Universidade Federal
do Espírito Santo

Centro Tecnológico

Departamento de Informática

Interpretador de Sistemas de Lindenmayer

Autor:
Vinicius Arruda

Professor:
Thomas W. Rauber

29 de junho de 2015

Resumo

Trabalho da disciplina de Estrutura de Dados I, que consiste no desenvolvimento de um interpretador de sistemas de Lindenmayer, utilizando conceitos de estrutura de dados e tipos abstratos de dados para a elaboração do trabalho.

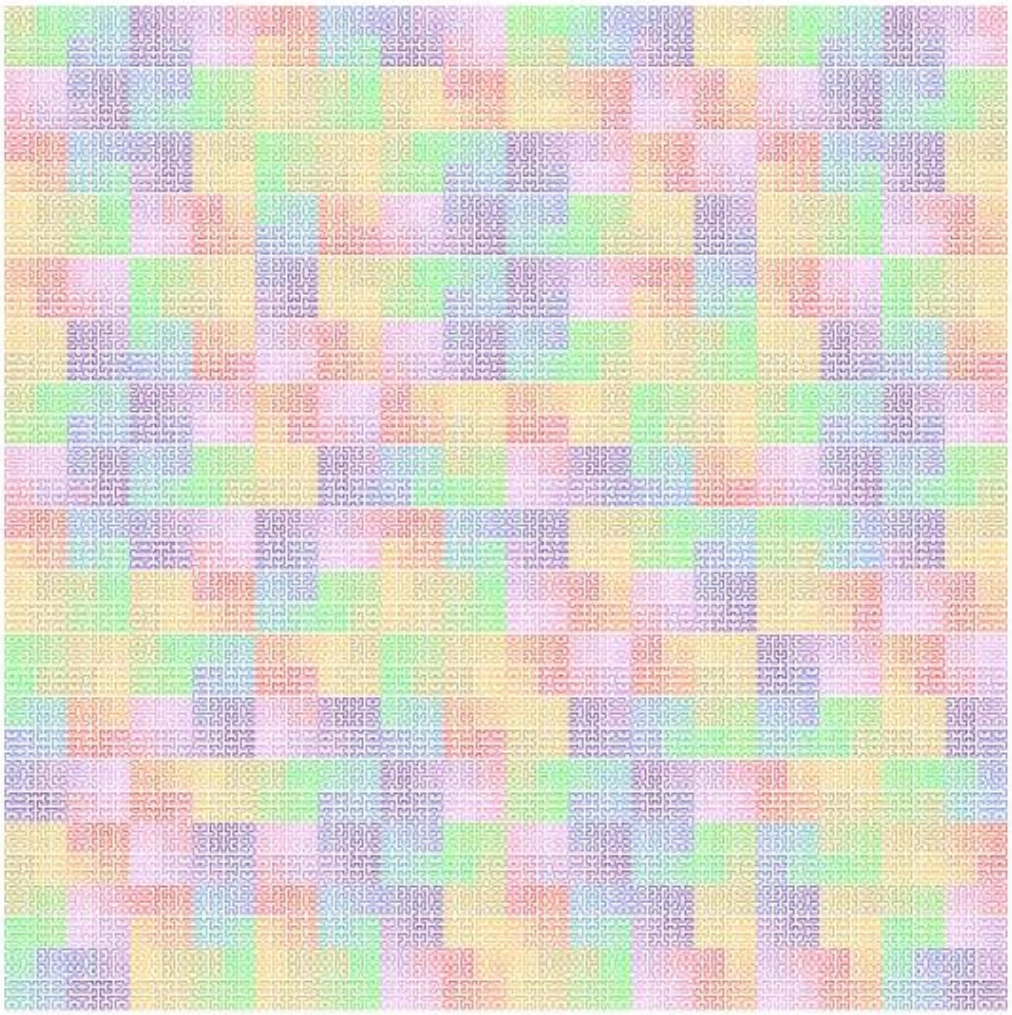

Figura 1: Hilbert.

Imagem gerada a partir do interpretador desenvolvido.

1 Introdução

Ao implementar um sistema, frequentemente programadores se deparam com o uso de estruturas de dados que na maioria das vezes são estáticas como os vetores e matrizes por exemplo. Porém, ao implementar um sistema mais complexo, surge a necessidade de se trabalhar com tipos dinâmicos e mais específicos para aquele problema.

O ideal é abstrair este tipo para que torne a vida do programador mais fácil. Essa é a idéia do tipo abstrato de dados, comumente conhecido como TAD.

