Zpravodaj Československého sdružení uživatelů TeXu

Michel Charpentier

Programujeme L-systémy v PostScriptu

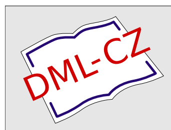
Zpravodaj Československého sdružení uživatelů TeXu, Vol. 22 (2012), No. 1, 9–19

Persistent URL: `http://dml.cz/dmlcz/150198`

Terms of use:

© Československé sdružení uživatelů TeXu, 2012

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.



This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* `http://dml.cz`

Programujeme L-systémy v PostScriptu

Michel Charpentier

Abstrakt

I když se PostScript tradičně považuje za formát souborů pro popis grafiky, jedná se ve skutečnosti o plnohodnotný programovací jazyk rozšířený o grafické funkce. Jeho vyjadřovací schopnosti sahají mnohem dál než pouhý popis vektorové grafiky. PostScript umožňuje naprogramování řady různých druhů vypočtů, včetně složitých aritmetických operací. V tomto článku ukážeme jak používat rekurzivní funkce v PostScriptu k implementaci skupiny přepisovacích systémů nazývaných L-systémy. Pomocí těchto systémů můžeme psát jednoduché programy v PostScriptu, které kreslí jak klasické fraktály tak i zajímavé obrázky připomínající rostliny.

Klíčová slova: PostScript, L-systémy, fraktály.

PostScript as a Programming Language

PostScript [1] is a graphical description language which was widely used in the graphics and typesetting worlds but is slowly being supplanted by the newer *Portable Document Format* (pdf). One fascinating thing about PostScript is that, in addition to its graphical capabilities, it possesses many of the features found in more traditional programming languages, like variables, conditional and loops (which pdf does not). Indeed, there is enough in PostScript to compute everything that is computable (in more pedantic terms, the language is said to be "Turing-complete").

It is always fun to learn a new programming language and, in my case, it always begins by calculating prime numbers.[1] Although it is possible to use PostScript to generate tables of prime numbers—the fun[2] part being that the numbers are computed by the *printer*—the language is first and foremost a graphical language, which begs the question: What can be done with prime numbers *and* graphics? One answer is the Ulam Spiral [10], which places integers along a (squarish) spiral and marks prime numbers with dots to display interesting patterns.

Fig. 1 shows Ulam's Spiral with a dot for each prime number, in which diagonal patterns start to emerge. Fig. 2 shows a closeup of the center of the spiral. Both pictures are generated using a PostScript program [4] that draws the spiral, the

[1] I grew tired of saying *Hello* to a world that never said *Hello* back.

[2] It stops being fun as soon as there are enough people waiting for urgent printouts while the main department printer is busy crunching prime numbers.

DOI: 10.5300/2012-1/9

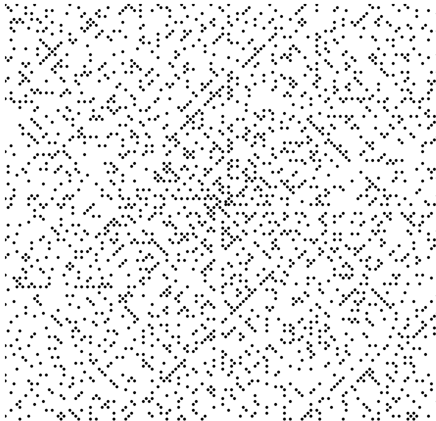


Figure 1: Ulam's Spiral.

