# L-systems, Twining Plants, Lisp:

Modeling of Twining Plants using Environmentally Sensitive L-systems, and an Extensible Framework for L-systems in Common Lisp.

Cand. Scient. thesis.

Knut Arild Erstad
Department of Informatics, University of Bergen


January 10, 2002

**Abstract**

Lindenmayer systems (L-systems) is a formalism for parallel rewriting, used for generating fractals and for the modeling and simulation of plants. This thesis presents L-systems and many of its extensions with focus on graphical and biological applications and mechanisms for incorporating L-systems into programming languages.

A twining plant model serves as an advanced example of plant simulation. The model includes elongation, rotation and curvature of the stem, attachment to surfaces, and leaves that follow the flow of growth.

L-Lisp is a framework for L-systems in Common Lisp, created by the author. The framework demonstrates how extensible programming languages like Lisp can incorporate the concept of L-systems in an elegant manner. Working with L-systems within a programming language gives great flexibility, but for less extensible languages than Lisp it would involve much detailed bookkeeping, leading to large and overly complicated definitions. By comparison, L-Lisp provides the same flexibility, yet the resulting L-system definitions are short and concise.

# Preface

This thesis is part of my Cand. Scient. (Candidatus Scientiarum) degree at the University of Bergen, Department of Informatics.

The thesis consists of the following parts:

**Chapter 1** gives a detailed introduction to L-systems and their extensions, including many examples. It is a summary of the formalisms described in the inspiring book *The Algorithmic Beauty of Plants* [17] and later research [19, 20] by Przemyslaw Prusinkiewicz et al. The inclusion of extensions such as homomorphism, decomposition, spline functions and programming statements is a direct result of visiting Prusinkiewicz at the University of Calgary during the autumn of 1999.

**Chapter 2** introduces environmentally sensitive L-systems. This is mostly excerpts from the work of Prusinkiewicz, Radomír Měch et al [19, 14]. The section on tropism shows a slightly different approach to the subject than most other texts.

**Chapter 3** is a description of an L-system based twining plant model, suggested by Prusinkiewicz and based upon earlier work by Měch, among others. The model was developed by the author in collaboration with Prusinkiewicz.

**Chapter 4** is a description of L-Lisp, an experimental framework for L-systems, created entirely by the author, although many of the features are inspired by the *cpfg* language [20]. All L-system extensions discussed in the thesis have been implemented and tested in L-Lisp. L-Lisp is a result of my fascination for extensible programming languages, in particular Common Lisp. The motivation was twofold; I wanted to learn Common Lisp, and see how well L-systems could be expressed in the language. One can safely say that Common Lisp met and exceeded my expectations. All fractal and plant images in this thesis were created by L-Lisp, by generating either Postscript graphics or 3D data for the Povray ray-tracer.

**Appendix A** contains partial source code for L-Lisp.

## Acknowledgements

First, a general thank-you to everyone who waited patiently for me to finish this thesis during far too many delays and obstacles.

Many thanks to Dr. Przemyslaw Prusinkiewicz, his colleagues and students at the University of Calgary, for a warm welcome and invaluable help during my stay there. Without it, this thesis would undoubtedly have been much shallower and less up-to-date.

Thanks to my supervisor Jan Arne Telle for giving me the freedom to write about what I wanted, even a subject matter which is not taught at the University of Bergen, for his great help on thesis writing in general, and for getting me in contact with Prusinkiewicz.

I would also like to thank the free software community for providing students and other young programmers with great development tools. Special thanks go to the Common Lisp community, in particular to the creators of CMU Common Lisp for one of the best free programming environments available, and to Richard Mann for creating OpenGL bindings for Allegro Common Lisp which I later ported to CMUCL for use in L-Lisp.

Thanks to Bergen, for plentiful water in numerous forms.

Finally, I want to thank my family, and especially my parents, for all their support.

# Contents

# List of Figures

# Chapter 1

# Introduction to L-systems

## 1.1 Lindenmayer systems

L-systems, named after Aristid Lindenmayer, were introduced as a a mathematical model for cellular plant growth[12]. A simple L-system consists of a starting string, usually called an *axiom* or *initiator*, and a *grammar* similar to those used for defining programming languages.

In normal (Chomsky) grammars the productions are applied for one symbol at a time, but in L-systems they are applied in parallel for *all* symbols in the string. Another difference is that in Chomsky grammars the alphabet is partitioned into terminal and nonterminal symbols. In L-systems there is no such distinction, and termination of the rewriting process is usually done after a given number of steps.

Good introductions to L-systems, with focus on graphical and biological applications, are given in [17, 19].

## 1.2 DOL-systems

The simplest form of L-systems are deterministic and context-free, and are called *DOL-systems*. A DOL-system consists of an *alphabet*, a start string called the *axiom* $\omega$ and exactly one *production* for each alphabet symbol. Usually we do not specify the alphabet explicitly.

For instance, a simple DOL-system could be given as:

$$\begin{array}{ccc} \omega & : & a \\ a & \to & b \\ b & \to & ab \end{array}$$

The alphabet is implicitly $\{a, b\}$. To find the strings derived from this DOL-system, we start with the axiom $a$, which is the first string. Then we apply the production $a \to b$ to get $b$, which is the second string. We apply the production $b \to ab$ to get $ab$, which is the third string. Now we apply both productions in parallel to get $bab$, which is the fourth string, and so on:

$$
\begin{array}{c}
a \\
\Downarrow \\
b \\
\Downarrow \\
ab \\
\Downarrow \\
bab \\
\Downarrow \\
abbab \\
\Downarrow \\
bababbab \\
\Downarrow \\
abbabbababbab \\
\cdots
\end{array}
$$

The strings that we can generate in this matter are said to be in the L-system's language.

In this example, the length of successive strings equal the Fibonacci sequence, and this DOL-system is sometimes referred to as the *Fibonacci L-system*.

## 1.3 Turtle graphics

This section shows how we can produce images from strings by simulating a "turtle" that accepts some basic commands, and how we can combine this with DOL-systems to produce some simple fractals.

Our turtle has two states: position and direction, and it accepts simple commands for turning, moving and drawing. The most basic commands are:

- `F` Move forward one unit, drawing a line.
- `f` Move forward one unit without drawing a line.
- `+` Turn left by a fixed angle $\delta$.
- `-` Turn right by a fixed angle $\delta$.

A *turtle interpretation* of a string is the image we produce by going through the string, left to right, and "feeding" the symbols as commands to the turtle.

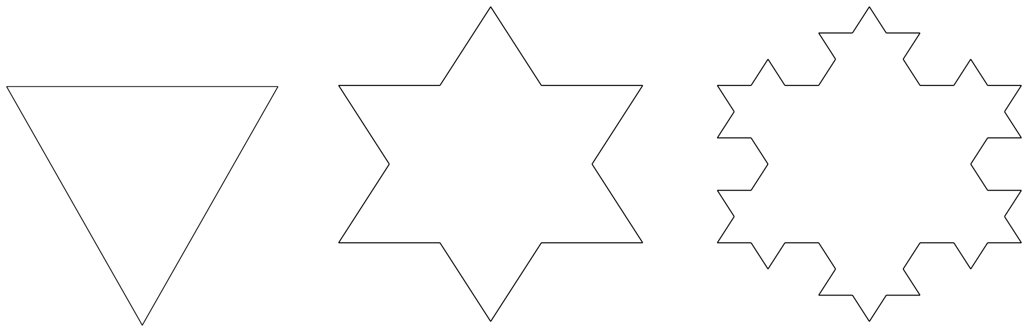
For instance, if $\delta = 90°$, then the turtle interpretation of the string `F+F+F+F` will be a square.

As shown by Prusinkiewicz [16], turtle interpretation of L-system strings can be used to generate pictures. Figure 1.1 shows the three first steps of a classic fractal, the *snowflake curve*, generated by an L-system. [1]

[1] Note that the productions `+` $\to$ `+` and `-` $\to$ `-` are implicit in this L-system. A production $a \to a$ for any symbol $a$ is

Figure 1.1: Snowflake curve

$$
\begin{array}{lcl}
\omega & : & \texttt{F--F--F} \\
\delta & : & 60^\circ \\
\texttt{F} & \rightarrow & \texttt{F+F--F+F}
\end{array}
$$



```
F--F--F
```