O sistema aqui implementado, faz o uso de vários TADs para abstrair a manipulação dos dados do próprio programador, fazendo com que o desenvolvimento do sistema seja feito em camadas, de maneira mais organizada.

Para a implementação dos TADs, foram utilizados estruturas de dados estáticas, alocação dinâmica de memória, tipo genérico, funções de callback e os conceitos de lista encadeada, árvore e pilha.

2 Objetivo

Aplicar o conhecimento adquirido na disciplina de Estruturas de Dados I para representar e manipular informações estruturada por linguagem de programação de alto nível na elaboração de um interpretador de sistemas de Lindenmayer.

3 Ferramentas

O interpretador foi implementado na linguagem de programação C. Para a compilação foi utilizado o compilador GCC versão 4.7.2 em uma máquina com o sistema operacional Debian GNU/Linux 7.8 (wheezy). O código foi escrito utilizando o editor de texto gedit versão 3.4.2. Para a depuração do programa, foi utilizado a ferramenta Valgrind versão 3.7.0.

4 Metodologia

O desenvolvimento do interpretador foi dividido em três etapas, que se basearam em construir o parser, a árvore de filhos variados para aplicar as regras e os comandos para as partes 2 e 3 do trabalho.

4.1 O Parser

A função do parser e de sua função auxiliar para a leitura do arquivo, consiste em interpretar do arquivo as informações para preencher as estruturas *preamble* e *productions* que armazenarão as informações necessárias para a geração do *l-system*.

As estruturas *preamble* e *productions* possuem a seguinte forma:

```
typedef struct
{
    int angle;
    long int order;
    double rotate;
} Preamble;
```

```
typedef struct
{
     char* axiom;
     List* rules
} Productions;
```

Onde *List* é um tipo definido em:

```
struct list
{
    void* info;
    struct list* next;
};

typedef struct list List;
```

Que é a estrutura de uma lista genérica encadeada utilizada para manipular *rules*, que é uma lista de regras onde cada elemento é definido por:

```
typedef struct
{
    char p;
    char* s;
} Rule;
```

Onde p é um símbolo único e s uma string que seguem a igualdade $p = s$.

Quando a função de leitura do arquivo *getStrings* lê uma linha do arquivo, ela chama a função *parser* para analisar esta linha.

Ao identificar uma palavra chave, uma função própria para manipular esta informação é chamada, preenchendo seu devido campo em *preamble* e *productions*.

Ao encontrar um símbolo, uma função para manipular as regras é chamada, encadeando uma nova regra *Rule* na lista *rules* da estrutura *productions*.

4.2 A árvore

A implementação do TAD árvore consistiu, primeiramente, na representação da informação, que consiste em um nó *Tree* que possui sua informação, um ponteiro para seu nó irmão, um ponteiro para seu primeiro filho e um ponteiro para seu último filho.

A estrutura da árvore possue a seguinte forma:

```
typedef struct tree
{
    char info;
    struct tree* firstChild;
    struct tree* lastChild;
    struct tree* next;
} Tree;
```

O motivo de um ponteiro extra, apontando para o último filho, se dá para otimizar no encadeamento dos nós, pois pela estrutura do interpretador, a informação deve ser encadeada na árvore da esquerda para a direita, tendo sempre que percorrer até o final da lista para encadear no último nó. Uma outra solução seria armazenar um ponteiro temporário para o final da lista a medida que as regras eram interpretadas e encadeadas, porém como o trabalho foi desenvolvido em camadas, foi preferido criar as estruturas de dados abstratas e ir trabalhando com essas abstrações.

A Figura 2 mostra um diagrama do TAD árvore elaborado.

4.2.1 Manipulação da árvore

Inicialmente, é criado a raíz da árvore, com a informação simbólica *'\0'*. A raíz da árvore possui uma lista de filhos que é formado pelos caracteres do axioma. A partir do axioma, é aplicado as regras contidas em *rules*.

A aplicação das regras se baseia em percorrer as folhas da árvore, e para cada folha, verificar se sua informação é igual a algum p da lista de regras *rules*, se sim, inserir naquela folha uma lista de filhos onde cada nó contém um caracter da string s. Este processo é repetido *order* vezes.

