Relatório: Fractais e L-systems

Gabriel Henrique Campos Medeiros

October 2025

1 Introdução

Um Sistema de Lindenmayer (L-system) é um método para definir estruturas fractais a partir de regras que geram cadeias sucessivas a cada iteração. Tais sistemas são definidos por: um conjunto de símbolos, denominado alfabeto; um axioma, que representa a produção ou o estado inicial do sistema; e um conjunto de regras de reescrita, que determinam a transformação de cada variável.

A natureza recursiva desse problema o torna particularmente interessante para ser abordado no paradigma da programação funcional, pois permite a exploração de conceitos e técnicas fundamentais a esse paradigma. Com o propósito de aprimorar a prática na linguagem de programação funcional Elixir, este trabalho propõe a implementação de funções associadas aos L-systems.

2 Instruções de Uso e Interface da Aplicação

ara a utilização dos recursos implementados neste trabalho, são necessários um compilador Elixir e um compilador Python. Embora o desenvolvimento em Python não constitua o foco principal, ele é requerido para funcionalidades específicas. O acesso às funções implementadas é obtido executando o comando iex -S mix no diretório onde se encontra o arquivo "mix.exs".

O módulo implementado expõe três funções públicas, acessíveis a usuários: `fractal/2`, `turtle/2` e `checkprod/3`. A seguir, detalha-se a assinatura e funcionalidades das funções, além de como os L-systems são representados no programa.

2.1 Definição de L-System

O L-system é formalmente representado por uma tupla que encapsula seus atributos, conforme a estrutura $\langle \mathrm{v}, \mathrm{w}, \mathrm{p} \rangle$.

- Os Símbolos (v) e o Axioma(w) são representados como listas de *strings*.
- As Regras de Produção (p) são representadas como um mapa , onde as chaves são os símbolos a serem substituídos e os valores associados são a substituição a ser feita, que devem também ser representadas como listas de strings.

2.2 Funções Públicas

2.2.1 `fractal(`n`, l_system)`

- **Assinatura:** `fractal(integer(), tuple())` → `list()`
- **Argumentos:**
  1. n: Um número natural que especifica o número de iterações.
  2. `l_system`: A tupla contendo a definição do L-system.
- **Funcionalidade:** Esta função computa e retorna a cadeia de produção do L-system após a n-ésima iteração. O caso-base, $n = 0$, retorna o Axioma.

2.2.2 `turtle(`n`, l_system)`

- **Assinatura:** `turtle(integer(), tuple())` → `:ok`
- **Argumentos:** Recebe os mesmos argumentos que a função `fractal/2`.
- **Funcionalidade:** Retorna um programa escrito em `Python`, que utiliza a biblioteca `turtle graphics`. A execução subsequente deste programa gera uma representação gráfica baseada na interpretação dos símbolos produzidos pelo L-system na n-ésima iteração.
- **Símbolos Interpretados pelo Turtle Graphics:** A funcionalidade gráfica está limitada aos seguintes símbolos:
  - `F`: Avança 20 passos para frente com a caneta na cor azul.
  - `G`: Avança 30 passos para frente com a caneta na cor vermelho.
  - `+x`: Gira no sentido anti-horário em x graus.
  - `-x`: Gira no sentido horário em x graus.
  - `[`: Salva a posição e a angulação atuais do cursor (função de pilha).
  - `]`: Restaura a posição e a angulação salvas mais recentemente na pilha.

  Quaisquer outros símbolos presentes no L-system serão processados na expansão da cadeia, mas serão ignorados durante a geração do programa python para o turtle graphics.

2.2.3 `checkprod(x, n, l-system)`

- **Assinatura:** `checkprod(string(), integer(), tuple())` → `boolean()`
- **Argumentos:**
  1. n: O número natural de iterações.
  2. `l_system`: A tupla contendo a definição do L-system.

3. x: Uma produção de teste.

- **Funcionalidade:** Verifica a equivalência entre a produção de teste x e a cadeia de produção do L-system na n-ésima iteração. Retorna true se a equivalência for verificada e false caso contrário.

Você pode achar o projeto em: Repositório do Projeto L-System em Elixir.

3 Implementação

3.1 Problema 1: Fractal

A base fundamental deste trabalho, que serve de apoio para as duas funções subsequentes, é a função fractal. O passo inicial da implementação consistiu em definir a representação dos L-systems em Elixir. Optou-se por representar as produções como listas, uma escolha que provê conveniência na implementação, a ser explicitada adiante. Alternativamente, as produções poderiam ser representadas como strings, o que, contudo, demandaria etapas adicionais de tradução e pattern-matching.

