L-Systems in Haskell

Lucas Ferreira[1] and Sumukh Atreya[1]

[1]Department of Computer Science
[1]University of California, Santa Cruz

March 15, 2016

Abstract

This paper presents a technical report on the implementation of Lindenmayer Systems (or L-Systems) using Haskell, a functional programming language. It is presented as part of the requirements of the *CMPS 203 - Programming Languages* course at *University of California, Santa Cruz*. One of the main motivations of this project is to develop experience in designing and implementing good Haskell programs via real-world applications. L-Systems were selected because of their recursive structure and their importance in the field of Procedural Content Generation, especially for the generation of trees and forests.

1 Introduction

Video game complexity has increased substantially over the years. Consequently the content (characters, maps, cities, etc) present in these games has also increased. One of the main reasons for the increase in complexity of modern games is that the virtual worlds contained on them are becoming more realistic. A common problem, for example, when creating these worlds consists in designing forests. Without the help of a computer, the designer would have to handcraft every visible tree, in order to create a realistic scene.

The time and effort spent producing these assets have motivated game developers to design algorithms capable of generating content automatically. This approach is called Procedural Content Generation (PCG) and is often used in order to speed up the production of content. In this case, developers have the opportunity to interact with PCG methods, whose content created may be used as inspiration or starting point. These PCG methods must emphasize the novelty and diversity as opposed to quality or playability, since the developers have the possibility to reject or edit any undesired generated results. Several PCG methods have been developed to support the generation of entire forests and one of them is known as the Lindenmayer System (or L-System).

L-systems are generative grammars that define fractal patterns, which are used to generate content in video games and virtual worlds. It was originally defined to describe the behavior of plant cells and to model the growth processes of plant development [0]. Later, it began being used for PCG in games [1], since this task has become increasingly demanding. This paper presents a Haskell implementation of L-Systems, in partial fulfillment of the requirements for the *CMPS203 - Programming Languages* course of the *University of California, Santa Cruz*.

This document is structured as follows: Section 2 defines L-Systems and how they are used to produce fractal patterns. Section 3 describes the overall structure of the solution and the details of the Haskell implementation. Section 4 shows some examples of grammars and the outputs generated by the implemented system. Finally, Section presents a conclusion for this work.

2 L-Systems

L-Systems are formal grammars that are used to produce intricate and complex patterns that are self similar across many scales. It consists of four components:

1. A set of variables that can be replaced using a set of production rules.
2. Constants, which are symbols that are not replaced.
3. The axiom, which is a string composed of variables and constants. It is the initial state of the system.
4. Production rules, which define the way that variables can be replaced by other variables and constants. The production consists of a predecessor string and a successor string, which is obtained by applying the rules to the predecessor string.

For example, let us consider the system presented in Table 1, where the axiom is the string with three symbols "peg" and the set of production rules is composed of only one rule "e=eie".

Axiom:	peg
Rules:	e = eie

Table 1: Example of simple recursive L-System.

The single "e" symbol in the axiom is replaced with "eie" creating the string "peieg". Since the rule in this L-system is recursive, the first generation string has two new "e" symbols, each of which get replaced with "eie" in the second generation. This process goes on until the desired amount of generations is reached. Table 2 shows the result of applying the production rules on the system presented in Table 2 for 3 generations.

In order to use this system to recursively generate visual patterns (ex: plants, shapes etc.), one has to interpret them visually. A common way of doing this consists of using the generated strings to control turtle graphics.

Generation 1:	peieg
Generation 2:	peieieieg
Generation 3:	peieieieieieieieg

Table 2: The first three generations of strings produced by the L-System.

Thus, constants are used to move and turn a drawing agent, similar to the Logo programming language. For instance, let's consider the grammar described in Table 3.

Axiom:	FX
Rules:	X = X+YF+ Y = -FX-Y

Table 3: Example of grammar that can be rendered.

If the robot interprets F to mean "move forward drawing one unit" and - and + to mean "turn right 90 degrees" and "turn left 90 degrees", respectively, then the agent will draw what is illustrated in Figure 1 after 10 generations.

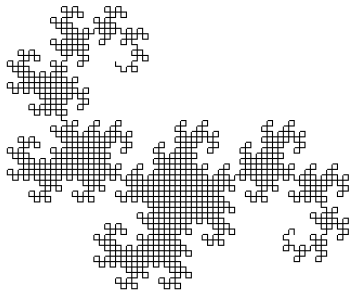

Figure 1: Rendering of the grammar defined in Table 3 after 10 generations.

3 Haskell Implementation

The Haskell implementation presented in this paper uses the turtle graphics structure defined in Section 2. Grammars are defined in text files, which are parsed into L-Systems. These systems are then rendered using the *Gloss* [1] package, which is used to draw simple vector graphics on a window (this implementation can be downloaded on GitHub). [2] and to compile and run the source code, first download and install *Gloss*. After that, use the following command lines:

[1] `https://hackage.haskell.org/package/gloss`
[2] `https://github.com/lucasnfe/Haskell-LSystems`

```
$ ghc lsystems.hs -o lsystems
$ ./lsystems Grammars/grammar.txt
```

The program expects only one argument in the command line, which is the path of the file that contains the grammar. A grammar file must contain at least 4 lines. The first line is an integer defining the number of times the axiom must be expanded (that is, the number of generations). The second line is a real number that defines the angle that the drawing agent will turn every time it encounters a constant "F". The third line is the axiom of the grammar and the fourth line is a production rule. A grammar must have at least one production rule. The remaining rules must be defined from the fifth line onward. Figure 2 shows how to define the grammar of Table 3 using the system implemented in Table 3.

```
1  10
2  90
3  FX
4  X=X+YF+
5  Y=-FX-Y
```

Figure 2: File describing the grammar of Table 3.

The grammar files are parsed into custom Haskell data types, which are then manipulated by functions that apply the production rules into the axiom. The figure below shows how an L-System is defined as a Haskell data type.

```
type Rule = (Char, String)
type LSystem = (Int, Float, String, [Rule])
```

To transform a file into the data type above, two functions were defined: *parseGrammarFile* and *parseGrammarData*, respectively. The first function has type String → [String] and is responsible for reading the file and transforming it into a list of strings. Each element of this list represents a line of the file. The second function has type [String] → LSystem and takes the list as input and creates an L-System, using the data type defined above. Both functions are defined in Figure below.

The last two steps of this implementation consists of expanding the axiom using the production rules and rendering the resultant string. Two functions were defined to accomplish this: *expandAxiom* and *renderExpandedAxiom*. The first one has type LSystem → String and expands the axiom by evaluating each of its constants recursively. At each step of the recursion, the current evaluated constant k is substituted by the first rule that has k as a predecessor. The functions that expand the axiom are the following:

```
parseGrammarFile fileName = do
  text <- readFile fileName
  return (lines text)

parseGrammarData grammarData = do
    steps <- parseStept grammarData
    angle <- parseAngle grammarData
    axiom <- parseAxiom grammarData
    rules <- parseRules grammarData
    return (steps, angle, axiom, rules)


execRule x (pre, sec) = if x == pre then sec else []

execAllRules rules x = if x `elem` consts then [x] else execRule x =<< rules

replace (_, _, s, rs) = execAllRules rs =<< s

expandAxiom (0, _ , axiom, _) = axiom
expandAxiom ls@(i, angle, axiom, rules) = expandAxiom (i-1, angle, replace ls, rules)
```

4 Results

5 Conclusion