of unit tests written with tasty-hunit
└── lsystems.cabal
```

You will work within the `LSystems` module and the tests can be found inside the `Tests` module, as usual. If you don't want to, you don't need to look inside or modify any other files – except for `Examples`,

which contains definitions for several different L-Systems.

That said, you might want to know exactly what is happening with the other modules. The `Renderer` module contains two predefined functions for rendering an L-System graphically – more on that in the next section. The `IC.Colour` and `IC.Graphics` are both internal modules we have provided: the `IC.Colour` module provides the `Colour` type and several values which you can use as is – you will not need to manipulate the type itself except for possibly as part of your extension, so just use the values; the `IC.Graphics` module provides interacts with OpenGL[1] (via a library called GLUT) to provide the `drawLines` function for the `Renderer` to use – again, you shouldn't need to look at this code, and you can treat it entirely opaquely. Similarly, `IC.Approx` and `IC.Exact` provide helper functions for constructing the test-cases.

Running L-Systems

Unlike the Macro-processor exercise, which provided an application entry-point, this project expects you to interact with the system via `cabal repl`. To make things easier for you, the `Renderer` module has been set as the "cabal default module" – this is done by writing it first in the `exposed-modules` list. This means that when you run `cabal repl`, it will import `Renderer`, and *transitively* import `Examples` and `LSystems` too (as these are both imported by the module). This will give you immediate access to everything you'll need to do interesting things or play with your functions[2]. The `Renderer` module provides two functions:

```
drawLSystem1 :: Bool -> LSystem -> Int -> Colour -> IO ()
drawLSystem2 :: Bool -> LSystem -> Int -> Colour -> IO ()
```

These are so-named as they call your `trace1` and `trace2` functions, respectively (see "What to do"). The arguments to these functions are the same: the first argument says whether or not the rendering should happen "live" or not – if `True`, the lines made by our turtle will be drawn one by one, which is much slower but more fun to watch, and if `False`, the image will be rendered in a single-step, which is faster. Then, the second argument is the L-System you want to draw, the `Int` is how many expansion steps should be performed, and the last argument is what colour it should be drawn in. For example, Figure 1 was rendered by calling:

```
*Renderer> drawLSystem1 False arrowHead 6 Red
```

What to do

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

  ```
  angle :: LSystem -> Float
  axiom :: LSystem -> [Char]
  rules :: LSystem -> Rules
  ```

[1] On your own *Linux/WSL*, `freeglut3-dev` needs to be installed, which can be done with `sudo apt install freeglut3-dev`.

[2] If this is not the case – i.e., you're on Windows not using WSL – then you'll need to run `import Examples` and `import IC.Colour` first.

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

```
*Renderer> lookupChar (rules peanoGosper) 'M'
"M+N++N-M--MM-N+"

*Renderer> lookupChar (rules triangle) '+'
"+"
```

- Using `lookupChar` define a function `expandOne :: Rules Char -> [Char] -> [Char]` 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:

```
*Renderer> expandOne (rules triangle) (axiom triangle)
"-M+M-M-M+M"
```

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

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

- Define a function `move :: Command -> Float -> 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 `Float` is the angle 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)
```

  For example:

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

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

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

- Implement the `parse :: Rules Command -> [Char] -> [Command]` function as described above. Given the rules to convert characters to commands (other than `'['` and `']'`, which must be handled explictly in `parse`), `parse` will process a list of characters (the input) and produce the recursive commands as a result.

  For example:

  ```
  parse commandMap "M+YN[--AM+[++N]]-MXY" =
    [F, L, F, B [R, R, F, L, B [L, L, F]], R, F]
  ```

- Using `move`, define **two** trace functions: `trace1, trace2 :: [Command] -> Float -> Colour -> [ColouredLine]` that implement the two turtle tracing methods described above. They each take a list 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 `Float` is the angle 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 anti-clockwise 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.

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

  ```
  expandLSystem :: LSystem -> Int -> [Command]
  expandLSystem (LSystem _ axiom rs) n = parse commandMap (expand axiom n rs)
  ```

  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 for testing.

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/lab2324_autumn/haskelllsystems_username and request an auto-test of your submission.

IMPORTANT: Make sure that you submit the correct commit to Scientia – you can do this by checking that the key submitted to Scientia 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.