**Sistemas-L e *Turtle Graphics***

Relatório do Trabalho de Avaliação de Programação II

**Curso de Engenharia Informática**

**2018/2019**

****

Diogo Solipa, nº43071

Leonardo Catarro, nº43025

**Introdução e Objetivo do trabalho**

Um sistema-L ou um Sistema de Lindenmayer é um sistema de [reescrita paralela](https://pt.wikipedia.org/wiki/Sistema_de_redu%C3%A7%C3%A3o) e um tipo de [gramática formal](https://pt.wikipedia.org/wiki/Gram%C3%A1tica_formal). Um L-Sistema consiste em um alfabeto de símbolos que podem ser usados para fazer uma [cadeia de caracteres](https://pt.wikipedia.org/wiki/Cadeia_de_caracteres), uma coleção de regras de produção em que se expande cada símbolo em uma maior cadeia de símbolos, uma sequência inicial "[axioma](https://pt.wikipedia.org/wiki/Axioma)", em alguns casos apenas uma letras, em outros mais que uma. A partir da qual começa a construção, é um mecanismo para traduzir as sequências geradas em estruturas geométricas.

Este trabalho estava dividido por níveis em que em cada nível deveria ser implementado os seguintes Sistemas e Classes/Instruções:

-Nível 0:

Sistemas: Koch Curve, Koch Snowflake, Sierpinski Triangle, Sierpinski Arrowhead, Dragon Curve;

Classes: forward, turn, penUp, penDown

-Nível 1:

Sistemas: Cantor Set

Classes: Nível 0 + leap

-Nível 2:

Sistemas: Fractal Plant

Classes: Nível 1 + save, restore

Assim sendo, o objetivo deste trabalho consistia em se conseguir, juntamente com o Package Galapagos (fornecedor das classes e instruções que possibilitam a execução de desenhos 2D) implementar o necessário para visualização do acima referido!

**Implementações**

**Interfaces:**

**-LSystem:**

package galapaxos;  
  
public interface LSystem {  
 void setStart(String start);  
 void addRule(Character symbol, String word);  
 String iter(String word);  
 String loopIter(int n);  
}

Esta é uma interface que contém os métodos relativos à construção dos L-Systems. Foi lhe adicionado um método (à interface inicialmente dada), loopIter, que não é mais que um método que escolhe o número de vezes que o método iter vai ser corrido.

**-Interpreter:**

package galapaxos;  
  
import java.util.List;  
  
public interface Interpreter {  
 void run(List<TurtleStatement> program);  
 void runForward(Forward statement);  
 void runTurn(Turn statement);  
 void runPenUp(PenUp statement);  
 void runPenDown(PenDown statement);

}

A interface Interpreter contém os métodos que vão ser implementados na InterpreterClass, que basicamente consiste num conjunto de métodos que irá passar de um TurtleStatement para um movimento tartaruga (em termos gráficos posteriormente).

**Classes:**

**-Compiler:**

package galapaxos;  
  
import java.util.HashMap;  
import java.util.Vector;  
  
public class Compiler {  
 HashMap<Character, TurtleStatement> charToAction;  
  
 public Compiler(){  
 charToAction = new HashMap<Character, TurtleStatement>();  
  
 }  
 public void addRule(Character letter, TurtleStatement statement) {  
 charToAction.put(letter, statement);  
  
 }  
 protected TurtleStatement compile(Character c) {  
 return charToAction.get(c);  
  
 }  
 protected Vector<TurtleStatement> compile(String word) {  
 Vector<TurtleStatement> result = new Vector<>();  
 for (int i = 0; i < word.length(); i++) {  
 result.add(compile(word.charAt(i)));  
 }  
 return result;  
 }  
}

Nesta class implementamos um mapa, que inicializamos no construtor. Este mapa recebe um caracter, e agrupa-o com um TurtleStatement. Quando chamamos o addRule, o que ele faz é ir ao mapa e associar um char a um statement. Temos ainda outro método o compile do tipo TurtleStatement, que recebe um caracter, o que acontece por trás é simplesmente outra ação do mapa, recebe um char e vê o que está associado a este. Por fim temos um outro método que compila o caracter num vetor de TurtleStatements, ou seja, pega numa String do tipo “F+F” e retorna @”memoria”FORWARD @”memoria”TURN90 @”memoria”FORWARD.