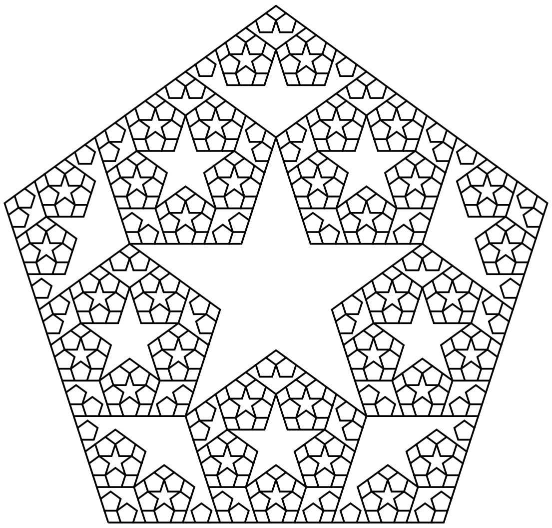
```
F+F--F+F--F+F-
-F+F--F+F--F+F
```

```
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 1.2 shows a somewhat occult-looking fractal and the L-system that generated it.

Figure 1.2: Occult L-system

| | | |
|---|---|---|
| $\omega$ | : | `F+F+F+F+F` |
| $\delta$ | : | $72°$ |
| `F` | $\rightarrow$ | `FF+F+F+F+F+FF` |



### 1.3.1 3D turtle

For a 2D turtle we only needed an angle to represent the heading. For a turtle that can move in three dimensions we need something more complex. To represent its *orientation* we use three vectors $\vec{H}, \vec{L}$ and $\vec{U}$, representing the turtle's *heading* (forward direction), *left* direction and *up* direction. They each have unit length and are perpendicular to each other, so strictly speaking we only need two of them. If we only had $\vec{H}$ and $\vec{L}$ we could calculate $\vec{U} = \vec{H} \times \vec{L}$.

We can rotate the turtle by the equation

$$[\vec{H}' \quad \vec{L}' \quad \vec{U}'] = [\vec{H} \quad \vec{L} \quad \vec{U}]\mathbf{R}$$

where $\mathbf{R}$ is a rotation matrix. To rotate by an angle $\alpha$ around $\vec{H}, \vec{L}$ or $\vec{U}$ we use standard rotation matrices:

called an *identity production* and is usually not listed.

$$\mathbf{R_H}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

$$\mathbf{R_L}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$\mathbf{R_U}(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A 3D turtle understands the following rotation symbols:

- `+` Turn left by angle $\delta$, using matrix $\mathbf{R_U}(\delta)$.
- `-` Turn right by angle $\delta$, using matrix $\mathbf{R_U}(-\delta)$.
- `&` Pitch down by angle $\delta$, using matrix $\mathbf{R_L}(\delta)$.
- `^` Pitch up by angle $\delta$, using matrix $\mathbf{R_L}(-\delta)$.
- `\` Roll left by angle $\delta$, using matrix $\mathbf{R_H}(\delta)$.
- `/` Roll right by angle $\delta$, using matrix $\mathbf{R_H}(-\delta)$.

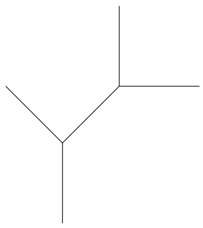
## 1.4 Bracketed L-systems

If we want to represent trees or other branching structures using turtle interpretation, we face a problem; when the turtle is finished with one branch and should start on the next, how does it get back to the branching point? We introduce *bracketed L-systems* to solve this problem.

The two bracket symbols have special meanings for the turtle:

- `[` Push the current turtle state onto a stack (start a branch).
- `]` Pop the turtle stack, restoring an earlier state (finish a branch).

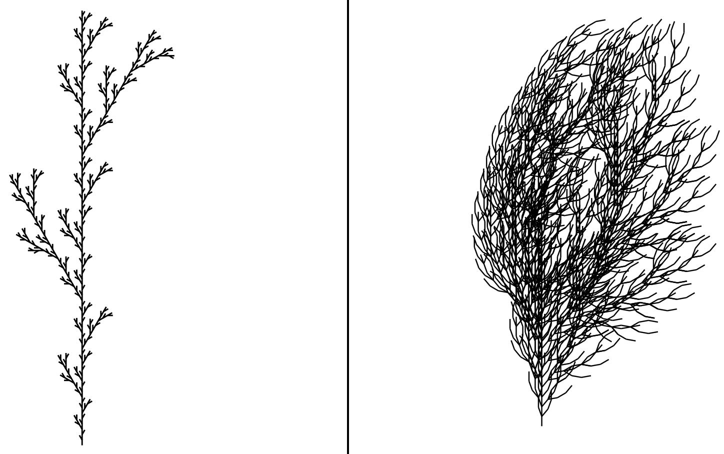
For instance, if $\delta = 45°$, then the string `F[+F]-F[+F]-F` will be interpreted as:



We can use brackets in L-system productions to produce tree-like structures. To avoid errors, the brackets in each production should be well-balanced. Figure 1.3 shows some examples of tree-like fractals generated by bracketed L-systems.

Figure 1.3: Fractal tree structures

| | | | | | |
|---|---|---|---|---|---|
| $\omega$ | : | `F` | $\omega$ | : | `F` |
| $\delta$ | : | $30°$ | $\delta$ | : | $20°$ |
| Steps | : | 5 | Steps | : | 5 |
| `F` | $\rightarrow$ | `F[+F]F[-F]F` | `F` | $\rightarrow$ | `F-[-F+F+F]+[+F-F-F]F` |



## 1.5 Parametric DOL-systems

In *parametric L-systems* each symbol can have numerical values called *parameters*. If a symbol $A$ has one parameter 5, this is simply written $A(5)$. If there are several parameters, they are written as a comma-separated list, for instance $A(1, 2.3, -4.5)$. A symbol with parameters is called a *module*.

A parametric production specifies how the right-hand parameters depend on the left-hand parameters. The left-hand side gives the names of the parameter, the right-hand parameters are given as expressions. As an example, consider the following L-system:

$$\begin{array}{lcl} \omega & : & A(1) \\ A(x) & \rightarrow & A(x*2)B(x) \\ B(x) & \rightarrow & B(x-1) \end{array}$$

The first six generated strings are:

$$
\begin{gathered}
A(1)\\
\Downarrow\\
A(2)B(1)\\
\Downarrow\\
A(4)B(2)B(0)\\
\Downarrow\\
A(8)B(4)B(1)B(-1)\\
\Downarrow\\
A(16)B(8)B(3)B(0)B(-2)\\
\Downarrow\\
A(32)B(16)B(7)B(2)B(-1)B(-3)
\end{gathered}
$$

The expressions for the right-hand parameters can use arithmetic operators like $+, -, *, /$, normal numerical functions like sin or log (more on this in section 1.10.1), and parenthesis. We can also add conditionals to get productions of the form $L(x) : \mathit{test} \rightarrow R(x)$, where the *test* expression in addition to the arithmetic operators can contain relational operators like &, $|$, ! (and, or, not) and the production is applied if and only if the test returns true.

For instance, consider the following parametric L-system:

$$
\begin{array}{lclcl}
\omega & : & A(1) & & \\
A(x) & : & x \leq 3 & \rightarrow & A(x+1)B(0,x) \\
A(x) & : & x > 3 & \rightarrow & A(x+1) \\
B(x,y) & : & y < 2 & \rightarrow & C \\
B(x,y) & : & y \geq 2 & \rightarrow & B(x+1,y+1)
\end{array}
$$

The first six strings will be:

$$
\begin{gathered}
A(1)\\
\Downarrow\\
A(2)B(0,1)\\
\Downarrow\\
A(3)B(0,2)C\\
\Downarrow\\
A(4)B(0,3)B(1,3)C\\
\Downarrow\\
A(5)B(1,4)B(2,4)C\\
\Downarrow\\
A(6)B(2,5)B(3,5)C
\end{gathered}
$$

### 1.5.1 Parametric turtle interpretation

When parameters are attached to symbols that the turtle understands, it interprets this as a distance or an angle, for instance:

| | |
|---|---|
| `F(x)` | Move forward a step of length `x` and draw a line. |
| `f(x)` | Move forward a step of length `x` without drawing. |
| `+(x)` | Turn left by an angle `x` (or right if `x` is negative). |
| `-(x)` | Turn right by an angle `x`. |
| `^(x)` | Pitch up by an angle `x`. |
| `&(x)` | Pitch down by an angle `x`. |
| `\(x)` | Roll left by an angle `x`. |
| `/(x)` | Roll right by an angle `x`. |
| `!(x)` | Set line width to `x`. |
| `;(x)` | Set color index to `x`. |

Revisiting the snowflake curve L-system in figure 1.1, notice that the curve in reality grows larger for each iteration (the images are simply scaled down in this figure). Using parameters we can construct a snowflake curve that does not grow in size[2] for each iteration:

```
ω     :   F(1)-(120)F(1)-(120)F(1)
F(x)  →   F(x/3)+(60)F(x/3)-(120)F(x/3)+(60)F(x/3)
```

The ability to work with real coordinates instead of an exponentially growing curve is just one of many possibilities, as will be seen in subsequent sections.

Figure 1.4 is a 3D tree generated by a parametric DOL-system, and it uses several of the parametric turtle commands, including "`!(x)`", which sets the line width. Also, note that all turning commands have parameters, and hence a default turning angle would have no effect and is omitted.

## 1.6 Context-sensitive L-systems

When applying a production in a *context-sensitive L-system*, we also consider modules to the left or right of the module to be replaced.

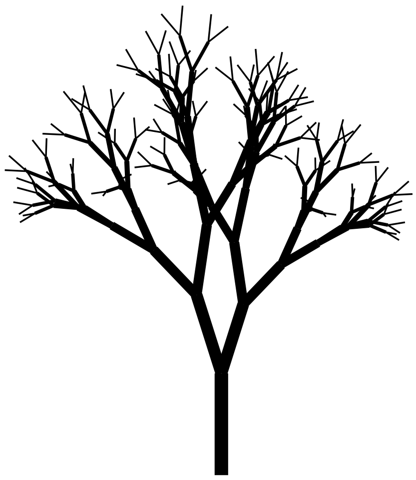
For a *2L-system* we consider the successor and predecessor, and a simple non-parametric production is written in the form $L < A > R \rightarrow X$, which means that $A$ is replaced by $X$ iff $A$ is preceded by $L$ and followed by $R$.

In a *1L-system* we consider the left or right symbol, but not both. Productions can be of the form $L < A \rightarrow X$ or $A > R \rightarrow X$.

[2] At least not exponentially like the original example.

Figure 1.4: Parametric DOL-system tree

| | | |
|---|---|---|
| Depth | : | 6 |
| $\omega$ | : | $!(1)F(1)A(1)$ |
| $!(x)$ | $\rightarrow$ | $!(x+1)$ |
| $A(s)$ | $\rightarrow$ | $[/(137)\&(25)!(1)F(0.85s)A(0.85s)]$ |
| | | $\backslash(43)\&(25)!(1)F(s*0.75)A(0.75)$ |



Generally we have *(k,l)-systems* where we look at the $k$ preceding and $l$ succeeding symbols. They are also called *IL-systems*.

Since 1L- or 2L-systems just look at the immediate surroundings of each symbol or module, this could be considered the most realistic model when simulating low-level biological growth, such as cell growth. We will however use a context-sensitive model which is more general and a bit more loosely defined:

- Any number of symbols are allowed as left or right context.
- If both a context-sensitive and a "regular" production exists for the same symbol, then the context-sensitive one should be considered first.
- The productions should be listed in the order that they are considered. In other words, the more specific productions should be listed before the more general ones, the context-sensitive before the context-free.
- Certain alphabet symbols can be ignored when checking the context. This is listed in the L-system definition as "Ignore: *list of symbols*" or "Consider: *list of symbols*". The latter ignores every symbol *except* the ones listed.

For example, consider the following context-sensitive L-system:

$$\begin{array}{lcl} \omega & : & ABCACBADB \\ Ignore & : & C \\ A > B & \rightarrow & X \\ A & \rightarrow & Y \end{array}$$

Each time the symbol $A$ occurs, the right context is examined. If it matches $B$, the $A$ is replaced by $X$, otherwise the more general production is used, and $A$ is replaced by $Y$.

During the first rewrite of the axiom $ABCACBADB$, the first $A$ will become $X$, the second also $X$ (because the $C$ is ignored when context-checking), and the third $Y$. The rewrite string will become $XBCXCBYDB$.

### 1.6.1 Parameters and context-sensitivity

The parametric and context-sensitive case is a pretty straightforward combination of the two sub-cases. You can use parameters from the context in the expressions, as shown in this production example:

$$L(x_1)L(x_2) < A(y) > R(z) : x_1 + x_2 < 1 \rightarrow S(y)T(x_1 + x_2 + y + z)$$

### 1.6.2 Brackets and context-sensitivity

For bracketed L-systems context-sensitivity is a bit more complex. The natural way to think of the left context of a branch is the "parent" section of the branching structure. This can be done by skipping over a (possibly nested) bracketed section. Similarly, the right context of a branch might consist of the closest segments of several outgoing branches.

For instance, the production

$$L_1L_2 < A > R_1[R_2]R_3 \rightarrow X$$

will match the $A$ in the string

$$L_1[L_2[XY]AR_1[R_2XY]R_3X]Y$$

Brackets should not be used in the production's left context, this would not make sense since there is only one path towards the root of any tree. The brackets of the production's right context should be well-balanced, and the symbols within brackets are matched against the beginning of the string branches.

### 1.6.3 Signals

Context-sensitive productions can be used for sending signals through the rewriting string. In the following example the module $S$ is a signal that starts as the leftmost module and is moved one position right for each rewrite:

$$\begin{array}{lcl} \omega & : & Saaaaaaaa \\ S & \rightarrow & \epsilon \\ S < a & \rightarrow & aS \\ a & \rightarrow & a \end{array}$$

At each step, the $S$ is erased from its current position ($S \rightarrow \epsilon$) and a new $S$ is inserted to the right of this position ($S < a \rightarrow aS$). The resulting rewriting strings will be:

$$\begin{gathered} Saaaaaaaa \\ \Downarrow \\ aSaaaaaaa \\ \Downarrow \\ aaSaaaaaa \\ \dots \end{gathered}$$

In this case, the signal was a module, but it could also have been a parameter to a module, as in the following L-system:

$$\begin{array}{lcl} \omega & : & a(1)a(0)a(0)a(0)a(0)a(0) \\ a(y) < a(x) : y = 1 & \rightarrow & a(1) \\ a(x) & \rightarrow & a(0) \end{array}$$

which produces the string sequence

$$\begin{gathered} a(1)a(0)a(0)a(0)a(0)a(0) \\ \Downarrow \\ a(0)a(1)a(0)a(0)a(0)a(0) \\ \Downarrow \\ a(0)a(0)a(1)a(0)a(0)a(0) \\ \dots \end{gathered}$$

For plant simulations, signals can be propagated towards the root (*basipetal* signals) or towards the apices[3] (*acropetal* signals). Examples of these are given in figures 1.5 and 1.6. In the examples it is assumed that the symbols $F_a$ and $F_b$ both work like the turtle command $F$, and $F_a$ is shown as thicker lines than $F_b$.[4]

## 1.7 Stochastic L-systems

All the L-systems described so far have been deterministic. If we start the rewriting process for a given L-system, it will always produce the same sequence of strings. Strictly speaking, we *could* create nondeterministic L-systems by allowing random functions to be used in parametric L-systems, and this has its uses. For instance, the production `F(x)` $\rightarrow$ `F(x+ran(1.0))` can represent a line that grows in a random manner (assuming that `ran(x)` returns a random number between 0 and `x`). We could even use random functions in test expressions to choose randomly between productions:

$$\begin{array}{lcl} \texttt{F} : ran(1) < 0.5 & \rightarrow & \texttt{F} \\ \texttt{F} : ran(1) < 0.5 & \rightarrow & \texttt{F[-F]} \\ \texttt{F} & \rightarrow & \texttt{F[+F]} \end{array}$$

[3] *Apices* is plural for *apex* (tip).

[4] This can be achieved by using an L-system with *homomorphism*, see section 1.8, page 13.

Figure 1.5: Acropetal signal (towards apices)

$$
\begin{array}{lcl}
\omega & : & F_a[+F_b]F_b[-F_b]F_b[+F_b]F_b \\
\text{Ignore} & : & +- \\
F_a < F_b & \rightarrow & F_a
\end{array}
$$

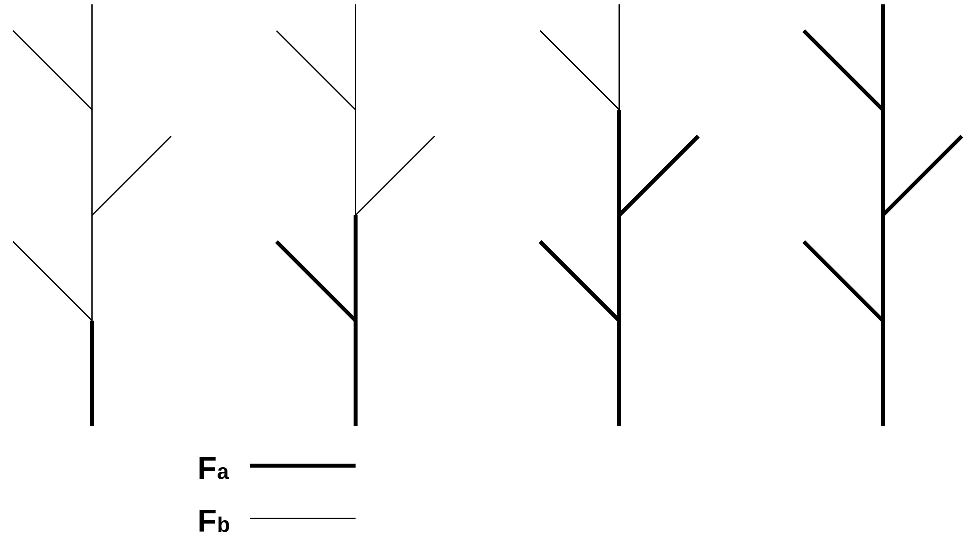
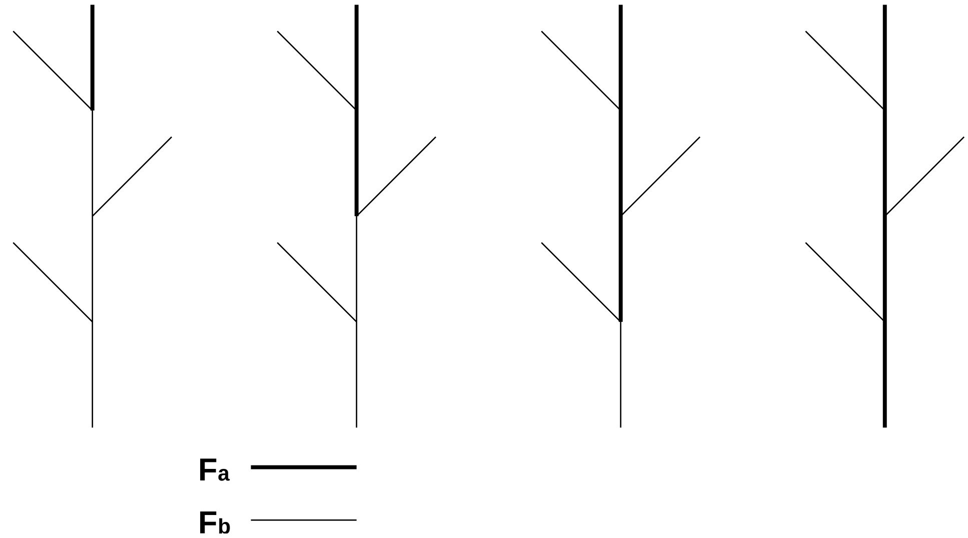



Figure 1.6: Basipetal signal (towards root)

$$
\begin{array}{lcl}
\omega & : & F_b[+F_b]F_b[-F_b]F_b[+F_b]F_a \\
\text{Ignore} & : & +- \\
F_b > F_a & \rightarrow & F_a
\end{array}
$$



In this case, whenever the symbol `F` is found, there is a 50% chance that the first production will be applied. If it is not applied, then there is a 50% chance between the second and third production. This means that the random distribution between the 3 productions is 50%, 25%, 25%, but this is not very obvious.

Using a *stochastic L-system* we can express the same set of productions in a more compact and intuitive way:

$$\begin{array}{lcllc} \texttt{F} & \rightarrow & \texttt{F} & : & 2 \\ \texttt{F} & \rightarrow & \texttt{F[-F]} & : & 1 \\ \texttt{F} & \rightarrow & \texttt{F[+F]} & : & 1 \end{array}$$

The numbers behind the productions are the "random weights" and show the random distribution to be $2:1:1$. The chance for one of the productions to be applied can be calculated by dividing its weight by the sum of all the weights for productions with equal left-hand side. The chance that the first production will applied is $2/(2+1+1) = \frac{1}{2} = 50\%$, the chance for either of the other productions is $1/(2+1+1) = \frac{1}{4} = 25\%$.

The random weights can be expressions, which can create a different random distribution depending on the left-hand parameters, like this:

$$\begin{array}{lcllc} A(x) & \rightarrow & A(x) & : & x \\ A(x) & \rightarrow & A(x+1) & : & max(1-x,0) \end{array}$$

All weights must be $\geq 0$.

Figure 1.7 shows an example of a stochastic L-system and three trees generated by it.

## 1.8 L-systems with homomorphism

When developing an L-system model, it can be useful to concentrate on the overall structure and the visual details separately. One way of achieving this, which turns out to be very useful, is to introduce a second set of productions called *homomorphism* productions.

In formal language theory, a homomorphism is a one-to-one mapping between two alphabets respecting certain operations. The L-system meaning is similar, but not quite the same. The homomorphism productions provide a mapping from high-level to low-level (visual) modules. For instance, we can represent a branch segment by $B(s,w)$, where $s$ is the length of the segment and $w$ is the thickness. We can then use a homomorphism production $B(s,w) \rightarrow !(w)F(s)$ to produce a simple visualization.

The following rules apply to homomorphism productions:

- The productions should be context-free.
- The productions are applied after each L-system iteration, but only if an image is to be generated during this iteration.

Figure 1.7: Stochastic L-system trees

| | | | |
|---|---|---|---|
| $\omega$ | : | `!(1)FA` | |
| Steps | : | 10 | |
| `!(n)` | $\rightarrow$ | `!(n+1)` | |
| `A` | $\rightarrow$ | `/(137)[+(20)!(1)FA]-(20)!(1)FA` | : 1 |
| `A` | $\rightarrow$ | `/(137)-(20)!(1)FA` | : 1 |

- Productions are applied recursively until they no longer match any symbol/module in the string.
- After image generation, the string is discarded. In other words, the changes from homomorphism productions do not stay in the rewrite string.[5]

Figure 1.8 should help clarify the rewriting scheme for L-systems with homomorphism.

As an example, consider the L-system in figure 1.5, page 12. The commands $F_a$ and $F_b$ should work like the turtle command $F$, except that $F_a$ draws a somewhat thicker line. With homomorphism, this is easy to accomplish; just use the homomorphism productions $F_a \rightarrow !(2)F$ and $F_b \rightarrow !(1)F$. Moreover, there is no technical reason to give the modules names which look like $F$. If we name them $A$ and $B$, the full L-system could be written as:

$$
\begin{array}{lcl}
\omega & : & A[+B]A[-B]B[+B]B \\
\text{Ignore} & : & +- \\
 & & \\
A < B & \rightarrow & A \\
 & & \\
\text{homomorphism} & & \\
A & \rightarrow & !(2)F \\
B & \rightarrow & !(1)F
\end{array}
$$

[5] When we get to environmentally sensitive L-systems this is a bit more complex (see section 2.1)

Figure 1.8: Homomorphism rewriting scheme

| | | |
|---|---|---|
| $s_i$ | : | $i$th rewriting string before homomorphism |
| $v_i$ | : | $i$th rewriting string after homomorphism |
| $f_i$ | : | $i$th frame (image) |
| $\Downarrow_H$ | : | Homomorphism productions |
| $\Downarrow_T$ | : | Turtle interpretation |
| $\Rightarrow^P$ | : | L-system rewrite |

$$\begin{array}{ccccccc}
s_1 & \Rightarrow^P & s_2 & \Rightarrow^P & s_3 & \Rightarrow^P & \ldots \\
\Downarrow_H & & \Downarrow_H & & \Downarrow_H & & \\
v_1 & & v_2 & & v_3 & & \\
\Downarrow_T & & \Downarrow_T & & \Downarrow_T & & \\
f_1 & & f_2 & & f_3 & &
\end{array}$$

In this example, there is one normal L-system production and two homomorphism productions. The syntax is the same for both kinds of productions, but those listed after the "homomorphism" keyword are homomorphism productions. The "homomorphism" keyword is always on a separate line.

