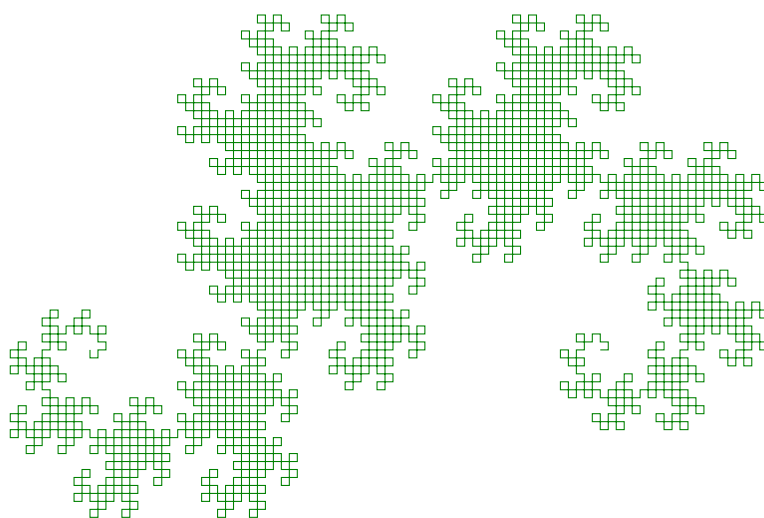


# Sistemas-L e *Turtle Graphics*

## Trabalho de Avaliação de Programação II

Salvador Abreu, Francisco Coelho

2018/19



## Conteúdo

<b>1</b>	<b>Sistemas-L</b>	<b>2</b>
<b>2</b>	<b>Turtle Graphics e Programas Tartaruga</b>	<b>4</b>
<b>3</b>	<b>Implementação</b>	<b>4</b>
3.1	Gerar Palavras Usando Sistemas-L . . . . .	5
3.2	Compilar Palavras Geradas para Programas Tartaruga . . . . .	6
3.3	Interpretar Programas Tartaruga como Gráficos . . . . .	10
<b>4</b>	<b>Avaliação</b>	<b>11</b>

## Resumo

Uma *Gramática* (ver a página em inglês na [Wikipedia](#)) é um sistema formal usado, por exemplo, para definir linguagens de programação. Resumidamente, uma gramática define um conjunto de “letras”, uma “palavra inicial” e um conjunto de “regras” de re-escrita.

Um *Sistema-L* (ou Sistema de Lindenmayer, ver a [página da wikipedia](#) e [este artigo](#).) é um tipo simplificado de gramática com uma interpretação gráfica especialmente adequada a *Turtle Graphics*.

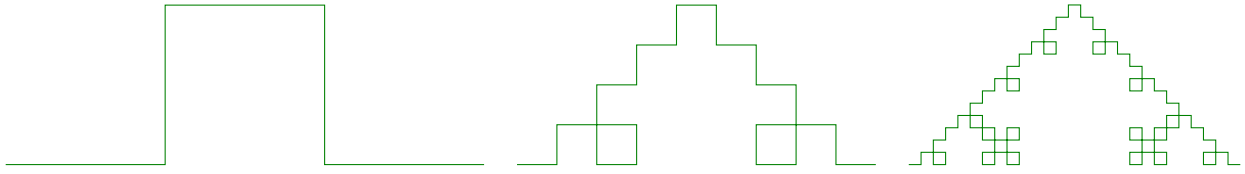


Figura 1: Uma representação gráfica de um Sistema-L, a **Curva de Koch**, definida na equação 1. Da esquerda para a direita estão a primeira, segunda e terceira iterações, a escalas diferentes.

Neste trabalho deve implementar um conjunto de classes Java que permita definir Sistemas-L, gerar palavras por esses sistemas e visualizar graficamente essas palavras.

A estratégia de resolução para este projeto está relacionado com assuntos que vai encontrar mais tarde no seu curso de *Engenharia Informática*. Em particular as *Gramáticas Formais* e as *Representações Intermédias* são primeiro tratadas em *Linguagens Formais e Autômatos* e, de novo, em *Linguagens de Programação*, onde são desenvolvidas para a *Compilação* e para a *Interpretação* de programas.