A função fractal aceita como argumentos:

1. O número natural n, que indica a iteração final desejada.

2. A tupla que representa o L-system, no formato ⟨Símbolos,axioma,regras⟩.

Por uma questão de formalidade do design do sistema, o campo de símbolos é mantido na tupla; no entanto, este não é utilizado na lógica de fractal, sendo representado com o prefixo _ para indicar seu descarte (ignore). O axioma deve ser fornecido como uma lista de strings, e as regras de produção como um mapa. Assume-se que o usuário fornecerá um L-system válido.

Para auxiliar o processo recursivo, foi introduzida uma função auxiliar, denominada aux_fractal. Esta função tem o propósito de gerenciar o número de iterações restantes e aplicar iterativamente as regras de produção.

- **Caso Base:** Quando n=0, o axioma é retornado.

- **Caso Recursivo:** Para n>0, a regra de substituição é aplicada à produção atual, e a função aux_fractal é chamada recursivamente com n−1 até chegar a n=0, onde a função retornara a só a produção no momento.

A função aux_fractal recebe como argumentos uma lista (a produção atual), um número natural correspondente às iterações restantes, e o mapa de regras de produção. Dentro da função fractal, a aux_fractal é invocada com o axioma (o estado inicial), o número total de iterações e as regras.

A aplicação das regras de produção é realizada por uma função dedicada, denominada substitui. Esta função recebe como argumentos a produção (lista de tokens) e as regras de reescrita. Neste ponto, a escolha de representar as

produções como listas se mostra vantajosa, pois permite o uso eficiente da função Enum.map.

A lógica da função substitui é concisa: para cada token na lista de produção, consulta-se o mapa de regras de reescrita utilizando Map.fetch. Caso o token seja uma chave presente no mapa, ele é substituído pelo valor correspondente. É crucial notar que símbolos de controle (e.g., +, -, [e]) não possuem regras de reescrita associadas e, portanto, são mantidos inalterados na produção.

A função substitui retorna a lista resultante da aplicação das regras, que é então retransmitida à aux_fractal. Eventualmente, a aux_fractal atinge seu caso base (n=0), e a produção final do L-system é obtida.

```
def fractal(n, {_v,w,p}) when is_integer(n) and n>= 0 do
    aux_fractal(w,n,p)
  end

def aux_fractal(producao, 0, _p), do: producao

def aux_fractal(producao, n, p) do
    continua = substitui(producao, p)
    aux_fractal(continua, n-1, p)
end

def substitui(producao, p) do
    Enum.flat_map(producao, fn x ->
      case Map.fetch(p,x) do
        {:ok, prod} -> prod
        :error -> [x]
      end
    end)
end
```

Listing 1: Exemplo da função em elixir

3.2 Problema 2: Turtle

A primeira etapa da função turtle, que aborda o Problema 2, consiste em obter a cadeia de produção do L-system na iteração desejada. Uma vez que esta etapa é idêntica à funcionalidade central da função fractal (Problema 1), será omitida a descrição do processo de geração da cadeia.

Com a produção do L-system finalizada, a qual deve estar composta exclusivamente pelos símbolos de controle aceitos (F, G, +x, -x, [,]), o objetivo passa a ser a geração de um arquivo executável em Python. Este arquivo é responsável por renderizar a geometria fractal por meio da biblioteca turtle graphics.

```
def turtle(n, {_v,w,p}) when is_integer(n) and n>= 0 do
   producao = aux_fractal(w,n,p)
   codigo = gerador_codigo(producao)

   case File.write("turtleCode.py", codigo) do
     :ok -> {:ok, "Arquivo Gerado"}
     {:error, erro} -> {:error, "Deu problema. Erro: #{erro}"}
   end
```

```
  end
```

Listing 2: Exemplo da função turtle

Para a construção do arquivo em Python, o texto correspondente é gerado primeiramente em Elixir.Para a construção do arquivo em Python, o texto correspondente é gerado primeiramente em Elixir.