It is perfectly possible to define an L-system using no normal productions, only homomorphism productions, even if the term "L-system" is a bit misleading in that case. After all, there is then no parallel rewriting going on, so essentially we have a Chomsky grammar instead of an L-system.

## 1.9 L-systems with decomposition

When L-systems are used to simulate growth, each rewrite is normally meant to represent one fixed time step. If, for instance, we want to simulate a simple one-dimensional chain of "cells" that grow and split, we could make some simple rules:

- Cells expand with a constant factor $C$ for each time step.
- When a cell becomes longer than a constant $L$, it splits into two equally sized cells.

If $A(s)$ represents a cell of length $s$, then the growth could be simulated by the L-system production $A(s) \to A(s * C)$. We also need to specify the splitting, however, and one way to achieve this is to use the following productions:

$$\begin{array}{lcccl}
A(s) & : & s * C > L & \to & A(s * C/2)A(s * C/2) \\
A(s) & & & \to & A(s * C)
\end{array}$$

This means that a cell grows from $s$ to $s * C$ in one time step, but if $s * C > L$, then the cell must have split sometime between the two time steps, and the first production makes sure that happens.

If $C = 1.5$, $L = 1$, and the axiom $\omega = A(0.5)$, then we will get the following sequence of strings:

$$
\begin{gathered}
A(0.5)\\
\Downarrow\\
A(0.75)\\
\Downarrow\\
A(0.5625)A(0.5625)\\
\Downarrow\\
A(0.84375)A(0.84375)\\
\Downarrow\\
A(0.6328125)A(0.6328125)A(0.6328125)A(0.6328125)\\
\Downarrow\\
\ldots
\end{gathered}
$$

We can get smaller or bigger time steps by choosing different values for $C$.[6] If we choose $C = 1.001$ we will get very small time steps. We can in fact get as small steps as we want by choosing $C$ close enough to 1.

However, we cannot get as *big* time steps as we wish. If we choose a $C > 2$, then the cells will grow faster than they split, and the simulation fails. An elegant solution to this problem is to introduce a second set of productions called *decomposition* productions.

With decomposition, the cell growth L-system could be like this:

$$
\begin{array}{llll}
\omega & : & A(0.5) & \\
A(s) & & & \rightarrow \quad A(s * C) \\
 & & & \\
\text{decomposition} & & & \\
A(s) & : & s > L & \rightarrow \quad A(s/2)A(s/2)
\end{array}
$$

The first production is a normal L-system production that grows each cell, then a "decomposition" keyword follows, which means that any following productions are decomposition productions. The second production is a decomposition production defining that each cell larger than $L$ will be split into two equal parts.

Using the same parameters as above ($C = 1.5$ and $L = 1$) we get the following strings:

[6] Strictly speaking, we are changing the growth rate, not the time steps, but since there is nothing to compare the cell growth with, there is no practical difference between the two. If we could choose a "smallest possible time step", then we could get multiples $M$ of that time step by using the expression $A(s * C^M)$.

$$
\begin{array}{rc}
s_1: & A(0.5) \\
 & \Downarrow \\
d_1: & A(0.5) \\
 & \Downarrow \\
s_2: & A(0.75) \\
 & \Downarrow \\
d_2: & A(0.75) \\
 & \Downarrow \\
s_3: & A(1.125) \\
 & \Downarrow \\
d_3: & A(0.5625)A(0.5625) \\
 & \Downarrow \\
s_4: & A(0.84375)A(0.84375) \\
 & \Downarrow \\
d_4: & A(0.84375)A(0.84375) \\
 & \Downarrow \\
 & \ldots
\end{array}
$$

where the $i$th rewriting string is labelled $s_i$ before decomposition and $d_i$ after decomposition. In the above example, the decomposition production was applied only once, between $s_3$ and $d_3$. If we choose a larger $C$ constant (to make the cells grow faster), it can be applied during every decomposition step, even several times. For instance, if $C = 5.0$:

$$
\begin{array}{rc}
s_1: & A(0.5) \\
 & \Downarrow \\
d_1: & A(0.5) \\
 & \Downarrow \\
s_2: & A(2.5) \\
 & \Downarrow \\
d_2: & A(0.625)A(0.625)A(0.625)A(0.625) \\
 & \Downarrow \\
s_3: & A(3.125)A(3.125)A(3.125)A(3.125) \\
 & \Downarrow \\
 & \ldots
\end{array}
$$

The $s_2$ string is split into $A(1.25)A(1.25)$, but each of those cells are still larger than $L$, so the productions are applied again and we get $A(0.625)A(0.625)A(0.625)A(0.625)$.

The following rules apply to decomposition:

- The productions should be context-free.

- The productions are applied after each L-system iteration, but before a possible homomorphism.
- The productions are applied recursively (like homomorphism).
- The decomposition changes stay in the rewriting string (unlike homomorphism).

### 1.9.1 Combining homomorphism and decomposition

L-systems with decomposition and homomorphism are very useful when simulating plant growth. Often we will use the three sets of productions like this:

**Normal productions:** specify how the different parts of the plant grow.

**Decomposition:** specify how the parts of the plant split into smaller parts.

**Homomorphism:** specify how the plant parts should look. This will normally be simple early in development and more detailed later.

Figure 1.9 shows the rewriting process for an L-system with decomposition and homomorphism. The scheme is essentially a hybrid between L-systems and Chomsky grammars.

Figure 1.9: Homomorphism and decomposition rewriting scheme

| | | |
|---|---|---|
| $s_i$ | : | $i$th rewriting string before decomposition |
| $d_i$ | : | $i$th rewriting string after decomposition, but before homomorphism |
| $v_i$ | : | $i$th rewriting string after homomorphism |
| $f_i$ | : | $i$th frame (image) |
| $\Downarrow_H$ | : | Homomorphism productions |
| $\Downarrow_T$ | : | Turtle interpretation |
| $\Rightarrow^P$ | : | L-system rewrite |
| $\Rightarrow^D$ | : | Decomposition productions |

$$
\begin{array}{ccccccccccccc}
s_1 & \Rightarrow^D & d_1 & \Rightarrow^P & s_2 & \Rightarrow^D & d_2 & \Rightarrow^P & s_3 & \Rightarrow^D & d_3 & \Rightarrow^P & \ldots \\
 & & \Downarrow_H & & & & \Downarrow_H & & & & \Downarrow_H & & \\
 & & v_1 & & & & v_2 & & & & v_3 & & \\
 & & \Downarrow_T & & & & \Downarrow_T & & & & \Downarrow_T & & \\
 & & f_1 & & & & f_2 & & & & f_3 & &
\end{array}
$$

## 1.10 Programming language constructs

Our representation of L-systems is already a programming language in a sense, especially with the introduction of parameters. This section will describe how to combine L-systems with more traditional programming statements such as variables and blocks, and is based upon the *cpfg*[7] language [20].

Note that in chapter 4 we describe a different approach, creating an L-systems framework within an already existing programming language.

### 1.10.1 Predefined numerical functions

The inclusion of numerical functions was already mentioned in section 1.5. Most of the functions are taken from the standard C math library, the most important thing to notice is that the trigonometric functions all operate on degrees instead of radians. Notice also the different random functions.

The predefined functions are:

**Trigonometric functions:** $\sin(\alpha)$, $\cos(\alpha)$, $\tan(\alpha)$.

**Inverse trigonometric functions:** $\text{asin}(x)$, $\text{acos}(x)$, $\text{atan}(x)$, $\text{atan2}(x, y)$.

Except for returning degrees, these functions work like the standard C functions. $\text{atan2}(x, y)$ returns the arc tangent of $y/x$ as an angle between $-180°$ and $180°$, using the signs of both $x$ and $y$ to determine the quadrant.

**Rounding functions:** $\text{fabs}(x)$, $\text{floor}(x)$, $\text{ceil}(x)$, $\text{trunc}(x)$, $\text{sign}(x)$.

- $\text{fabs}(x)$ returns $x$'s absolute value.
- $\text{floor}(x)/\text{ceil}(x)$ rounds up/down.
- $\text{trunc}(x)$ rounds toward zero.
- $\text{sign}(x)$ returns 0 for $x = 0$, 1 for $x > 0$ and $-1$ for $x < 0$.

**Miscellaneous functions:** $\exp(x)$, $\log(x)$, $\text{sqrt}(x)$.

Logarithm, exponential and square root functions.

**Random functions:** $\text{srand}(x)$, $\text{ran}(x)$, $\text{nran}(x, \delta)$.

- $\text{srand}(x)$ initializes a random number generator.
- $\text{ran}(x)$ generates random numbers uniformly in the interval $(0, x)$.
- $\text{nran}(x, \delta)$ generates random numbers using a normal distribution with mean $x$ and standard deviation $\delta$.

**Output function:** $\text{printf}(format, ...)$.

The standard C printf function for output to a terminal window.

[7] *cpfg* is an acronym for Continuous Plant and Fractal Generator.

### 1.10.2 Statements

There will be three kinds of statements in the language, all with C-like syntax:

**Assignment:** Variable assignment is defined as:

$$variable = expression;$$

where *expression* can include all operators and functions explained this far.

**Conditionals:** There are two conditionals, an if-statement:

$$\text{if}\,(condition)\,\{stmt_1;\ \ldots\ stmt_n;\}$$

and an if-else-statement:

$$\text{if}\,(condition)\,\{stmt_1;\ \ldots\ stmt_n;\}\,\text{else}\,\{stmt_1;\ \ldots\ stmt_n;\}$$

**Loops:** There are two loop statements, a while-loop:

$$\text{while}\,(condition)\,\{stmt_1;\ \ldots\ stmt_n;\}$$

and a do-while-loop:

$$\text{do}\,\{stmt_1;\ \ldots\ stmt_n;\}\,\text{while}\,(condition);$$

### 1.10.3 Production statements

The syntax of a production with statements can be as follows:

$$lc < A > rc : \{\alpha\}\ condition\ \{\beta\} \rightarrow B$$

where $\alpha$ and $\beta$ are C-like statements.

Proper use of these statements can sometimes improve performance. Consider the production

$$A(x,y) : x * y > 1 \rightarrow B(x * y, x * y)$$

The multiplication $x * y$ is calculated three times.[8] With a production statement, this is easily fixed:

$$A(x,y) : \{z = x * y\}\ \ z > 1 \rightarrow B(z,z)$$

### 1.10.4 Global statements

We also have four blocks of global statements:

| | |
|---|---|
| Start: { $stmt_1;\ \ldots\ stmt_n;$ } | executed at the start of the simulation. |
| End: { $stmt_1;\ \ldots\ stmt_n;$ } | executed at the end of the simulation. |
| StartEach: { $stmt_1;\ \ldots\ stmt_n;$ } | executed at the start of each step. |
| EndEach: { $stmt_1;\ \ldots\ stmt_n;$ } | executed at the end of each step. |

The “Start” and “StartEach” statements are often used for initializing variables.

[8] A smart compiler could optimize it, but this is probably too much to expect from an L-systems program.

## 1.11 L-systems with a cut symbol

We can remove modules from an L-system string by using an *erasing production* such as $A \to \epsilon$. Since $\epsilon$ represents the empty string, this production means "replace $A$ by nothing" or in other words "erase $A$".

Sometimes we may need a way to remove whole branches instead of just simple modules, and for this purpose a special cut symbol % is introduced. A cut symbol inserted in the string will be removed in the next step together with the rest of the current branch. That is, the cut symbol and all subsequent modules until an unmatched ] (or the end of the string) will be removed.

For instance, the following L-system:

$$\begin{array}{ccl} \omega & : & ab[abc[ab]a]abcab \\ a & \to & b \\ b & \to & a \\ c & \to & \% \end{array}$$

will produce the strings

$$\begin{array}{c} ab[abc[ab]a]abcab \\ \Downarrow \\ ba[ba\%[ba]b]ba\%ba \\ \Downarrow \\ ab[ab]ab \\ \Downarrow \\ \ldots \end{array}$$

Cut symbols are useful when simulating plant parts that for some reason fall off or are cut off the plant, whether it be leaves, fruit or whole branches.

Figure 1.10 shows an L-system tree where every branch has a small chance (5%) of being cut off at each step. Every cut branch is replaced by a new shoot, as seen by the (stochastic) production `X --> A%`.

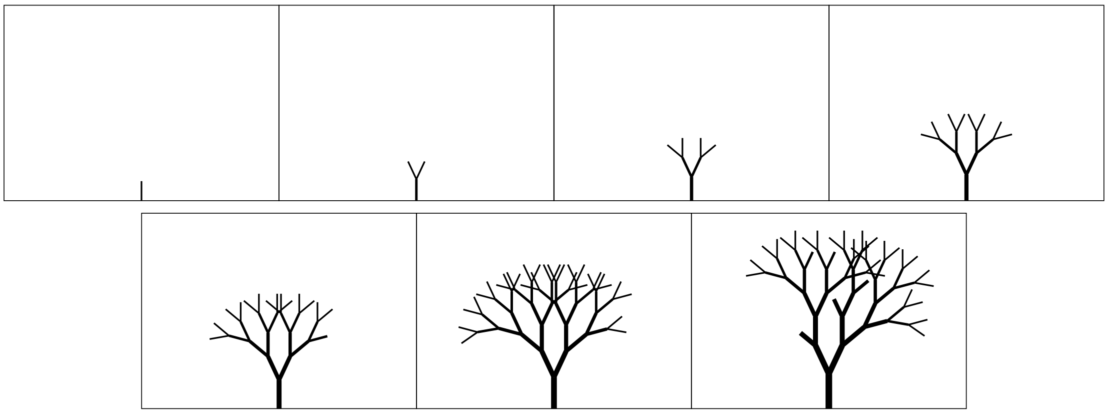
## 1.12 Visually defined spline functions

Sometimes we want to model plants or plant parts with a certain shape or other property that is best mimicked by drawing and experimenting rather than by mathematical definitions. This can be combined with L-systems by using a *spline editor* or a similar tool to define a function *visually*, then use the resulting function in the L-system definition for producing the shape. This way the shape is not "hard-coded" into the L-system, but also depends on the spline function, and the user can experiment by changing the spline.

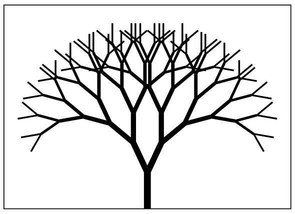
Figure 1.10: Cutting off branches