Os Sistemas-L ainda proporcionam um excelente exemplo de *Complexidade Computacional*. Embora, em termos de programação, sejam simples de compreender e de implementar, a sua execução muito rapidamente excede qualquer memória e/ou o tempo disponíveis.

Nas secções seguintes encontra uma breve explicação dos Sistemas-L e dos *Turtle Graphics*. Os pormenores que dizem respeito à sua implementação deste projeto estão descritas na secção 3. Por fim, na secção 4, está enquadrada a avaliação do seu trabalho.

## 1 Sistemas-L

Um **Sistema-L** é um sistema formal  $L = (\mathcal{A}, \mathcal{S}, \mathcal{R})$  em que  $\mathcal{A}$  é um conjunto de **letras** (ou **símbolos**),  $\mathcal{S} \in \mathcal{A}^*$  é a **palavra inicial** e  $\mathcal{R}$  é o conjunto de **regras** (ou **produções**).

$$CdK = \begin{cases} \mathcal{A} = \{F, +, -\} \\ \mathcal{S} = F \\ \mathcal{R} = \{F \rightarrow F + F - F - F + F\} \end{cases} \quad (1)$$

Uma regra tem a forma  $X \rightarrow w$  em que  $X \in \mathcal{A}$  é uma letra e  $w \in \mathcal{A}^*$  é uma **palavra** escrita com as letras de  $\mathcal{A}$ . Por exemplo, o sistema da equação 1 tem uma única regra,  $F \rightarrow F + F - F - F + F$ , que pode ser interpretada da seguinte forma: “Substituir cada  $F$  por  $F + F - F - F + F$ ”.

Um Sistema-L (e, em geral, uma Gramática) gera palavras começando pela palavra inicial e substituindo letras, de acordo com as regras, e assim obtendo novas palavras a partir das anteriores. A repetição deste processo designa-se por **iteração** e produz uma **linguagem** (um conjunto de palavras).

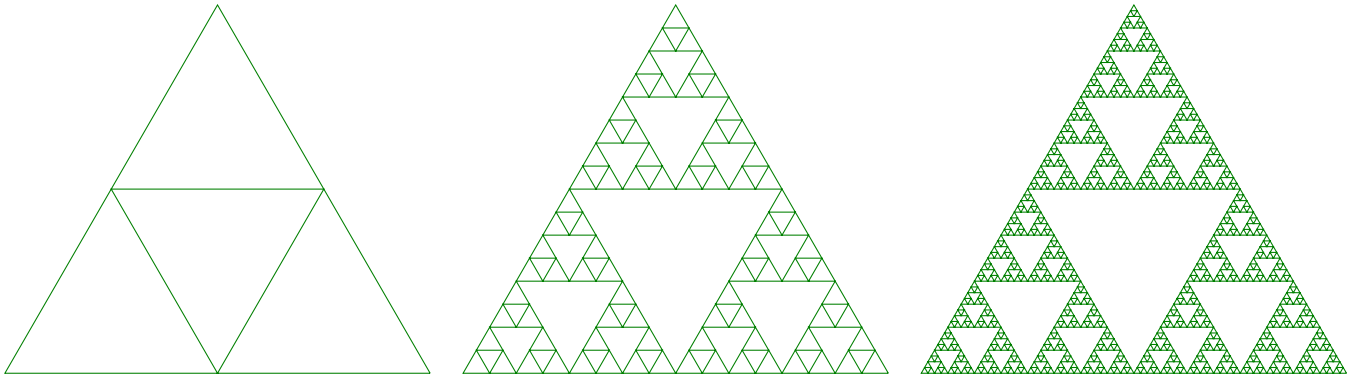


Figura 2: O **Triângulo de Sierpinski** é um conhecido Sistema-L. Está definido por duas regras, na equação 2. Nesta figura estão representadas a primeira, quarta e sexta iterações, a escalas diferentes.