A primeira seção desse texto constitui o header. O header é composto pelos comandos necessários para a correta execução do programa Python, incluindo:

1. A importação das bibliotecas requeridas para a execução do programa gráfico.
2. O ajuste das configurações iniciais da biblioteca turtle graphics.
3. A definição de variáveis globais que serão utilizadas no decorrer do programa.

```
defp gerador_codigo(producao) do
  header =
    """
    import turtle
    import math

    t = turtle.Turtle()
    t.screen.setup(width=1920, height=1080)
    t.speed(0)
    t.penup()
    t.goto(0,-300)
    t.setheading(90.0)
    t.pendown()

    F_LEN = 20.0
    G_LEN = 30.0
    PEN_STACK = []

    """

  commands = Enum.map(producao, &producao_to_python/1)
  body = Enum.join(commands, "\n")

  footer = "\n\nt.screen.exitonclick()"

  header <> body <> footer
end
```

Listing 3: Exemplo da função gerador$_codigo$

Após a definição do header, o próximo passo consiste na criação do corpo do programa Python. Para tanto, é necessário traduzir cada token da cadeia de produção do L-system para uma string de comando válida em Python. Com esse objetivo, foi desenvolvida uma função auxiliar denominada producao_to_python. Esta função recebe a produção (uma lista de strings) e aplica, a cada posição

da lista, uma estrutura de controle (case) que mapeia o token para o comando Python equivalente.

Em relação aos tokens especiais:

1. Símbolos de Rotação (+ e -): Estes símbolos são projetados para vir sempre acompanhados de um valor numérico que representa o ângulo. O ângulo é extraído por meio de um pattern matching, utilizando o formato $+ <> rest$ (ou $- <> rest$), onde a variável rest captura o valor do ângulo.
2. Símbolo de Pilha ("["): Este token é traduzido para um comando que salva a posição e a angulação atuais do cursor na estrutura de dados de pilha (PEN_STACK).
3. Símbolo de Retorno ("]"): Este token corresponde a um comando que remove o valor mais recente da pilha e restaura o cursor para a posição e angulação armazenadas.

A função producao_to_python concatena as strings de comandos Python resultantes, formando o corpo completo do programa a ser executado.

```
1   defp producao_to_python(prod) do
2   case prod do
3     "F" -> "t.pencolor('blue')\nt.forward(F_LEN)"
4     "G" -> "t.pencolor('red')\nt.forward(G_LEN)"
5     "[" -> "PEN_STACK.append((t.heading(), t.pos()))\nt.penup()"
6     "]" -> """
7           if PEN_STACK:
8             heading, pos = PEN_STACK.pop()
9             t.setheading(heading)
10            t.goto(pos)
11            t.pendown()
12          """
13    "+"<> rest -> angle = String.to_integer(rest)
14          "t.right(#{angle})"
15    "-"<> rest -> angle = String.to_integer(rest)
16          "t.left(#{angle})"
17    " " -> ""
18    _ -> "# Comando nao reconhecido: #{prod}"
19  end
20 end
```

Listing 4: Exemplo da função producao_to_python

Por fim, todos os segmentos de código gerados (o header e o corpo do programa traduzido) são concatenados em uma única string. Utiliza-se a função File.write para criar o arquivo de saída, garantindo que o conteúdo seja sobrescrito a cada execução.

Para a visualização do gráfico resultante, o usuário deve executar o arquivo gerado, denominado turtleCode.py, utilizando um compilador Python.

3.3 Problema 3: CheckProd

O terceiro problema abordado, implementado na função fractal_check, é de natureza direta: verificar se uma determinada cadeia de produção fornecida é

equivalente à produção gerada por um L-system específico na n-ésima iteração.

A solução para este problema foi implementada através de uma comparação direta entre a string de produção recebida como argumento e o resultado da chamada à função fractal (utilizando os mesmos parâmetros de iteração n e o L-system). A função fractal_check retorna um valor booleano indicando a equivalência ou não das duas produções.

```
1    def checkprod(producao, n, {v, w,p}) do
2        producao == fractal(n, {v,w,p})
3    end
```

Listing 5: Exemplo da função Checkprod

4 Conclusão

A implementação dos Sistemas de Lindenmayer (L-systems) constituiu uma experiência prática interessante em Elixir. O projeto serviu como um excelente estudo de caso para exercitar e solidificar o uso de recursividade e pattern matching, elementos centrais do paradigma funcional.

Além do núcleo de processamento das cadeias, o trabalho também exercitou o aprendizado prático sobre o sistema de I/O e criação de arquivos do Elixir.

É válido também mencionar como foi importante pensar bem em como implementar cada detalhe do trabalho antes de escrever de fato o código. A maneira como foi representada o L-system neste trabalho acabou por facilitar bastante alguns passos.

Em relação específicamente ao problema 3, tenho dúvidas se interpretei algo de maneira errada pois ele é realmente muito simples.