```
Axiom: !(1)A
!(x) --> !(x+1)
F(x) --> F(x*1.1)
A    --> F(1)[+(25)!(1)XA][-(25)!(1)XA]
X    --> X   : 19
X    --> A%  :  1
```

Steps 1–7:



By comparison, the full tree with no cutting would look like this:



A schematic example of an L-system with a spline function is shown in figure 1.11. Two spline functions together with the resulting tree structures are also shown. The value of the spline function `spline_func(x)` is used to determine the length of a branch at a distance `x` from the root of the tree; as a result, the overall shape of the tree corresponds almost exactly to the spline. Note how this L-system only contains one production, and it is a homomorphism production. Using only homomorphism productions is often best when we do not simulate the actual growth of the plant.

The details of how the spline function is imported into the L-system will depend on the L-system program, and is not specified in the figure.

It is of course possible to use several spline functions to describe different properties of the plant. Figure 1.12 shows an example of this. As in the previous example, the first spline function determines the length of the branches, and hence the overall shape. The second spline determines the angle of the branches: positive values make the branches point upwards and negative downwards.

Stochastic L-systems are good for producing different plant models that look like they belong to the same species. With visually defined functions we can use a very different approach and create models that look like *specific* plants.

Figure 1.11: L-system with a spline

```
#define LENGTH 1.0
#define SEGMENTS 40
#define SEGMENT_LENGTH (LENGTH/SEGMENTS)
#define WIDTH_MULTIPLY 1.0
#define spline_func(x) (...)

Axiom: F(SEGMENT_LENGTH)A(0)

homomorphism
A(x) : x < LENGTH { width = spline_func(x/LENGTH)*WIDTH_MULTIPLY; }
     --> [+(90)F(width)][-(90)F(width)]
         F(SEGMENT_LENGTH)A(x+SEGMENT_LENGTH)
```

**Spline function** **Corresponding model**

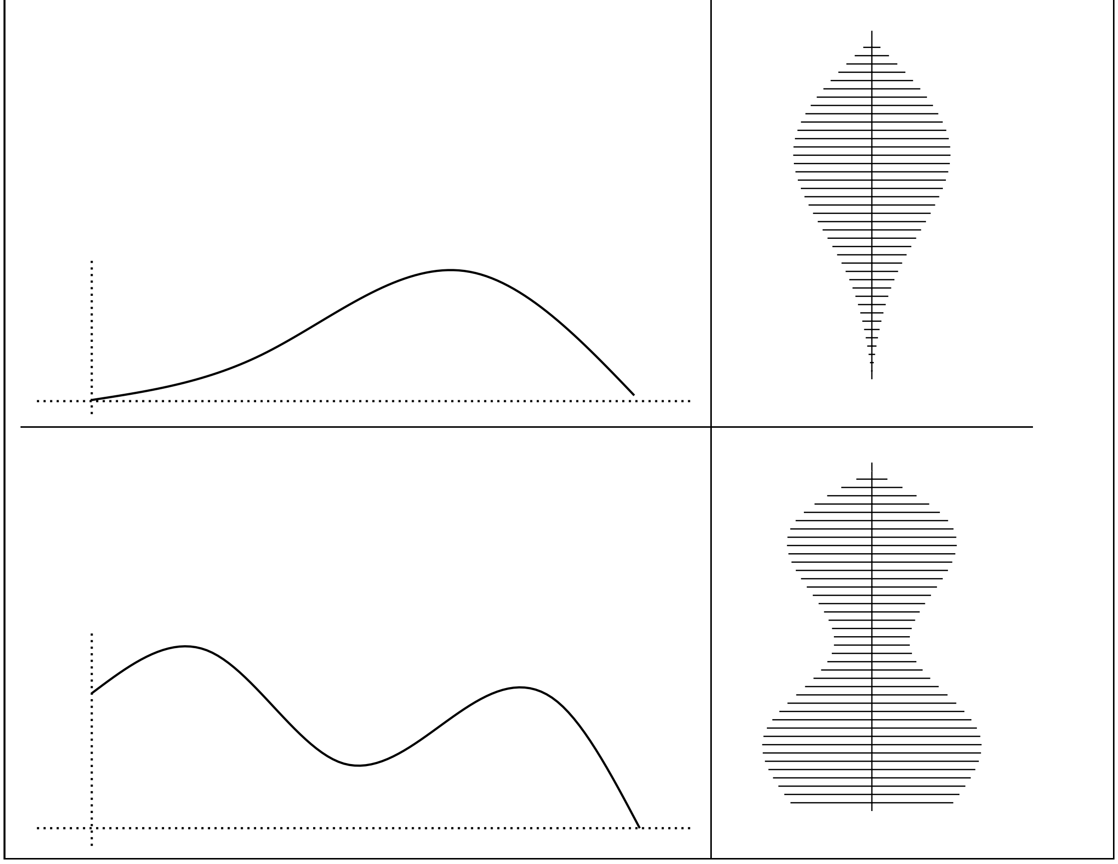
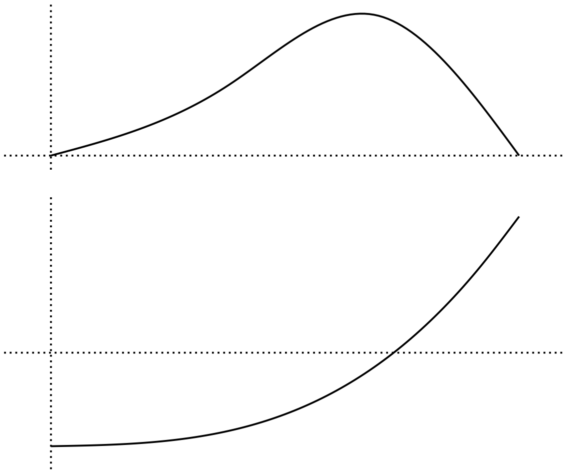
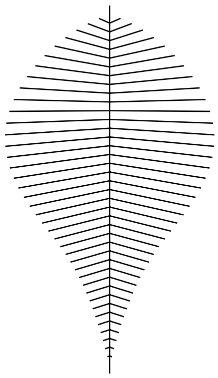
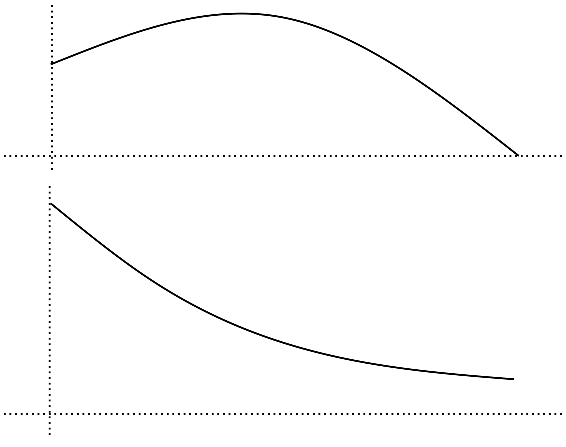
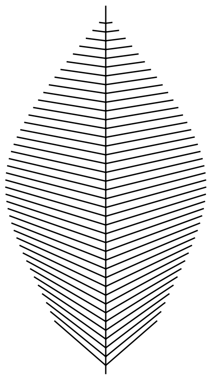


Figure 1.12: L-system with two splines

```
#define LENGTH 1.0
#define SEGMENTS 40
#define SEGMENT_LENGTH (LENGTH/SEGMENTS)
#define WIDTH_MULTIPLY 1.0
#define ANGLE_MULTIPLY 90.0
#define spline_func_1(x) (...)
#define spline_func_2(x) (...)