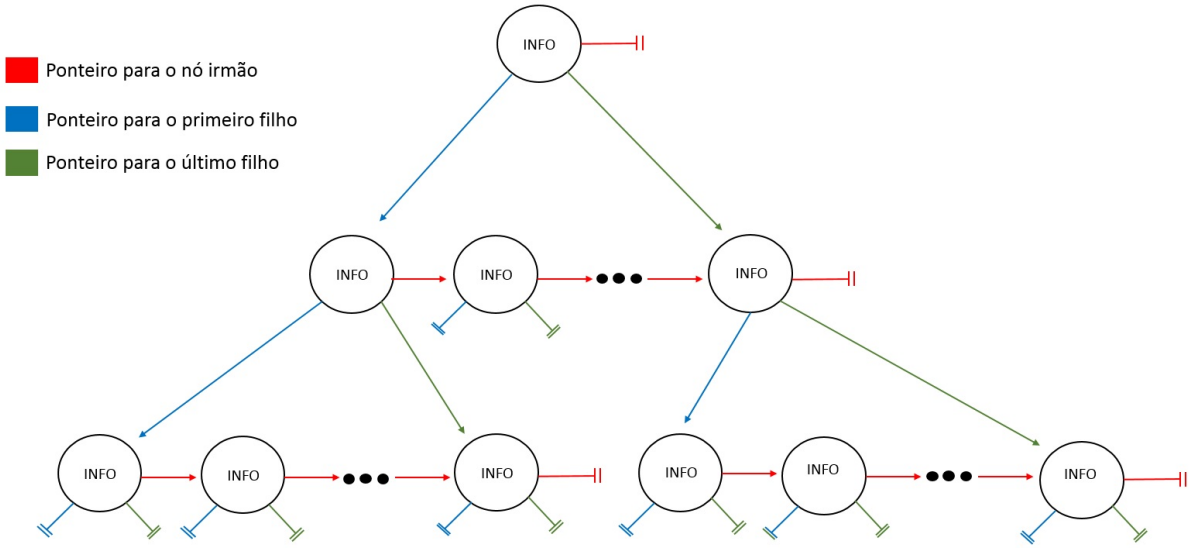

Figura 2: Diagrama do TAD árvore

Ao final de *order* aplicações das regras, a string final é recolhida a partir de todas as folhas da árvore.

4.3 Os comandos

O desenvolvimento dos comandos foram divididos em duas etapas. A primeira é descrita na parte 1 do trabalho e a segunda etapa é descrita nas partes 2 e 3 do trabalho.

4.3.1 Parte 1

A string final, que é a concatenação das informações das folhas da árvore, é formada por qualquer caracter, podendo ser letras, números e símbolos, porém, para a primeira parte do trabalho, foi implementado apenas os comandos representados pelas letras F e G, e pelos símbolos +, -, [e].

Devido a possibilidade de haver símbolos ou caracteres indesejados além dos implementados, a string final é passada por um filtro eliminando-os. Após o filtro, a string é impressa no arquivo de saída, junto ao preâmbulo.

Um exemplo de resultado obtido foi o triângulo de *Sierpinski* na Figura 3, gerado a partir do seguinte *grammar*:

```
order 7
angle 3
axiom F

F = FXF
X = +FLXRF-FLXR<F-F>LXR<F+
L = >6
R = <6
```

A pesar de não ter sido implementado os comandos $<$ e $>$, a figura ainda é gerada devido ao filtro que é aplicado à string final, porém em preto e branco.