Por exemplo, o sistema da equação 1:

1. Começa por gerar a palavra inicial:  $F$ .
2. Na primeira iteração, **substituindo todos os**  $F$  (só há um...) de acordo com a (única) regra, obtém-se a palavra

$$F + F - F - F + F.$$

3. Continuando, a segunda iteração deste Sistema-L resulta de **substituir todos os**  $F$  da palavra anterior. O resultado é:

$$F + F - F - F + F + F + F - F - F + F - F + F - F - F + F - F + F - F - F + F + F + F - F - F + F.$$

4. Sucessivamente, sempre que possível, substituindo cada letra de acordo com uma regra, geram-se outras palavras.

Note que num Sistema-L podem haver várias regras:

$$TdS = \begin{cases} \mathcal{A} = \{F, G, +, -\} \\ \mathcal{S} = F - G - G \\ \mathcal{R} = \begin{cases} F \rightarrow F - G + F + G - F, \\ G \rightarrow GG \end{cases} \end{cases} \quad (2)$$

**Resumindo**, um Sistema-L define um conjunto de **regras** de re-escrita e uma **palavra inicial**. Dessa palavra, re-escrevendo as letras, obtém-se outra palavra, e depois outra e assim sucessivamente.

O processo de gerar uma palavra a partir da anterior chama-se **iteração**. No caso da Curva de Koch (equação 1) a iteração 0 é a palavra inicial,  $F$ ; a iteração 1 é  $F + F - F - F + F$ ; *etc.*

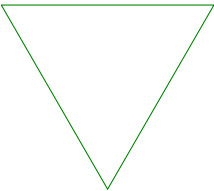
Palavra	Programa	Gráfico
$F - G - G$	<pre>forward(1) turn(-120) forward(1) turn(-120) forward(1)</pre>	

Figura 3: O programa que resulta da palavra inicial do Triângulo de Sierpinski,  $F - G - G$ , com a respetiva representação gráfica, no lado direito. Note que os desenhos da figura 2 estão “de cabeça para baixo”, isto é, rodados  $180^\circ$ .

## 2 Turtle Graphics e Programas Tartaruga

Os *Turtle Graphics* (ver a [página](#) da wikipedia) são uma variante simples dos caminhos usados nos sistemas gráficos 2D: Uma “tartaruga” é controlada por instruções `forward(distance)`, `turn(angle)` e `penUp()`, `penDown()`. O “rasto” define um desenho.

A relação entre os Sistemas-L e os *Turtle Graphics* é feita pela **tradução** gráfica das letras do sistema. Fazendo corresponder uma operação da tartaruga a cada letra do Sistema-L, então **uma palavra define um percurso** no plano  $XY$ .

Por exemplo, para o Triângulo de Sierpinski (figura 2) é usada a tradução indicada na tabela 1. Desta forma a palavra inicial do Triângulo de Sierpinski,  $F - G - G$ , é interpretada como um PT a que, por sua vez, corresponde uma figura triangular (ver figura 3).

Para a Curva de Koch (da figura 1) foi usada a tradução indicada na tabela 2.

## 3 Implementação

A implementação deve **separar** as seguintes tarefas:

1. **Gerar** palavras usando Sistemas-L.
2. **Compilar** palavras geradas para Programas Tartaruga.
3. **Interpretar** Programas Tartaruga como gráficos.

Tabela 1: Tradução entre as letras do Triângulo de Sierpinski e “Instruções Tartaruga”. As letras  $F$  e  $G$  definem a mesma instrução, `forward(1)` enquanto que  $+$  e  $-$  definem rotações de  $\pm 120^\circ$ .