Axiom: F(SEGMENT_LENGTH)A(0)

homomorphism
A(x) : x < LENGTH { width = spline_func_1(x/LENGTH)*WIDTH_MULTIPLY;
                    angle = spline_func_2(x/LENGTH)*ANGLE_MULTIPLY; }
     --> [+(90)-(angle)F(width)][-(90)+(angle)F(width)]
         F(SEGMENT_LENGTH)A(x+SEGMENT_LENGTH)
```

| **Spline functions** | **Corresponding model** |
|---|---|
|  |  |
|  |  |

# Chapter 2

# L-systems and the environment

A plant can be affected by, or interact with, the environment in many ways. We can distinguish between the global and local cases:

- Global environment variables affect the plant as a whole. Examples are temperature and oxygen levels. We already have a framework to deal with global changes; we can simply use global variables.
- The local case is when parts of the plant are affected differently according to their position or orientation in space. A simple example is a plant growing near a wall; a simulation should stop it from growing through the wall.

This chapter will examine the latter case, which is called *environmentally sensitive L-systems.* A more complex case where the L-system can also affect the environment, called *open L-systems*, will not be discussed here.

## 2.1 Environmentally sensitive L-systems

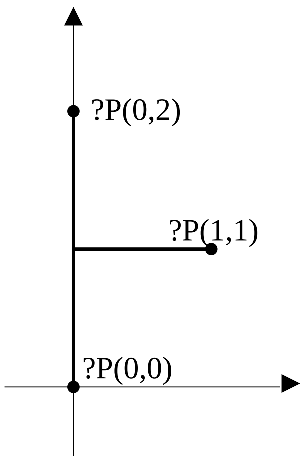
The position and orientation of plant parts is known at the time of turtle interpretation, but even if we do a turtle interpretation after each rewriting step, the information is not available during the next step.

Environmentally sensitive L-systems (see for instance [19, 14]) provide a set of *query modules* to save this information during turtle interpretation:

| | |
|---|---|
| $?P(x, y, z)$ | Get the turtle's position. |
| $?H(x, y, z)$ | Get the turtle's heading vector (forward direction). |
| $?U(x, y, z)$ | Get the turtle's up vector. |
| $?L(x, y, z)$ | Get the turtle's left vector. |

The $x$, $y$ and $z$ values are changed during each turtle interpretation, which is done after the normal rewrite (and homomorphism/decomposition, if present).

Consider the rewriting string $?P(x,y)F[-(90)F?P(x,y)]F?P(x,y)$. In this case the $z$ coordinate is ignored. The actual $(x,y)$ values are meaningless at this stage; they can be set to $(0,0)$ or random values. If the turtle's initial position is at the origin and it is heading along the $y$-axis, the $?P$ parameters will be changed as described by the following figure:




Thus, after turtle interpretation, the rewriting string will have changed to $?P(0,0)F[-(90)F?P(1,1)]F?P(0,2)$.

## 2.2 Pruning

Pruning (cutting) trees or bushes to shapes can be simulated by environmentally sensitive L-systems. A simple solution is to use $?P$ query modules at the branch apices, and keep the branches growing until they are outside a predefined prune shape, then stop the growth.

Figure 2.1 shows a very simple example of 2D pruning. Notice how the branches grow one step beyond the prune shape. To avoid this, we can use a two-face simulation, where we alternate between growing and cutting.

Figure 2.2 shows an example of this. During the grow step all branches that have not already been cut have a query module at the apex. Those branches then grow one step further. During the cut step the branches that have grown past the prune shape are cut away together with the query module.

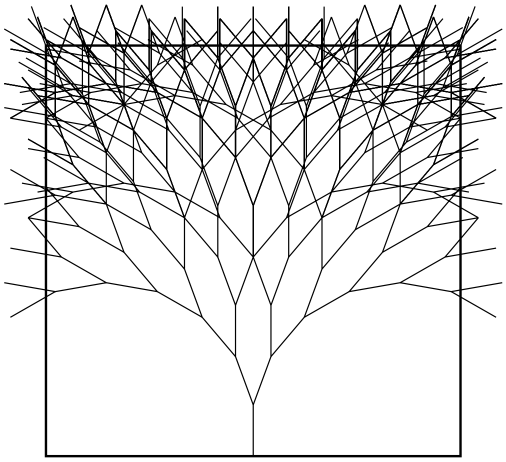
Ideally we should be able to use decomposition productions to cut the branches, but since the L-system depends on query modules this is not easy to achieve. The problem is that we need the turtle to fill in the position of each branch, but there is no turtle interpretation step between L-system rewriting and decomposition rewriting.

Extending the pruning L-systems to 3D is straightforward. Figure 2.3 shows a rendering of a plant pruned to a cylinder.

Figure 2.1: 2D pruning

```
#define PRUNE(x,y) ((x<-4) or (x>4) or (y<0) or (y>8))
```

| | | | | |
|---|---|---|---|---|
| $\delta$ | : | $20°$ | | |
| Depth | : | 10 | | |
| $\omega$ | : | `F?P(0,0)` | | |
| `?P(x,y)` | : | `!PRUNE(x,y)` | $\rightarrow$ | `[+F?P(0,0)]-F?P(0,0)` |
| `?P(x,y)` | : | `PRUNE(x,y)` | $\rightarrow$ | $\epsilon$ |



## 2.3 Climbing plants

As shown by Měch [14], environmentally sensitive L-systems can be used to detect if a plant collides with objects in the environment in the same manner as pruning L-systems detect growth outside the prune volume.

A simple climbing plant model can be created by creating a stem of segments, and let the apex segment search for support using environmental sensitivity. The apex can search for support by the following algorithm:

1. Pitch the apex segment down by fixed angle steps until the tip hits an obstacle, or until a maximum pitch angle is reached. (*Search mode*)

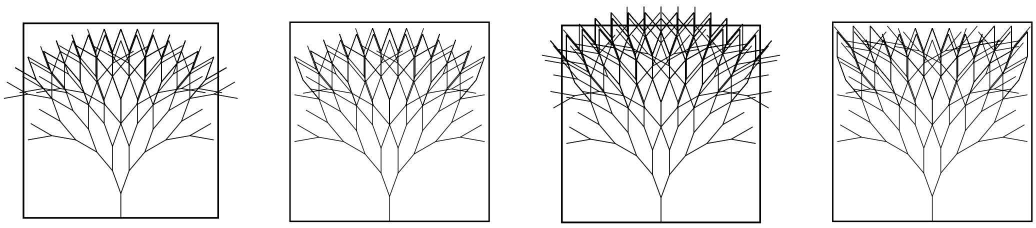
2. If no obstacle is found, return the segment to upright position, change the pitch direction by "rolling" it a fixed angle (say, 45 degrees) and go to step 1.

3. When the apex hits an obstacle, pitch it backwards (using smaller steps than in step 1) until it is out of the obstacle. (*Backtrack mode*)

Figure 2.2: Two-face pruning

```
#define PRUNE(x,y) ((x<-4) or (x>4) or (y<0) or (y>8))
#define GROW 0
#define CUT  1
Angle     : 20
Start     : { mode = GROW; }
StartEach : { if (mode == GROW) { mode = CUT; }
              else { mode = GROW; } }
Axiom     : F?P(0,0)

?P(x,y) : (mode == GROW)
            --> [+F?P(0,0)][-F?P(0,0)]
F < ?P(x,y) : (mode == CUT and PRUNE(x,y))
            --> %
```

Steps 14–17:



4. Grow a new apex segment and go to step 1.

The search and backtrack method is illustrated in figure 2.4, and an L-system that uses this algorithm is given in figure 2.5.

Some further refinements of the climbing plant model are relatively easy to add, such as:

- Adding leaves to the stem.
- Creating branches as the stem grows.
- Multiple obstacles for the plant to attach to.

Figure 2.6 shows such a model of some plants climbing a wall.

## 2.4 Tropism

Tropism is the study of branches bending towards certain orientations, often straight upwards or downwards (gravitropism), but more complex examples, such as plants turning towards sunlight or exposed to wind, are not uncommon.

Figure 2.3: 3D pruning

Figure 2.4: Climbing plant: searching and backtracking

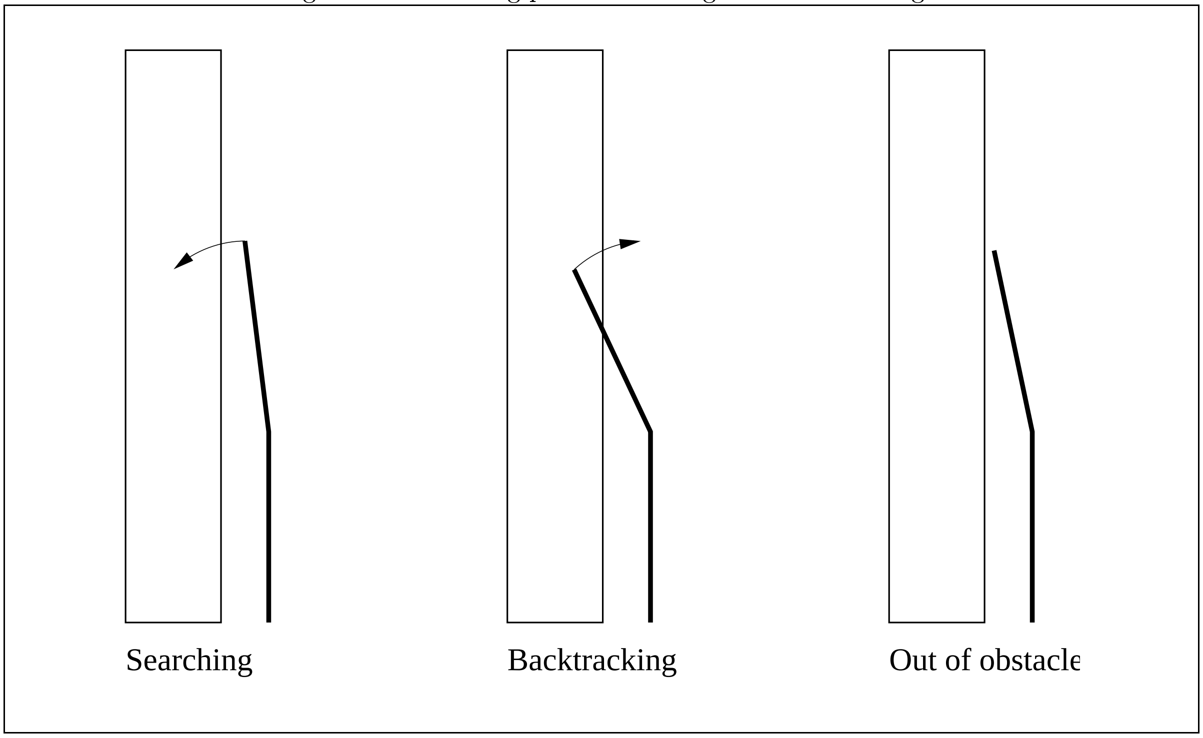
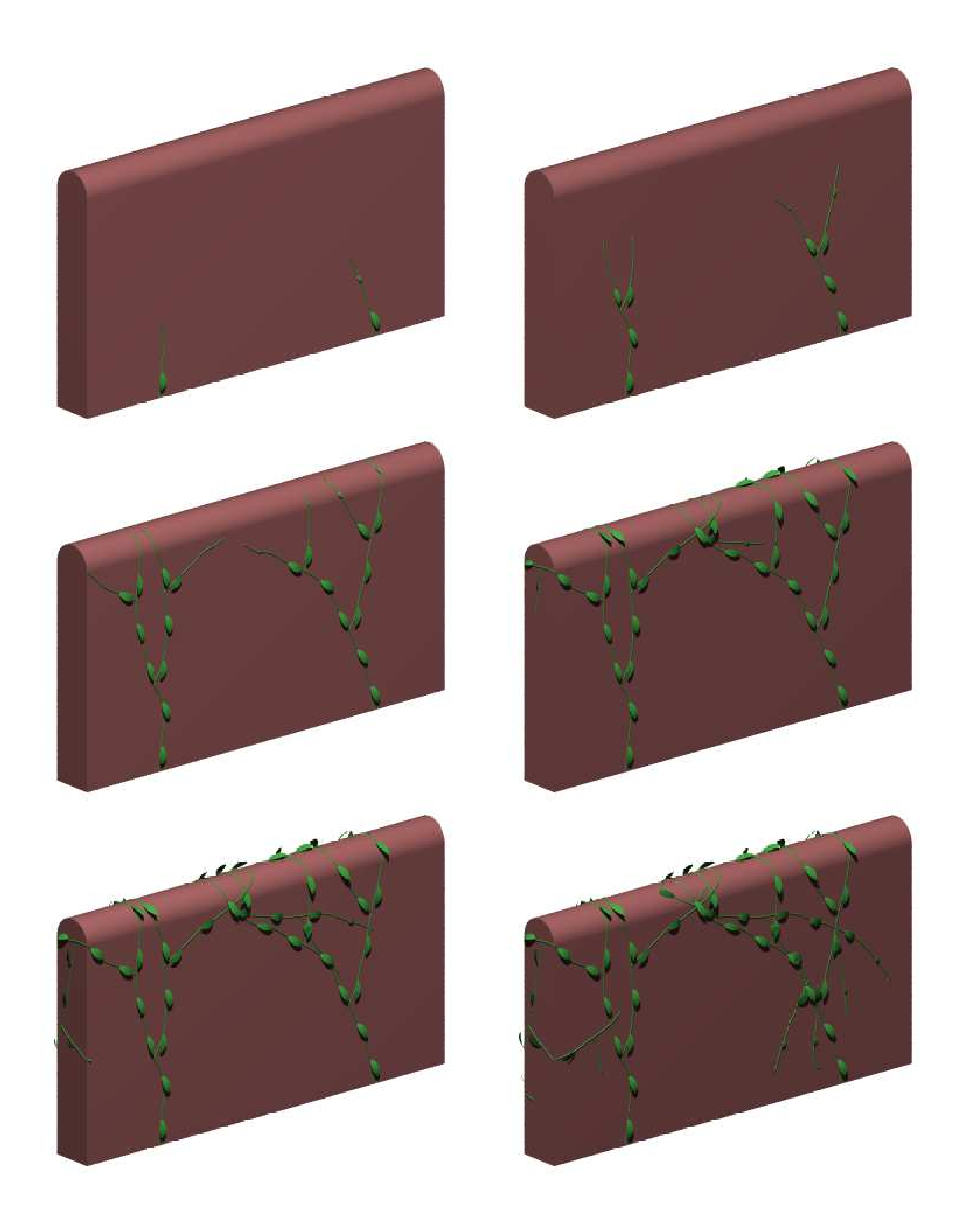



Figure 2.5: Climbing plant L-system