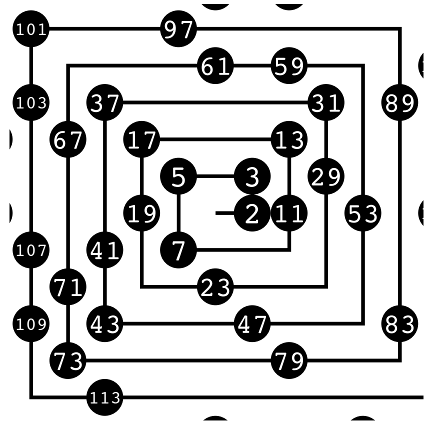


Figure 2: Ulam's Spiral (closeup).

dots, the numbers inside the dots and, more importantly, implements its own primality testing. Fig. 3 shows how trial division can be written in PostScript to implement a `prime?` function that tests whether a number is prime. PostScript also has arrays, which makes it possible to calculate prime numbers with the sieve of Eratosthenes, but the code is a bit longer (see [4] for details).

```
/prime? {
  /n exch def
  n 1 eq {
    false % 1 is not prime
  }{
    n 2 mod 0 eq {
      n 2 eq
    }{
      /divide? false def
      3 2 n sqrt {
        n exch div dup ceiling eq
          {/divide? true def exit} if
      } for
      divide? not
    } ifelse
  } ifelse
} bind def
```

Figure 3: Primality testing by trial division.

Without getting too much into PostScript syntax for now, a few things are worth noting. We see that the language has all the usual good stuff, like Booleans (`true`, `false`, `not`), conditionals (`if`, `ifelse`), loops (`for`, used here from 3 to

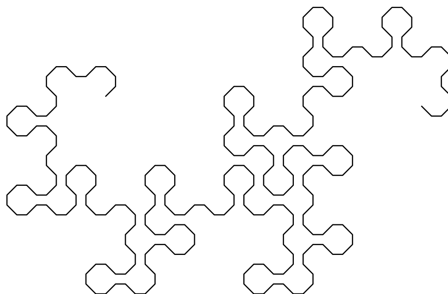
$\sqrt{n}$ by 2 increments), tests (`eq`) and variable names (`n`, `divide?`). We also see that it is based on a stack (`exch`, `dup`) and uses a postfix notation (as in (`n 1 eq`) or when pushing the condition, "then" block and "else" block on the stack before calling `ifelse`). This use of stack in a postfix manner does not make PostScript the easiest programming language to read.[3]

Fractals as L-Systems

An L-system [9, 7], named after the biologist Lindenmayer, is a model based on rewriting rules, most famously used to study the self-similarity found in plants [6]. Before going into plants (which require branching, as plants would), consider a (mythical) animal, the Dragon. The Dragon Curve [8] is a famous fractal that is obtained by repeatedly replacing a line segment by a "corner" made of two shorter line segments.

The Dragon Curve can be described as a L-system with three rewriting rules:[4] `X` $\rightarrow$ `-FX++FY-`, `Y` $\rightarrow$ `+FX--FY+` and `F` $\rightarrow \Lambda$ (where Λ represents the empty string to indicate that `F` symbols are removed). Starting with `X`, apply the rules N times, each time replacing *all* the `X`, `Y` and `F`. After N iterations, remove the remaining the `X` and `Y` and what is left is a string of `F`, `-` and `+`. If `F` means "move forward"; `-` means "turn 45° right"; and `+` means "turn 45° left", in the classic "turtle" interpretation, the string describes a curve. Intuitively, the rewriting rules implement a process by which a line segment is being removed (rule `F` $\rightarrow \Lambda$) and replaced by a "corner" below (rule `X`) or above (rule `Y`) it, in an alternating fashion. Fig 4 shows the first seven iterations, with the corresponding curves.

After seven iterations, and using "round corners" to make the curve easier to follow, it looks like this (and yes, this is a Dragon, not a French poodle):



[3]Not that this would scare an experienced LaTeX user, I am sure.

[4]There are simpler L-systems for the Dragon Curve, described in [6], but the system chosen here has the benefit of drawing the Dragon always oriented in the same direction.