Símbolo	Instrução	Símbolo	Instrução
$F$	<code>forward(1)</code>	$G$	<code>forward(1)</code>
$+$	<code>turn(120)</code>	$-$	<code>turn(-120)</code>

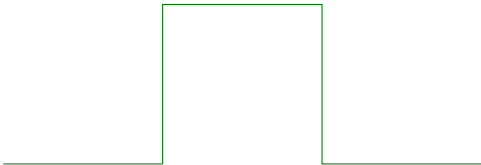
Palavra	Programa	Gráfico
$F + F - F - F + F$	<pre> forward(1) turn(90) forward(1) turn(-90) forward(1) turn(-90) forward(1) turn(90) forward(1) </pre>	

Figura 4: Ao centro o Programa Tartaruga que resulta da primeira iteração da Curva de Koch,  $F + F - F - F + F$ , à esquerda, com a respetiva representação gráfica, à direita.

Este processo é semelhante ao que acontece quando está a trabalhar com a linguagem Java (ou C ou qualquer outra linguagem *compilada*): **primeiro**, gera o código fonte do programa num editor de texto; **segundo**, o código do programa é *compilado* com o comando `javac`; em **terceiro**, o ficheiro `.class` que obteve no passo anterior é *interpretado* pelo comando `java`.

Neste caso, **primeiro** o código é gerado por um Sistema-L; **segundo**, compilado para uma **representação intermédia**; e **terceiro**, interpretado para produzir, por exemplo, uma imagem no ecrã.

### 3.1 Gerar Palavras Usando Sistemas-L

Para gerar palavras usando um Sistema-L é necessário considerar a **palavra inicial**, as **regras** do Sistema-L e também o **número de iterações** pretendido. Para tal use o interface `LSystem`, indicado no algoritmo 1, para representar Sistemas-L e para gerar palavras desses sistemas.

**Algoritmo 1** Interface `LSystem` para definir Sistemas-L.

```

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

```

O método `setStart(String start)` é usado para definir a palavra inicial do sis-

Tabela 2: Tradução das letras usadas na Curva de Koch. O ângulo associado às letras  $+$  e  $-$  passa a ser de  $\pm 90^\circ$ , em vez dos  $\pm 120^\circ$  usados no Triângulo de Sierpinski.

Símbolo	Instrução	Símbolo	Instrução
$F$	<code>forward(1)</code>		
$+$	<code>turn(90)</code>	$-$	<code>turn(-90)</code>

tema, `addRule(c, w)` para adicionar a regra  $c \rightarrow w$  e, *depois de se ter definido a palavra inicial e adicionado todas as regras*, `iter(n)` serve para gerar a palavra que resulta de iterar o sistema  $n$  vezes.

---

**Algoritmo 2** Exemplo parcial de uma classe para representar e calcular, palavras da Curva de Koch. Esta classe implementa o interface `LSystem`

---

```
public class KochCurve implements LSystem {
    // O que entender necessário
    public void setStart(String start) {
        // A sua implementação
    }
    public void addRule(Character symbol, String word) {
        // A sua implementação
    }
    public String iter(int n) {
        // A sua implementação
    }
}
```

---

Suponha que quer representar a Curva de Koch com uma classe específica. Pode seguir o esquema esboçado no algoritmo 2 e, para encontrar a sétima iteração, pode seguir o exemplo do algoritmo 3.

---

**Algoritmo 3** Exemplo de uso de uma classe que implementa `LSystem`.

---

```
...
LSystem kochCurve = new KochCurve();
kochCurve.setStart("F");
kochCurve.addRule('F', "F+F-F-F+F");
String word7 = kochCurve.iter(7);
...
```

---

**Resumindo**, as tarefas de definir Sistemas-L e gerar palavras ficam representadas por classes que implementem o interface `LSystem`.

## 3.2 Compilar Palavras Geradas para Programas Tartaruga

Para obtermos figuras a partir das palavras geradas pelos Sistemas-L é necessário considerar **primeiro** como é que as palavras são traduzidas em PTs e, **depois**, como é que esses programas produzem figuras. Isto é, precisamos de saber como **representar um programa** de forma a podermos gerar um a partir de uma palavra e, depois, interpretar esse programa como, por exemplo, um gráfico.