```
#define MAXP 95.0  /* max pitch */
#define PSTEP 5.0  /* pitch step */
#define BSTEP -2.0 /* backtrack step */
#define RSTEP 45   /* roll step */
#define crash(x,y,z) (...) /* true if in obstacle */

/* searching apex */
A(roll,pitch) > ?P(x,y,z) : crash(x,y,z) --> B(roll,pitch)
A(roll,pitch) : pitch < MAXP --> A(roll,pitch+PSTEP)
A(roll,pitch) --> A(roll+RSTEP,0)
/* backtracking apex */
B(roll,pitch) > ?P(x,y,z) : crash(x,y,z) --> B(roll,pitch+BSTEP)
B(roll,pitch) --> S(roll,pitch)+(nran(0,1))A(0,0)

homomorphism
A(roll,pitch) --> S(roll,pitch)
B(roll,pitch) --> S(roll,pitch)
S(roll,pitch) --> /(roll)&(pitch)F
```

Figure 2.6: Plants climbing a wall



We will give some brief examples of L-systems that simulate tropism. First, assume that we have a turtle command $@R(a, x, y, z)$ that rotates the turtle an angle $a$ towards the vector $(x, y, z)$. Then simulating a simple form of tropism is straightforward; just rotate the turtle slightly towards a predefined vector before each branch segment is drawn. An example of this is given in figure 2.7. The turtle is rotated towards the vector $(TX, TY, TZ)$, which will be called the *tropism vector*.

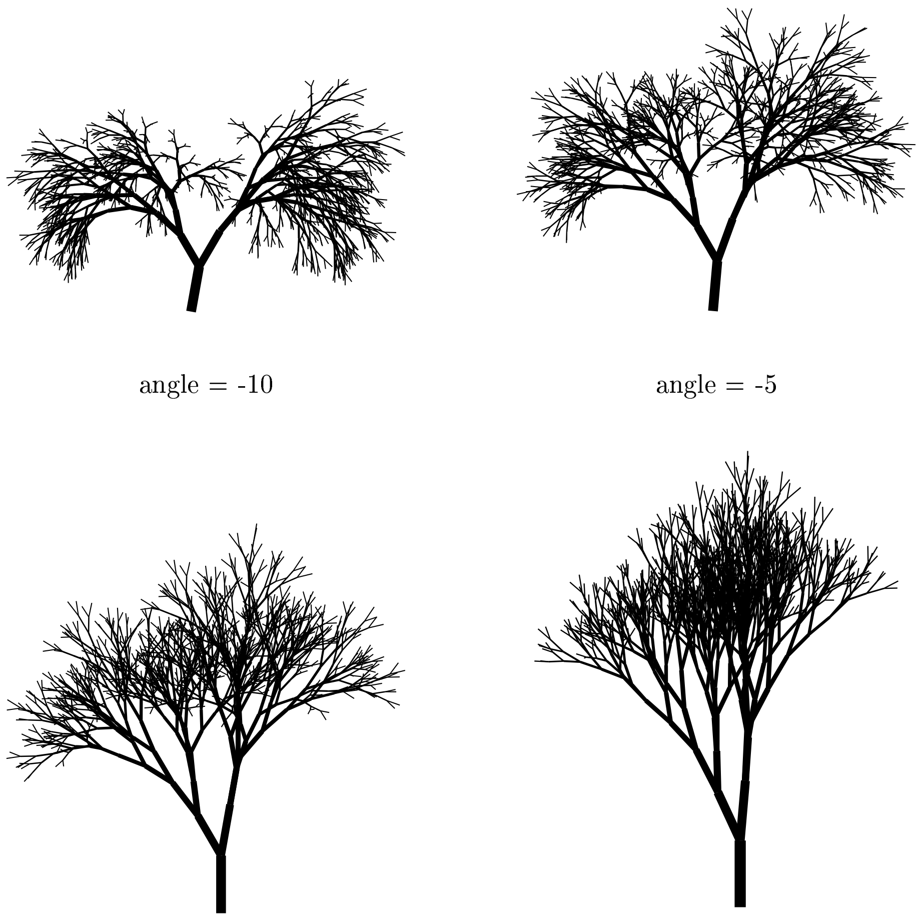
One disadvantage of the previous example is that the turning angle is the same independently of the branch's length and heading. A more physically based model (described in [17]) is to calculate the turning angle $\alpha$ by the formula $\alpha = e\mathbf{H} \times \mathbf{T}$, where $\mathbf{H}$ is the segment's heading, $\mathbf{T}$ is the tropism vector and $e$ is a parameter, called the *elasticity*, which is a measurement of how much the branch can bend.

To calculate the turning angle $\alpha$ in an L-system, we need the turtle's heading at each branch. To accomplish this, we can use an environmentally sensitive L-system with the query module $?H$, as shown in figure 2.8. Note that the elasticity $e$ does not have to be a constant; in our example it is a function $e(w)$ where $w$ is the branch segment's width. Figure 2.9 shows examples of generated trees with different tropism vectors and elasticity functions.

Figure 2.10 shows a 3D rendering of a tree which uses tropism to create a "weeping willow" effect.

Figure 2.7: Simple tropism

```
#define TROPISM_ANGLE 3.0
#define TX 0
#define TY 1
#define TZ 0

Axiom: A(100,12)
Depth: 12

A(s,w) --> S(s,w)
           [+(30)/(137)A(s*0.8,w*0.8)]
           [-(10)/(-90)A(s*0.9,w*0.8)]

homomorphism
S(s,w) --> @R(TROPISM_ANGLE,TX,TY,TZ)!(w)F(s)
```



angle = -10

angle = -5

angle = 0

angle = 5

Figure 2.8: Improved tropism

```
#define TX 0
#define TY 1
#define TZ 0
#define elasticity(w) ...
#define cross_product(x,y,z, ax,ay,az, bx,by,bz) \
  { x = ay*bz - az*by; y = az*bx - ax*bz; z = ax*by - ay*bx; }
#define length(x,y,z) sqrt(x*x+y*y+z*z)

Axiom: A(100,12)?H(0,0,0)
Depth: 12

A(s,w) > ?H(hx,hy,hz)
  --> S(s,w,hx,hy,hz)
      [+(30)/(137)A(s*0.8,w*0.8)?H(0,0,0)]
      [-(10)/(-90)A(s*0.9,w*0.8)?H(0,0,0)]
?H(x,y,z) --> *