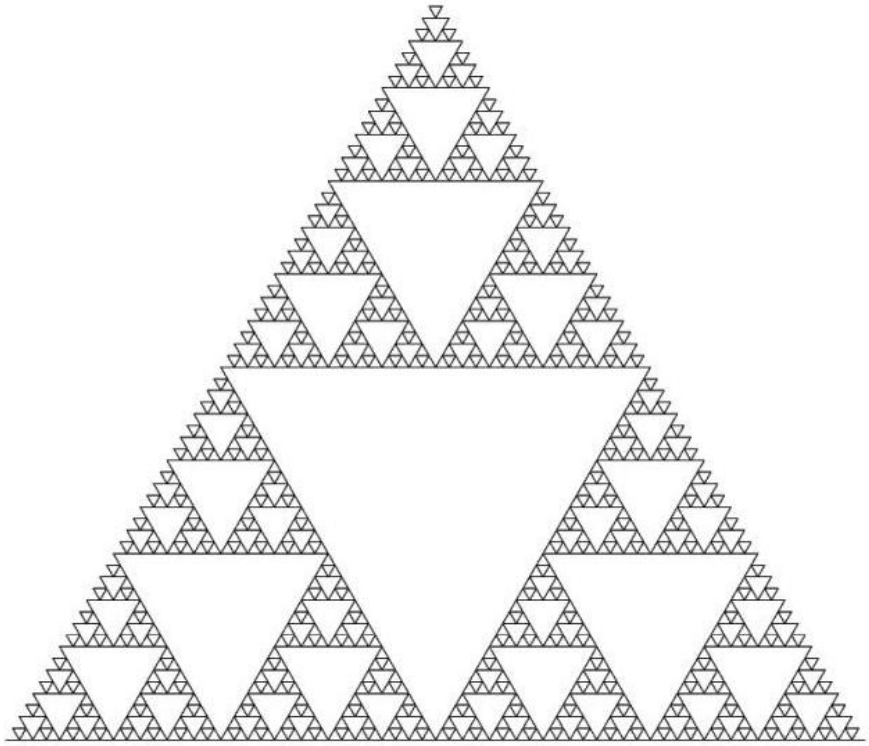

Figura 3: Triângulo de Sierpinski

Um outro exemplo de resultado obtido a partir do *grammar* a seguir é a Figura 4, que ficou indesejada, pois o comando | não foi implementado para a primeira parte do trabalho, e como este comando realmente influencia como a imagem é gerada, ela ficou desconfigurada em relação a desejada.

```
angle 6
order 5
axiom X

X = +FF-YFF+FF--FFF|X|F--YFFFYFFF|
Y = -FF+XFF-FF++FFF|Y|F++XFFFXFFF|
F = GG
G = G>G
```

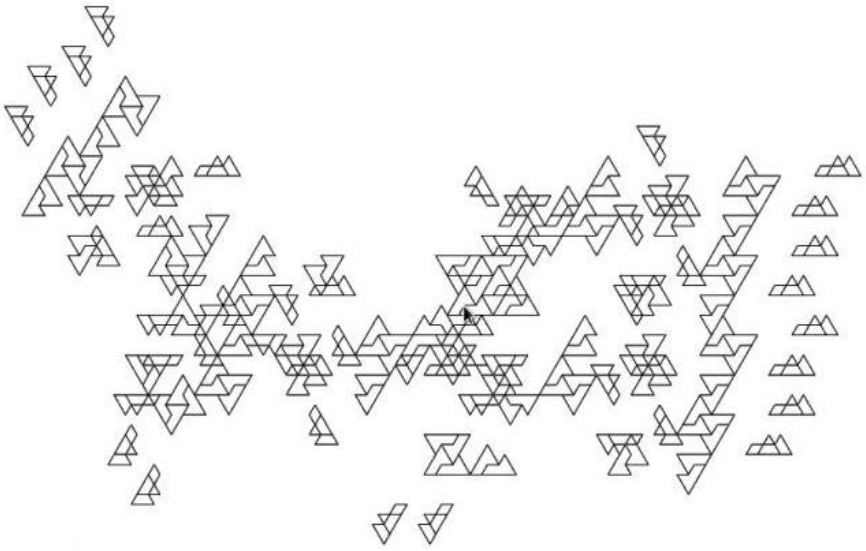

Figura 4: Sphinx (imagem indesejada)

Como para a primeira parte não foi implementado alguns comandos descritos no *grammar*, a imagem não saiu como desejada, gerando apenas a resposta dos comandos que passaram pelo filtro.

4.3.2 Partes 2 e 3

Para a segunda e terceira parte, foi implementado uma estrutura de dados baseada no princípio *LIFO*, que possui como representação da informação a seguinte estrutura:

```
typedef struct                      Estrutura Turtle.
{
    double x;                       Posição x.
    double y;                       Posição y.
    double orientation;             Orientação (Ângulo).
    double length;                  Comprimento da linha.
    int color;                      Cor da linha.
    int counterclockwise;           Flag para sentido horário ou anti-horário.
} Turtle;
```

Esta estrutura, modela uma tartaruga, com sua posição atual, orientação, comprimento do passo, sentido em que ela gira e a cor que ela risca quando abaixa a caneta.

Antes de passar pelo último filtro, a string final gerada pela parte 1 é passada para a função *secondPart()*, que é interpretada caracter a caracter, gerando os comandos em postscript. Estes comandos são desenhados pela tartaruga, e são interpretados de acordo com a lista a seguir:

- O comando *F* é convertido para a string *n x0 y0 m x1 y1 l s*, e a posição em *Turtle* é atualizada.
- O comando *G* apenas atualiza a posição de *Turtle*.
- Os comandos *+* e - atualizam a orientaçao de *Turtle*.
- O comando *[* salva a posição atual e todas as outras características de *Turtle* naquele instante.
- O comando *]* recupera a última posição com as características desta posição e gera a string *r g b setrgbcolor*, onde *r*, *g* e *b* são os tons de vermelho, verde e azul da cor descrita no campo *color* de *Turtle*.
- Os comandos *<*, *>* e *c* atualizam a cor descrita no campo *color* de *Turtle*.
- O comando @ atualiza o comprimento do passo da tartaruga, atualizando o campo *length* de *Turtle*.
- O comando *!* atualiza o sentido em que a tartaruga irá girar, atualizando o campo *counterclockwise* de *Turtle*.
- O comando | gira a tartaruga 180 graus, atualizando o campo *orientation* de *Turtle*.