$N = 1$

```
-F++F-
```

$N = 2$

```
-F++F-+++F-F+-
```

$N = 3$

```
--F++F-+++F-F+-+++-F++F--+F-F++-
```

$N = 4$

```
--F++F-+++F-F+-+++-F++F--+F-F++-+++-F++F-+++
F-F+--+-F++F--+F-F+++-
```

$N = 5$

```
---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++F-+++F-F+--+-F++F--+F-
F++++-
```

$N = 6$

```
---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++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-+++
F-F+-+++-F++F--+F-F++--+-F++F-+++F-F+--+-F+
+F--+F-F+++++-
```

$N = 7$

```
----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++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-++
+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-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+-+++
-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++F--+F-F++++++-
```

Figure 4: Dragon Curve, as generated by the system `X` $\rightarrow$ `-FX++FY-`, `Y` $\rightarrow$ `+FX-FY+`, `F` $\rightarrow$ Λ.

Programming L-Systems in PostScript

Since PostScript is stack-based, it obviously supports recursive functions. The three rewriting rules of the Dragon Curve system can be implemented as three functions `X`, `Y` and `F`, where `X` and `Y` are mutually recursive. The PostScript program for the Dragon Curve is given below, with relevant comments.

First, the file starts with a special comment to indicate that it should be interpreted as PostScript.

```
%!PS
```

A variable `N` is then defined. This is the number of iterations we want to perform.

```
/N 7 def
```

Three functions `X`, `Y` and `F` are defined. They take as parameter the number of remaining iterations, which is pushed on the stack before each call. The idea is that, at the last iteration, when this counter reaches zero, the symbols (here, the function calls) are interpreted graphically: `F` draws a line segment while `X` and `Y` do nothing. When the counter is non-zero, the functions are interpreted as rewriting rules, triggering calls to more functions.

Function `X` tests the number of remaining iterations to see if it is zero. This is done by duplicating it, then comparing to zero. This way, if the top of the stack is $\langle 3 \rangle$, it becomes $\langle 3, \mathsf{true} \rangle$ and if it is $\langle 0 \rangle$, it becomes $\langle 0, \mathsf{false} \rangle$. If the count is zero, the function does nothing (it has no graphical counterpart). If the count is non-zero, `X` makes 7 function calls to functions `-`, `+`, `F`, `X` and `Y`. Functions `-` and `+` represent 45° rotations and do not need the iteration count. The count is thus pushed 4 times on the stack (for the 4 calls to `F`, `X` and `Y`) after having been decremented by one. After the calls, the function terminates by popping its parameter from the stack.

```
/X {
  dup 0 ne
  {1 sub 4 {dup} repeat - F X + + F Y -}
  if pop
} def
```

Function `Y` is similar to function `X`. Function `F` does nothing when the count is non-zero (corresponding to the $\mathtt{F} \to \Lambda$ rule). When the counter reaches zero, the function performs a graphical operation, namely drawing a horizontal line segment of length 10.

```
/Y {
  dup 0 ne
  {1 sub 4 {dup} repeat + F X - - F Y +}
  if pop
} def
```

```
/F {
  0 eq { 10 0 rlineto } if
} bind def
```

Although the line is always drawn horizontally, the graphics context is rotated by the `-` and `+` functions. Function `-` rotates it clockwise (right turn) and function `+` does it counterclockwise (left turn).

```
/- { -45 rotate } bind def
/+ { 45 rotate } bind def
```

Following are simple settings so lines have rounded tips and they join nicely.

```
1 setlinejoin
1 setlinecap
```

The call to `newpath` starts a new curve. We move to a carefully calculated location and we scale the picture by a factor equal to $\frac{50}{(\sqrt{2})^N}$, where N is the total number of iterations. Since the line segment have a constant length of 10, the Dragon ends up always having the same size, for any number of iterations.

```
newpath
220 180 moveto
50 N { 2 sqrt div } repeat dup scale
```

Finally, a 90° rotation places the page in landscape orientation and an initial call to function `X` is made with N on the top of the stack. The path that is built by the series of calls to `rlineto` is drawn as a line by `stroke` and the page is printed or displayed, depending on where the PostScript code is interpreted.

```
90 rotate
N X
stroke
showpage
```

The full program can be downloaded from the Web [2]. Other well-known fractals, besides the Dragon Curve, can be represented as L-Systems with well-chosen rotation angles and rewriting rules. Many can be drawn with the same PostScript program by only changing a few lines. Fig. 5 shows the classic examples of Hilbert's curve and Koch's snowflake, with their associated L-systems.