A linguagem dos PTs é especialmente simples pois um programa válido é apenas uma lista com instruções básicas. Nesta secção tratamos da representação dos PTs e na próxima tratamos do problema da sua execução.

## Representar Instruções

As **instruções** usadas nos exemplos anteriores são apenas de dois tipos: `forward(double)` e `turn(double)`. No entanto *temos de considerar a possibilidade de serem necessários outros tipos de instruções*, o que de facto acontece nos exemplos da [página da wikipedia](#) e também [desta página](#).

É (muito) comum esperar que um sistema possa desenvolver-se de formas não previstas quando é concebido e implementado. No caso do Java uma solução para este problema consiste em suportar *hierarquias* de classes.

A **representação das instruções** dos Programa Tartaruga assenta numa classe abstracta, `TurtleStatement`, raiz para as representações de todas as instruções, indicada no algoritmo 4.

Na especificação desta classe está considerado, desde já, que as instruções serão processadas por um certo interpretador, capaz de concretizar as ações adequadas a cada instrução. Por exemplo, desenhar uma linha reta de comprimento 10 quando processa a instrução `forward(10)`.

A **relação entre uma instrução e um interpretador** é suportada por um conjunto de métodos, todos com nome a começar por `run`, que fazem parte das classes `TurtleStatement` (algoritmo 4) e do interface `Interpreter` (algoritmo 8).

---

**Algoritmo 4** Classe `TurtleStatement` para representar instruções dos Programas Tartaruga. O método `run` serve para ligar a instrução a um interpretador.

---

```
public abstract class TurtleStatement {  
    public abstract void run(Interpreter interpreter);  
}
```

---

Para representar tipos concretos de instruções estende-se essa classe. Por exemplo, as instruções do tipo `forward(d)` podem ser representadas pela classe `Forward`, completamente implementada no algoritmo 5.

---

**Algoritmo 5** Classe `Forward`, descendente de `TurtleStatement`, para representar as instruções do tipo `forward(double)`. Nesta classe a concretização do método `run` usa o método `runForward` do interpretador.

---

```
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);  
    }  
}
```

---

O método `run` **tem de ser implementado em cada descendente direto** da classe `TurtleStatement` para que, **na classe descendente**, seja usado o método adequado do interpretador.

Note, no algoritmo 5, que `Forward.run` usa `interpreter.runForward(this)`, que, **conforme veremos mais tarde**, é o método de `Interpreter` específico para processar as instruções `Forward`.

## Representar Programas Tartaruga

A classe `TurtleStatement` representa qualquer instrução, independentemente de cada tipo particular. Portanto, é possível desenvolver código indiferente ao tipo específico de cada instrução. Em particular, os Programas Tartaruga podem ser representados pelo tipo `List<TurtleStatement>`.

Formalmente, um Programa Tartaruga (PT) é uma lista de “Instruções Tartaruga” dos seguintes tipos: `Forward(double)`, `Turn(double)`, `PenUp()`, `PenDown()` e, eventualmente, outros tipos como por exemplo, `Leap(double)` ou `Save()`.

Nesta fase temos definida uma **Representação Intermédia** dos Programas Tartaruga. Estes programas são representados por uma **estrutura de dados** (neste caso, listas povoadas por instâncias `TurtleStatement`) e isso permite-nos escrever **algoritmos que geram, analisam e processam programas**.

Consideremos o problema de traduzir uma palavra (isto é, uma `String`) para um PT. Como já sabemos que tipo é usado para representar estes programas, podemos pensar na assinatura de um método adequado para compilar uma palavra para um programa. Por exemplo, um método com assinatura

```
Vector<TurtleStatement> compile(String word)
```

Note que, de acordo com a documentação oficial do [Java 10](#), `Vector` é uma classe concreta descendente da classe abstrata `AbstractList` e que implementa o interface `List`. Portanto, a assinatura do método `compile` é compatível com a representação de PTs como `List<TurtleStatement>`.