A implementação da cor se baseou em montar uma paleta de 256 cores, passando pelas cores marrom, verde, azul, anil, violeta, vermelho, laranja, e voltanto ao marrom, em degradê. Para montar a paleta, foi utilizado como auxilio uma ferramenta disponível no site `www.strangeplanet.fr`[1].

Um exemplo de resultado obtido foi a Sphinx na Figura 5, gerada a partir do seguinte *grammar*:

```
angle 6
order 5
axiom X

X = +FF-YFF+FF--FFF|X|F--YFFFYFFF|
Y = -FF+XFF-FF++FFF|Y|F++XFFFXFFF|
F = GG
G = G>G
```

A Figura 5 é a imagem desejada para a Figura 4.

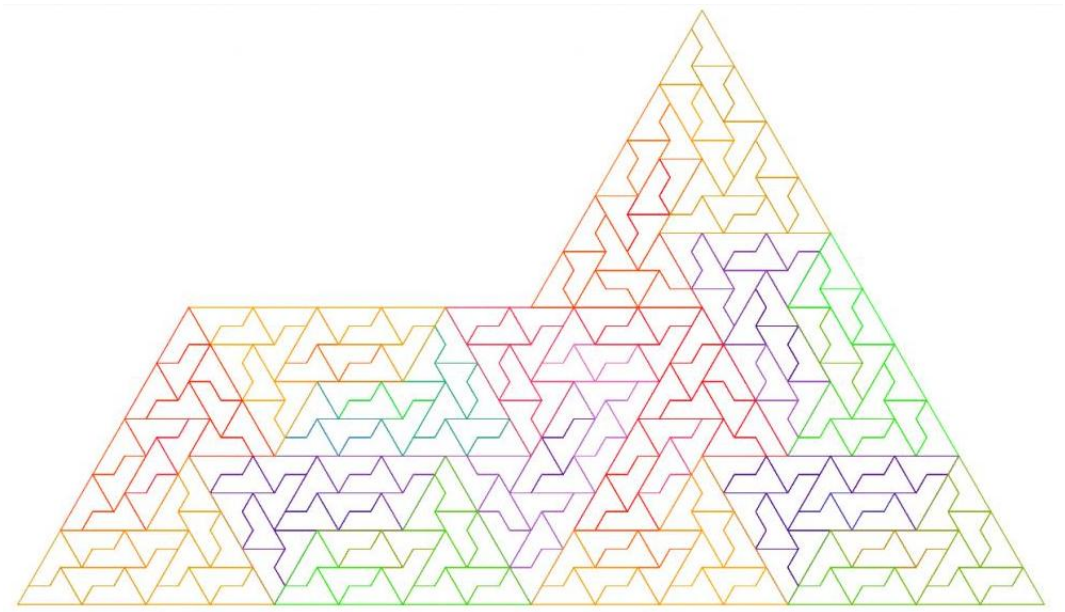

Figura 5: Sphinx

[1]Para o link completo da paleta de cores gerada, ver Referência Bibliográfica 4

5 Resultados e Avaliação

Uma série de arquivos contendo os *grammars* foram copiados do applet do `www.cgjennings.ca/toybox/lsystems/` para teste.

O resultado pode ser visto nos arquivos postscript que estão nas pastas *Parte1-PS* e *Parte2&3-PS* que estão dentro da pasta *Testes*.

A ferramenta valgrind foi frequentemente utilizada durante o desenvolvimento do sistema, sendo de grande ajuda para a depuração do programa.

6 Referências Bibliográficas

1. Livro Introdução a Estrutura de Dados. Autores: Waldemar Celes, Renato Cerqueira e José Lucas Rangel.

2. `http://www.cgjennings.ca/toybox/lsystems/`

3. `http://algorithmicbotany.org/`

4. `http://www.strangeplanet.fr/work/gradient-generator/?c=257:B8870B:00FF00:4169E1:4C0082:EE82EE:FF0000:FFA600:B8870B`

5. `http://www.mat.ufmg.br/~regi/topicos/intlat.pdf`

6. `http://ctan.mirrorcatalogs.com/info/symbols/comprehensive/symbols-a4.pdf`

7. `http://tex.stackexchange.com`