Implementing Branching

Dragons—and, for that matter, French poodles—need trees and trees need branches. L-systems include a branching operator, usually represented with square brackets. Basically, '[' marks a point and ']' goes back to it. This allows the system to recursively build a branch (a subtree) before it continues from the trunk (or main branch) it left from. Fig. 6 shows examples of branching L-systems from [6] and their graphical representations. Note how some systems use nested branching. Colors are obtained here by using lighter shades of green

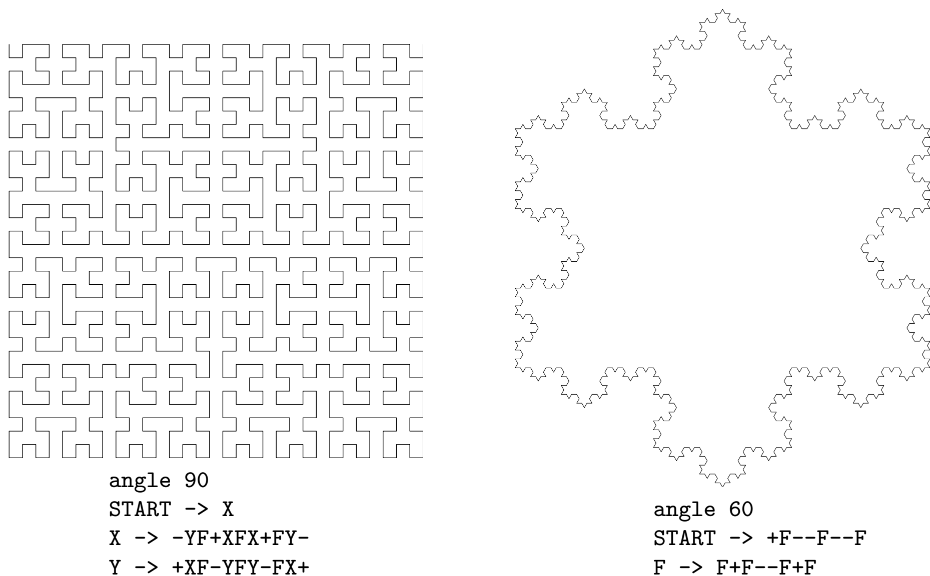


Figure 5: Hilbert's curve and Koch's snowflake.

as the depth of the computation increases for a "realistic" effect, or are chosen randomly at branching points, each variant being straightforward to implement in PostScript (the language has a `rand` operator).

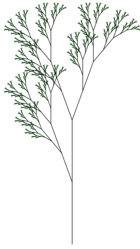
Branching can be implemented straightforwardly using PostScript's `gsave` and `grestore` operators. The first operator saves the current graphics context (including color and current point) by pushing it onto the stack; the second operator restores the graphics context from the stack. They result in the following PostScript implementation of '[' and ']' (square brackets are part of PostScript's syntax for arrays, so `B` and `E` are used instead):

```
/B { gsave } bind def
/E { stroke grestore } bind def
```

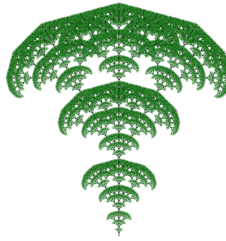
`B` saves the graphics context and `E` draws the current subtree (`stroke`) before restoring the context.