## Compilar Palavras para Programas Tartaruga

Com a Representação Intermédia também podemos esboçar uma classe responsável pela compilação das palavras geradas por Sistemas-L para PTs.

No “esqueleto” da classe `Compiler` (no algoritmo 6) consideramos que, para compilar as palavras de certos Sistemas-L, pode ser conveniente pré-definir um ângulo (para as rotações) e uma distância (para os avanços).

O método `Compiler.addRule`:

- Serve para definir, uma a uma, as regras da compilação de uma letra para uma “Instrução Tartaruga”.



---

**Algoritmo 6** Descrição parcial da classe `Compiler`, usada para compilar uma palavra gerada por um Sistema-L para um Programa Tartaruga. Note que esta classe tem dois métodos com nome `compile` mas as assinaturas são diferentes.

---

```
public class Compiler { // *** Incomplete class specification ***
    public void addRule(Character letter,
                        TurtleStatement statement) { ... }
    protected TurtleStatement compile(Character 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;
    }
}
```

---

- Não é o método `LSystem.addRule(Character symbol, String word)`.

O fragmento de código do algoritmo 7 ilustra como se pode definir um compilador para a Curva de Koch e traduzir a sétima iteração para um Programa Tartaruga.

---

**Algoritmo 7** Exemplo de aplicação do interface `LSystem` com a classe `Compiler`. Inicialmente é definido um Sistema-L, com a palavra inicial e a (única) regra e um compilador com as regras comuns para a Curva de Koch. De seguida é calculada a sétima iteração, `word7` e o PT que resulta da compilação de `word7`.

---

```
...
LSystem kochCurve = new KochCurve();
kochCurve.setStart("F");
kochCurve.addRule('F', "F+F-F-F+F");

Compiler compiler = new Compiler();
compiler.addRule('F', new Forward(1));
compiler.addRule('+', new Turn(90));
compiler.addRule('-', new Turn(-90));

String word7 = kochCurve.iter(7);
Vector<TurtleStatement> program = compiler.compile(word7);
...
```

---

**Resumindo**, a tarefa de compilar palavras para PTs assenta na **representação das instruções** por classes descendentes de `TurtleStatement` e na **Representação Intermédia** de programas por listas de instâncias dessas classes.

Para **ligar a representação de instruções à respetiva interpretação** são requeridos os métodos `run`, que têm obrigatoriamente uma implementação específica em cada classe diretamente descendente de `TurtleStatement`.

A **compilação** propriamente dita é feita na classe `Compiler`. Esta classe define o processo da compilação pela adição de regras de compilação e compila palavras com o método `compile`.

### 3.3 Interpretar Programas Tartaruga como Gráficos

Agora resta saber como construir um gráfico usando um PT. De novo, **procuramos maximizar aplicações futuras assumindo o mínimo necessário**.

Neste caso consideramos que um PT pode, num caso, ser interpretado como uma imagem no ecrã usando uma certa biblioteca gráfica, noutro caso como um ficheiro gráfico, ou uma mensagem enviada por `http`, *etc*.

---

**Algoritmo 8** Interface `Interpreter` para definir interpretadores de PTs. São especificados métodos para ligar cada interpretador à instrução a ser executada. Aqui **estão previstas apenas instruções de Nível 0** (segundo a tabela 3). Para outros níveis é necessário acrescentar métodos adequados.

---

```
public interface Interpreter {  
    void run(List<TurtleStatement> program);  
    void runForward(Forward statement);  
    void runTurn(Turn statement);  
    void runPenUp(PenUp statement);  
    void runPenDown(PenDown statement);  
}
```

---

Para acomodar a possibilidade de diferentes destinos para a interpretação de PTs, a interpretação de um programa tem de ser feita em classes que implementem o interface dado no algoritmo 8.

Desta forma é possível desenvolver vários destinos (*backends*), todos compatíveis com o restante projeto. Um destino possível pode usar o *package* `galapagos` para construir um gráfico no ecrã. Outros destinos incluem o desenvolvimento de uma biblioteca para *Turtle Graphics* própria, a produção de ficheiros SVG, *etc*.