**-InterpreterClass:**

package galapaxos;  
  
import java.util.List;  
  
public class InterpreterClass implements Interpreter {  
  
 Turtle turtle = new Turtle(Turtle.*CREATE\_DEFAULT\_WINDOW*); // para so ter 1 janela aberta  
 TurtleDrawingWindow turtleDraw = new TurtleDrawingWindow();  
  
 InterpreterClass(){  
 //turtleDraw.setOrigin(230,0);  
 turtle.speed(6000); // de modo a ser instantaneo a construção do gráfico  
 turtle.penSize(1);  
  
 turtleDraw.setGrid(false);  
 turtleDraw.add(turtle);  
 turtleDraw.setVisible(true);  
 turtle.hide();  
 }  
  
 public void run(List<TurtleStatement> program){  
 for(int i = 0; i<program.size();i++) {  
  
 TurtleStatement t = program.get(i);  
 t.run(this); //simplesmente diz que vai correr nesta class (this)  
  
 }  
 }  
  
 public void runForward(Forward statement){  
 turtle.move(statement.getDistance());  
  
 }  
  
 public void runTurn(Turn statement){  
 turtle.turn(statement.getRotation());  
  
 }  
  
 public void runPenUp(PenUp statement){  
 turtle.penUp();  
  
 }  
  
 public void runPenDown(PenDown statement){  
 turtle.penDown();  
 }  
}

Implementa a interface Interpreter, logo contém todo os métodos da mesma. Esta class tem um método run, que o que faz é percorre uma lista de TurtleStatements (forwards, turns, leaps etc), e atribuí por cada iteração um statement a um TurtleStament criado, que posteriormente vai ser executado nesta class com a chamada do método .run(this). (O argumento this serve para informar o programa que o código a correr encontra-se nesta classe, por exemplo: se t fosse um Forward, t.run(this) indicaria que iriamos usar o método runForward).

**-Forward:**

package galapaxos;  
  
public class Forward extends TurtleStatement {  
 double distance;  
 public Forward(double distance) { this.distance = distance; }  
 public double getDistance() { return distance; }  
 public void run(Interpreter interpreter) {  
 interpreter.runForward(this);  
 }  
}

Uma class muito básica que estende a classe TurtleStatement, pois utiliza métodos da mesma. Serve para definirmos uma distância para depois usarmos nos movimentos tartaruga e para referenciar o método run do Interpreter.

**-Turn:**

package galapaxos;  
  
public class Turn extends TurtleStatement{  
  
 double rotation;  
 public Turn(double rotation) { this.rotation = rotation; }  
 public double getRotation() { return rotation; }  
 public void run(Interpreter interpreter) {  
 interpreter.runTurn(this);  
 }  
}

Na mesma base que a class Forward, esta simplesmente muda o objetivo, em vez de uma distância, temos uma rotação (em graus).

**-PenUp:**

package galapaxos;  
  
public class PenUp extends TurtleStatement {  
  
 public PenUp() {  
  
 }  
  
 public void run(Interpreter interpreter) {  
 interpreter.runPenUp(this);  
 }  
}

Define a ação de PenUp que não é mais que o levantar da “caneta”, ou seja, quando esta classe é chamada e utilizada a tartaruga não ira deixar rasto.

**-PenDown:**

package galapaxos;  
  
public class PenDown extends TurtleStatement {  
  
 public PenDown() {}  
  
 public void run(Interpreter interpreter) {  
 interpreter.runPenDown(this);  
 }  
}

À semelhance da PenUp, a PenDown é exatamente igual, apenas muda o resultado, pois o objetivo desta é voltar a deixar rasto.