This implementation of branching in PostScript, however, turns out to be quite inefficient: When function `E` is invoked (at the tip of a branch), a path is being stroked all the way from the origin (the root of the tree). As a result, branches shared by many leaves are being drawn many times. This can be avoided by committing the path up to the branching point before branching. An alternate definition of `B` could be:

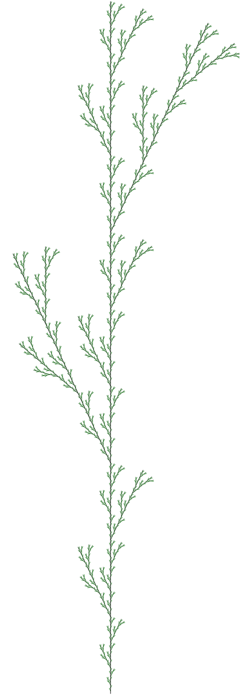
```
/B { currentpoint stroke moveto gsave } bind def
```

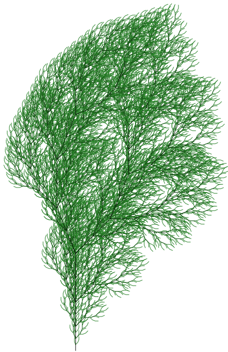


```
angle 20
START -> X
X -> F[+X]F[-X]+X
F -> FF
```



```
angle 30
START -> F
F -> F[+F[+F][-F]F][-F
      [+F][-F]F]F[+F][-F]F
```

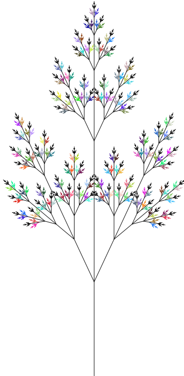




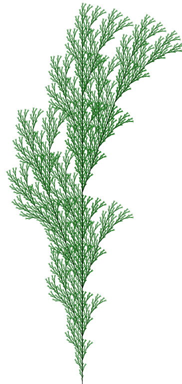
```
angle 22.5
START -> F
F -> FF-[-F+F+F]+[+F-F-F]
```

```
angle 22.5
START -> X
X -> F-[[X]+X]+F[+FX]-X
F -> FF
```

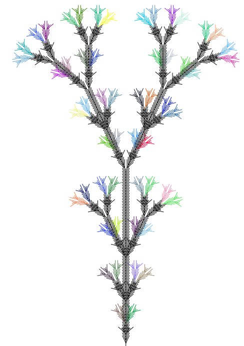
```
angle 25
START -> F
F -> F[+F]F[-F]F
```



```
angle 25
START -> X
X -> F[+X][-X]FX
F -> FF
```



```
angle 20
START -> F
F -> F[+F]F[-F][F]
```



```
angle 25
START -> Y
X -> X[-FFF][+FFF]FX
Y -> YFX[+Y][-Y]
```

Figure 6: Plants as branching L-systems.

Figure 7: Single path versus multiple paths.

Function B now draws the path up to the current point and moves the origin of a new path to the branching point. The resulting program is faster, but branches are drawn as a series of successive strokes, which does not always look as nice. Fig. 7 shows a closeup of a branching point, with the new implementation of branching on the right. To give the user a choice between faster computations or nicer graphics, one can define a Boolean flag within the PostScript program. It could be inefficient, however, to test this flag at every branching point. An alternative approach is to use the flag to *build* variants of the branching function. This strategy can also be used to choose a coloring scheme once and for all, without the need for further testing when the drawing takes place. PostScript was to some extend inspired by Lisp and like Lisp, it offers ways to dynamically build blocks of code to be later evaluated (basically, a block is just an array that is flagged as executable). The resulting implementation of B is as follows:

```
/B [
  fast? {
    {currentpoint stroke moveto}
    aload pop
  } if
  {gsave} aload pop
  currentdict /color known {
    {dup color}
    aload pop
  } if
] cvx bind def
```