**Resumindo**, a **interpretação** de Programas Tartaruga está organizada pelo interface `Interpreter`, que impõe um conjunto de métodos `run***` que ligam o interpretador aos diferentes tipos de instruções.

Os próprios PTs são executados pelo método `run`, que pouco mais fará do que executar sequencialmente as instruções do programa dado.

**É em cada um dos métodos `run***` que o efeito de cada instrução fica definido.** Por exemplo, será no método `runTurn(Turn statement)` que a "tartaruga" do *package* `galapagos` é devidamente rodada.

No entanto, este interface **não assume nada sobre o destino da interpretação dos programas**, o que **permite definir destinos (muito) diferentes**, com implementações diferentes, cada uma especializada num certo destino.

## 4 Avaliação

O seu trabalho é avaliado pelos critérios indicados a seguir e considera vários Sistemas-L, conforme a tabela 3.

1. **Geração de Iterações dos Sistemas-L** com classes que implementam o interface `LSystem`.
  - (a) Por **cinco valores**, implemente classes para gerar palavras de qualquer um dos Sistemas-L de nível 0 indicados acima.
  - (b) Por **meio valor adicional**, defina uma classe capaz de representar qualquer Sistema-L de Nível 1, mesmo que não esteja listado acima.
  - (c) Por **meio valor adicional**, defina uma classe capaz de representar qualquer Sistema-L de Nível 2, mesmo que não esteja listado acima.
2. **Representação de Instruções dos Programas Tartaruga** com classes descendentes de `TurtleStatement`.
  - (a) Por **cinco valores**:
    - i. Implemente as classes `Forward`, `Turn`, `PenUp` e `PenDown` para representar as instruções de Nível 0.
    - ii. Complete a implementação da classe `Compiler`.
  - (b) Por **meio valor adicional**, implemente a classe `Leap`, para representar as instruções de Nível 1.

A instrução `leap(d)` é semelhante a `forward(d)` mas a tartaruga **não deixa rasto**, isto é `leap(d) = {penUp(); forward(d); penDown();}`.
  - (c) Por **meio valor adicional**, implemente as classes `Save` e `Restore` para representar as instruções de Nível 2.

A instrução `save()` guarda a posição e a orientação atuais da tartaruga numa *pilha*. Esses valores são recuperados com a instrução `restore()`, que repõe a tartaruga na posição e orientação guardadas no topo da *pilha*. Note que a pilha é LIFO e que o topo é removido quando é lido.

Tabela 3: Classificação dos tipos de instruções (e respectivos PTs). Estão também indicados alguns Sistemas-L obtidos com instruções desses tipos, segundo a [página da wikipedia](#).

Nível	Instruções	Sistemas-L
Nível 0	<code>forward</code> , <code>turn</code> , <code>penUp</code> , <code>penDown</code>	Kock Curve, Koch Snowflake, Sierpinski Triangle, Sierpinski Arrowhead, Dragon Curve
Nível 1	Nível 0 + <code>leap</code>	Cantor Set
Nível 2	Nível 1 + <code>save</code> , <code>restore</code>	Fractal Plant

3. **Interpretação dos Programas Tartaruga** com classes que implementam o interface `Interpreter`.

- (a) Por **cinco valores**, implemente um interpretador que mostra a imagem num dispositivo gráfico.
- (b) Por **meio valor adicional**, implemente um interface gráfico que permita escolher o Sistema-L (de uma lista pré-definida que inclui os Sistemas-L de nível 0) e o número de iterações.
- (c) Por **meio valor adicional**, implemente um interpretador que gera um ficheiro no formato `SVG`. Veja o tutorial “[An SVG Primer for Today’s Browsers](#)”.

4. **Mais Valores Adicionais.**

- (a) Por **um valor adicional**, o interface gráfico permite definir as regras do Sistema-L e as regras do Compilador.
- (b) Por **um valor adicional**, o interpretador centra e dimensiona automaticamente a imagem.