**-Koch Curve:**

package galapaxos;  
  
import java.util.HashMap;  
  
 public class KochCurve implements LSystem {  
  
 public String symbolIter; // string que é transformada dps de aplicar a regra ao char introduzido  
  
 HashMap<Character, String> charToString; // dicionario para passar de 1 simbolo para uma string  
  
 public KochCurve() {  
  
 this.charToString = new HashMap<Character, String>(); // inicialização do mapa  
  
 }  
  
 public void setStart(String start) {  
 this.symbolIter = start; //inicializa a string para a 1 iteração  
 }  
  
 public void addRule(Character symbol, String word) {  
 charToString.put(symbol, word);  
 }  
  
 public boolean hasChar(char symbol) { // função que verifica se o simbolo introduzido refere-se a uma string ou não  
 if (charToString.get(symbol) != null)  
  
 return true;  
  
 else  
 return false;  
 }  
  
 public String iter(String word) { // função que itera 1 so vez, depois faz as iterações por cada vez que é chamada na loopIter  
  
 String finalString = ""; // Inicilizar a String para ser concatenada ; NOTA : NÃO INCIALIZAR A NULL !  
 char c;  
 for (int i = 0; i < word.length(); i++) {  
 c = word.charAt(i);  
 if (hasChar(c))  
 finalString += charToString.get(c);  
 else  
 finalString += c;  
 }  
  
 return finalString;  
 }  
  
 public String loopIter(int n) {  
  
 String aux = symbolIter; //inicializar uma auxiliar para usar na função iter  
 for (int i = 0; i < n; i++) {  
  
 charToString.get(aux); // processo inverso a obter a palavra, ou seja vai buscar o simbolo introduzido que gerou a palavra  
 aux = iter(aux);  
  
 }  
 return aux;  
  
  
 }  
 }

Esta classe é a classe dos Sistemas-L, pois é a classe responsável por executar as iterações após ser atribuído o start, por vezes uma letra, por outras mais que uma!

É também responsável pela “conversão” do start na sequência que é gerada, ou seja, se F for o start, mas F for F+G, por exemplo. É esta a classe responsável por executar este tipo de condições

Esta classe tem também o método iter que é responsável por iterar a primeira vez e é apenas chamado quando chamado o método loopIter.

O método loopIter, responsável por processo inverso a obter a palavra, ou seja, vai buscar o símbolo introduzido que gerou a palavra

**-Leap:**

package galapaxos;  
  
public class Leap extends TurtleStatement {  
  
 double distance;  
 public Leap(double distance) { this.distance = distance; }  
 public double getDistance() { return distance; }  
 public void run(Interpreter interpreter) {  
 interpreter.runLeap(this);  
 }  
}

Leap, embora não funcionando corretamente, esta seria a classe responsável que permitiria a execução do Sistema-L, Cantor Set, de nível 1.

**MAIN:**

package galapaxos;  
  
  
  
import java.util.InputMismatchException;  
import java.util.Scanner;  
import java.util.Vector;  
  
public class Main {  
  
 public static void main(String [] args) {  
 KochCurve charMult = new KochCurve();  
 Compiler comp = new Compiler();  
 InterpreterClass it = new InterpreterClass();  
  
  
 Scanner sc = new Scanner(System.*in*);  
 Vector<TurtleStatement> vec;  
 String word;  
  
 try {  
 System.*out*.println("Introduza o L-System que pretende Iterar: ");  
 String system = sc.nextLine();  
 System.*out*.println("Introduza o numero de vezes que quer iterar o L-System pretendido: ");  
 int n = sc.nextInt();  
  
  
 switch (system.toLowerCase()) { // o lowercase é para a palavra ser case insensitive  
  
  
 case "koch curve":  
  
 charMult.setStart("F");  
 charMult.addRule('F', "F+F-F-F+F");  
  
 word = charMult.loopIter(n);  
  
 comp.addRule('F', new Forward(5));  
 comp.addRule('+', new Turn(90));  
 comp.addRule('-', new Turn(-90));  
  
 vec = comp.compile(word);  
 it.run(vec);  
  
 break;  
  
 case "koch snowflake":  
  
 charMult.setStart("F");  
 charMult.addRule('F', "F+F--F+F");  
  
 word = charMult.loopIter(n);  
  
 comp.addRule('F', new Forward(5));  
 comp.addRule('+', new Turn(90));  
 comp.addRule('-', new Turn(-90));  
  