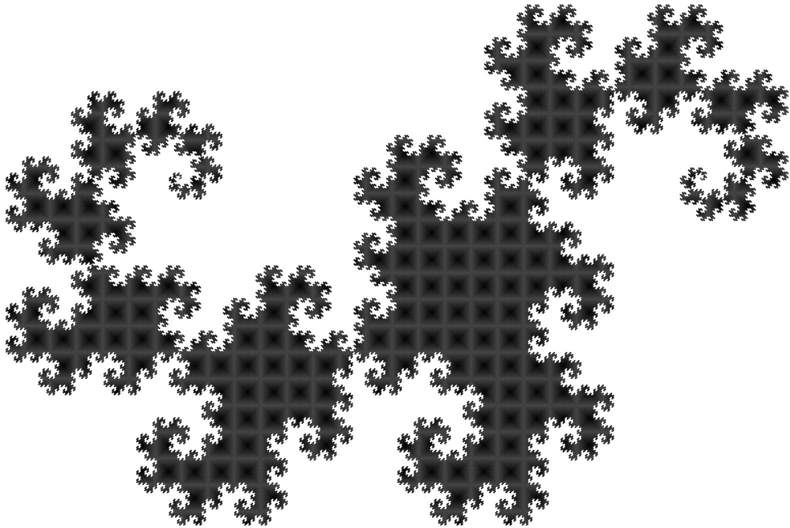
The code tests the Boolean `fast?` and looks up a `color` function to build an array that is then given the executable property by the `cvx` operator. There is no further testing of `fast?` or look-up of `color` when function B is executed

(i.e., when a branching point is reached). If no `color` function is defined, no function call takes place and the drawing remains black. A complete program, with branching and a few different color schemes can be downloaded from the Web [3].

With branching L-systems and imaginative coloring schemes, short PostScript programs can be written that produce impressive looking plants. One thing the examples from fig. 6 show is that L-systems that are almost identical can result in very different looking plants. One can therefore study the effects of mutation-like variations to an L-system to evaluate what happens to the corresponding fractal (see [5] for a system that implements such random mutations).

Conclusions

There is a lot more to L-systems, including fancy operators not described here, generalizations to 3D graphics, and theoretical studies of their expressive power. Sadly, though, PostScript is being superseded by the *Portable Document Format* which, in spite of its qualities as a document exchange format, lacks the programming capabilities that make PostScript so remarkable. The Web is full of Java applets that implement L-systems, but it is just not the same thing. When PostScript is gone, how will we use all those CPU cycles wasted in printers?



References

[1] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley, third edition, February 1999.

[2] Michel Charpentier. Dragon Curve in PostScript. `http://www.cs.unh.edu/~charpov/Programming/L-systems/simple-dragon.ps`.

[3] Michel Charpentier. L-systems in PostScript. `http://www.cs.unh.edu/~charpov/Programming/L-systems/plant2.ps`.

[4] Michel Charpentier. Ulam's Spiral in PostScript. `http://www.cs.unh.edu/~charpov/Programming/PostScript-primes/primes-distribution.ps`.

[5] Jim Lund. DoodleTron (a L-system Iterator). `http://elegans.uky.edu/jiml/lsystem/ls_index.html`.

[6] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[7] Pavel Tišnovský. L-systémy: přírodní objekty i umělé artefakty. `http://www.root.cz/clanky/l-systemy-prirodni-objekty-i-umele-artefakty`.

[8] Eric W. Weisstein. Dragon Curve. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/PrimeSpiral.html`.

[9] Eric W. Weisstein. Lindenmayer Systems. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/LindenmayerSystem.html`.

[10] Eric W. Weisstein. Ulam's Spiral. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/PrimeSpiral.html`.

Summary: Programming L-Systems in PostScript

Although we tend to think of PostScript as a file format used to describe graphics, it is in reality a full-fledged programming language with graphical capabilities. Thus, the power of PostScript goes far beyond that of simple vector-graphics formats. All sorts of computations can be programmed, including complex arithmetic calculations. In this paper, we show how to use recursive functions in PostScript to implement a family of rewriting structures known as L-systems. Based on these systems, one can write short PostScript programs that draw classic fractals and beautiful plant-like pictures.

Keywords: PostScript, (Lindenmayer) L-systems, fractals, unconventional programming languages.

Michel Charpentier, `charpov@cs.unh.edu`
University of New Hampshire, Computer Science Department
Kingsbury Hall, rm N215A, Durham, NH 03824
The United States of America