 vec = comp.compile(word);  
 it.run(vec);  
  
 break;  
  
 case "sierpinski triangle":  
  
 charMult.setStart("F");  
 charMult.addRule('G', "F");  
 charMult.addRule('G', "GG");  
 charMult.addRule('F', "F-G+F+G-F");  
  
  
 word = charMult.loopIter(n);  
  
 comp.addRule('F', new Forward(5));  
 comp.addRule('G', new Forward(5));  
 comp.addRule('+', new Turn(120));  
 comp.addRule('-', new Turn(-120));  
  
 vec = comp.compile(word);  
 it.run(vec);  
  
 break;  
  
 case "sierpinski arrowhead":  
  
 charMult.setStart("F");  
 charMult.addRule('G', "F");  
 charMult.addRule('F', "G-F-G");  
 charMult.addRule('G', "F+G+F");  
  
 word = charMult.loopIter(n);  
  
 comp.addRule('F', new Forward(5));  
 comp.addRule('G', new Forward(5));  
 comp.addRule('+', new Turn(60));  
 comp.addRule('-', new Turn(-60));  
  
 vec = comp.compile(word);  
 it.run(vec);  
  
 break;  
  
 case "dragon curve":  
  
 charMult.setStart("F");  
 charMult.addRule('F',"G");  
 charMult.addRule('F', "F+G");  
 charMult.addRule('G', "F-G");  
  
 word = charMult.loopIter(n);  
  
 comp.addRule('F', new Forward(5));  
 comp.addRule('G', new Forward(5));  
 comp.addRule('+', new Turn(90));  
 comp.addRule('-', new Turn(-90));  
  
 vec = comp.compile(word);  
 it.run(vec);  
  
 break;  
  
 default:  
 System.*out*.println("O nome do L-System está incorreto!"+"\n"+"Cuidado com espaços incorretos!");  
  
 }  
 }catch (Exception e){  
  
 System.*out*.println("Algo de errado não está certo!");  
 }  
  
  
 }  
}

A Main serve para na prática ligarmos todas as classes para um fim e para o output desejado. Neste caso começamos por declarar e inicializar os objetos das classes a utilizar, a Koch Curve, onde está toda a lógica de passagem de um caracter para uma string correspondente e iterar a mesma. O compiler que passa de F’s para statements do tipo Forward e por fim o InterpreterClass que passa de Forwards, para turtle move.

Na Main implementamos um Switch dentro de um Try Catch. O switch era o mais adequado pois permite-nos ser muitos específicos no input desejado e num possível feedback ao utilizado, no caso de existir uma interface gráfica, este switch iria desaparecer. O Try Catch serve para evitar erros graves de input, por exemplo o utilizador fazer um MissMatch no tipo colocado.

Dentro de cada “case” temos um addRule em que passamos de um caracter para uma string, temos um setStart que inicializa uma String para iniciar as iterações (consideramos o F o mais standard para tal). Associamos a uma String word, declarada fora do Try Catch, que se responsabilizada pela iteração. Compilamos cada ação, um + 🡪 Turn; F 🡪 Forward; etc. Por fim vamos usar um vetor para ser compilado na class Interpreter e passar de Forwards e Turns para movimentos tartaruga e desenhos gráficos.

**Apreciação Crítica**

Embora não tenhamos conseguido implementar tudo o que era necessário para execução dos Sistemas-L de níveis 1 e 2, pois ainda se conseguiu implementar o Leap (embora a não funcionar corretamente). O objetivo principal foi conseguido, pois os Sistemas-L de nível 0, juntamente com as classes e interfaces necessárias para a visualização dos mesmo, foi implementado com sucesso (talvez não da forma mais simples, pois como sabemos é inúmeras formas de implementar a coisa!).

Inicialmente, tivemos bastante dificuldade a entender o que seria preciso fazer para realizar este trabalho, mas após diversas leituras e pesquisas rapidamente percebemos o que se pretendia com este trabalho.

Em suma, projeto um quanto trabalhoso, mas que desenvolveu, certamente, a nossa capacidade de raciocínio e de programação na linguagem Java.