homomorphism
S(s,w,hx,hy,hz)
  { el = elasticity(w);
    cross_product(x,y,z, hx,hy,yz, TX,TY,TZ)
    angle = el*length(x,y,z); }
  --> @R(angle,TX,TY,TZ)!(w)F(s)
```

Figure 2.9: Tropism trees

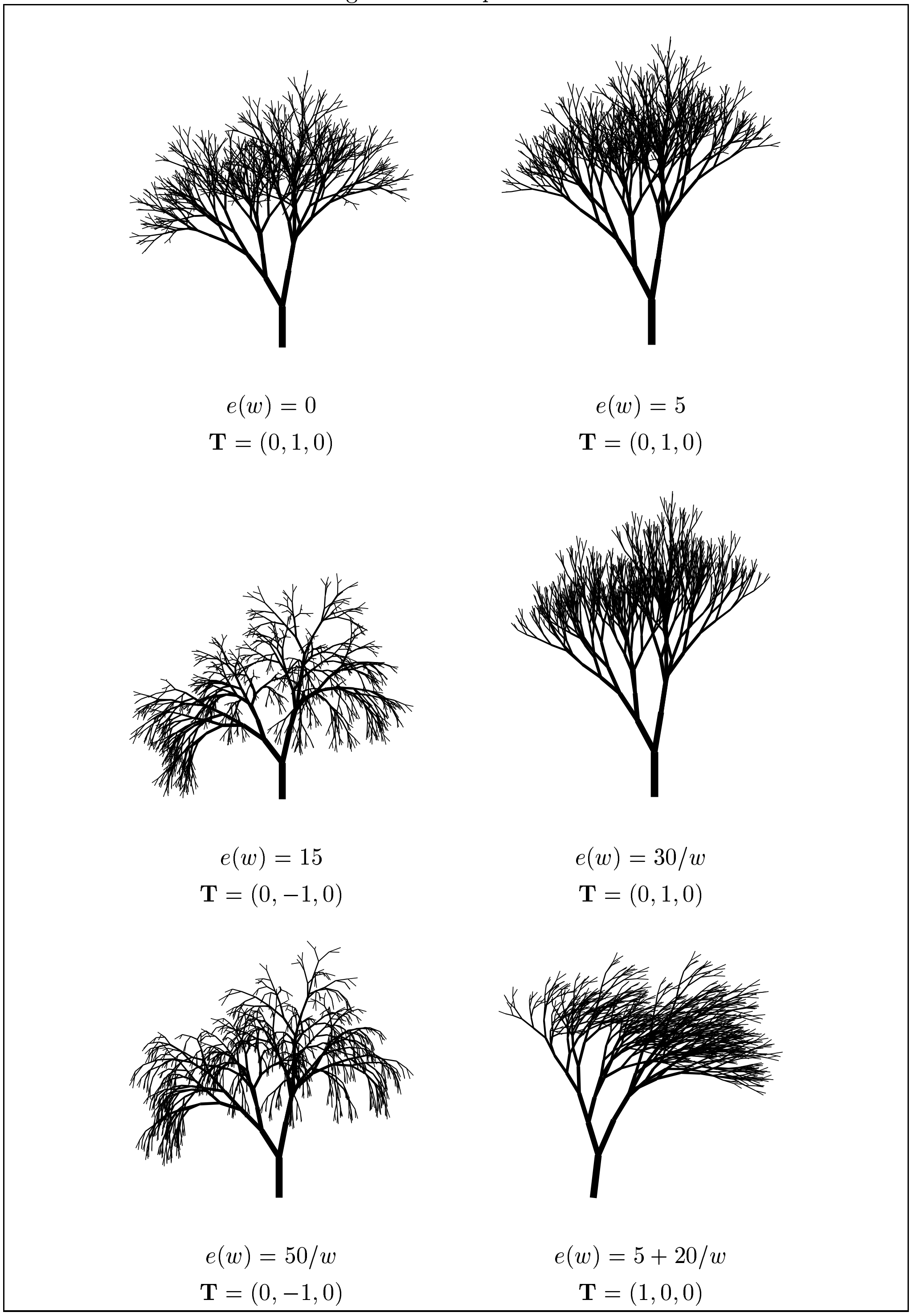


Figure 2.10: Weeping willow

# Chapter 3

# Modeling twining plants

This chapter examines the modeling of twining plants through L-systems. A twining plant is a climbing plant that seeks support by twining around poles or other plants. Different aspects like stem elongation, curvature and attachment to surfaces are described separately and then combined in a final model, which is an example of advanced use of L-systems.

## 3.1 Stem elongation

We first consider growth of a stem as a one-dimensional case; in other words, we consider only the elongation of the stem, not the curving.

The area of elongation of a stem (and also a root) is constant, seen from the tip (also called the apex). In many plants the area of growth is small, often only a few millimeters. In such cases a model that only grows exactly at the apex can be a good enough approximation. This means that once a leaf has been formed, it stays in fixed position from the base. In twining plants, however, the area of growth tends to be larger, so a leaf will follow the "flow" of the stem for a while.

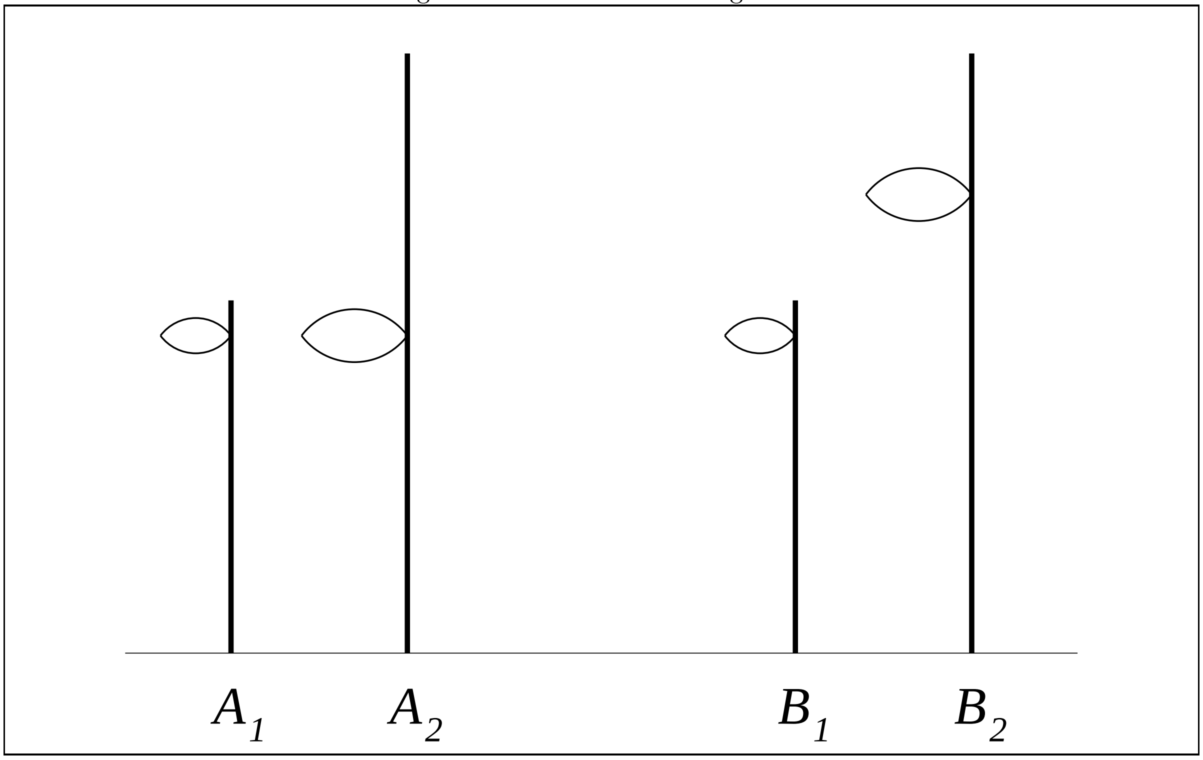
The difference between a small and a large area of growth is shown schematically in figure 3.1. Both plants $A$ and $B$ produce leaves near the apex, as shown by $A_1$ and $B_1$. Plant $A$ has a small area of growth, so the leaf's position from the ground (root) does not change with time, as shown by $A_2$. Plant $B$ has an area of growth large enough that the leaf follows the growth of the stem for a while, as shown by $B_2$.

A large area of growth can also be seen in the roots of many plants. The elongation of the root of *Zea Mays* has been studied by Erickson and Sax [4, 5].

### 3.1.1 Mathematical background

We will use a *kinematic description of growth* [6, 7] to describe the elongation mathematically.

Figure 3.1: Leaves and stem growth




Consider a particle on the stem which is at a distance $x$ from the apex; the particle will simply be called $x$. Then, at some given time, $v(x)$ is the *spatial* velocity function, that is, the velocity of $x$ relative to the apex. This function will be zero at the apex, increase through the area of growth, then typically be constant for higher $x$ values.

The gradient of this velocity, $dv(x)/dx$, is a measurement of the elongation rate of the cells in the region near $x$. It will be called the *elongation function* from now on.

It turns out that these functions do not change much over time. Overall the stem grows at a nearly constant rate once it is longer than its area of growth.

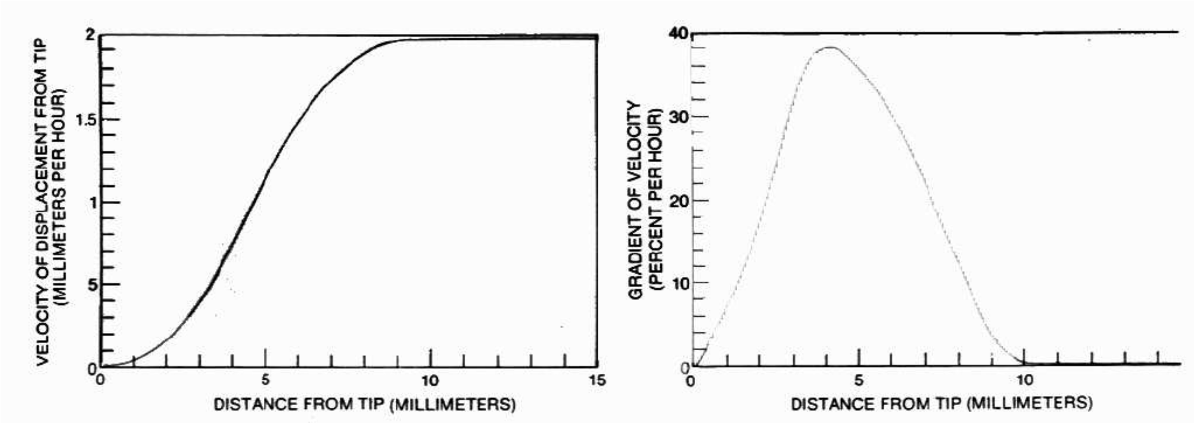
Figure 3.2 (from [6]) illustrates the rate of elongation for a corn root. This elongation function is for a root, not a stem, but the same concepts apply to the stem of most twining plants.

### 3.1.2 L-system example

To simulate the stem elongation, we will use an L-system with a spline (section 1.12) that approaches the elongation function. The L-system string is a sequence of stem segments `A(s,p)`, where `s` is the segment's length and `p` is its position on the stem. The length of the stem's area of growth is specified by the constant `GROW_AREA`.[1] Segments further away from the tip than `GROW_AREA` do not grow, segments

[1]The spline function's x axis is set from 0.0 to 1.0, and `GROW_AREA` is used to scale this size.

Figure 3.2: Elongation function of a corn root (from [6])




within the area grow according to the percentage given by the spline function.

The L-system is given in figure 3.3. Note that there are some inaccuracies in the way the elongation function is used:

- Point sampling is used to find the growth rate of a single segment. An integration of the elongation function would produce a more accurate answer. If the length of the segments is small, the difference will be of little importance.
- The length of the stem is needed to calculate the current distance from the tip, but this length is not known until all segment lengths are calculated. The current solution is to use the stem length of the previous time step, and keep the time steps small enough that the difference is negligible.

The same principles will be used in the twining plant model, but no visualizations are given for this L-system. We could use a homomorphism production $A(s,p) \rightarrow F(s)$ to visualize a straight stem, but the result would not be interesting.

## 3.2 Stem curvature

We now turn to the curving of the stem rather than the elongation. From a biological perspective, curving can be a result of faster cell growth on the outer side than the inner side of the stem, but our model will describe growth and curving separately.

### 3.2.1 Mathematical background

The curvature of a plane (2D) curve is intuitively the “amount of turning” at each point of the curve. Formal terminology and definitions are given below:

A finite parametric curve can be expressed as

Figure 3.3: Elongation function L-system

```
#define GROW_AREA 5.0
#define MAX_LENGTH 1.0
#define spline_func(x) (...) /* implementation-dependent */
#define GROW(s,p,length) ((length-p > GROW_AREA) ? s : \
        s + s*spline_func((length-p)/GROW_AREA))

Axiom: A(1.0, 0.0)

Start: { position = 1.0; }
StartEach: {
  length = position;
  position = 0.0;
}

A(s,p) { new_s = GROW(s,p,length);
         this_position = position;
         position = position + new_s; }
  --> A(new_s, this_position)

decomposition
A(s,p) : s > MAX_LENGTH --> A(s/2, p)A(s/2, p+s/2)
```

$$\mathbf{r}(t) = x(t)\mathbf{i} + y(t)\mathbf{j}, \quad a \leq t \leq b$$

A parametrization like this can represent a particle which moves along the curve such that it is a the point $(x(t), y(t))$ at time $t$. The velocity of the particle is then given as

$$\mathbf{v}(t) = \frac{d\mathbf{r}(t)}{dt}$$

We can also define a tangent vector which is parallel with the velocity and has unit length:

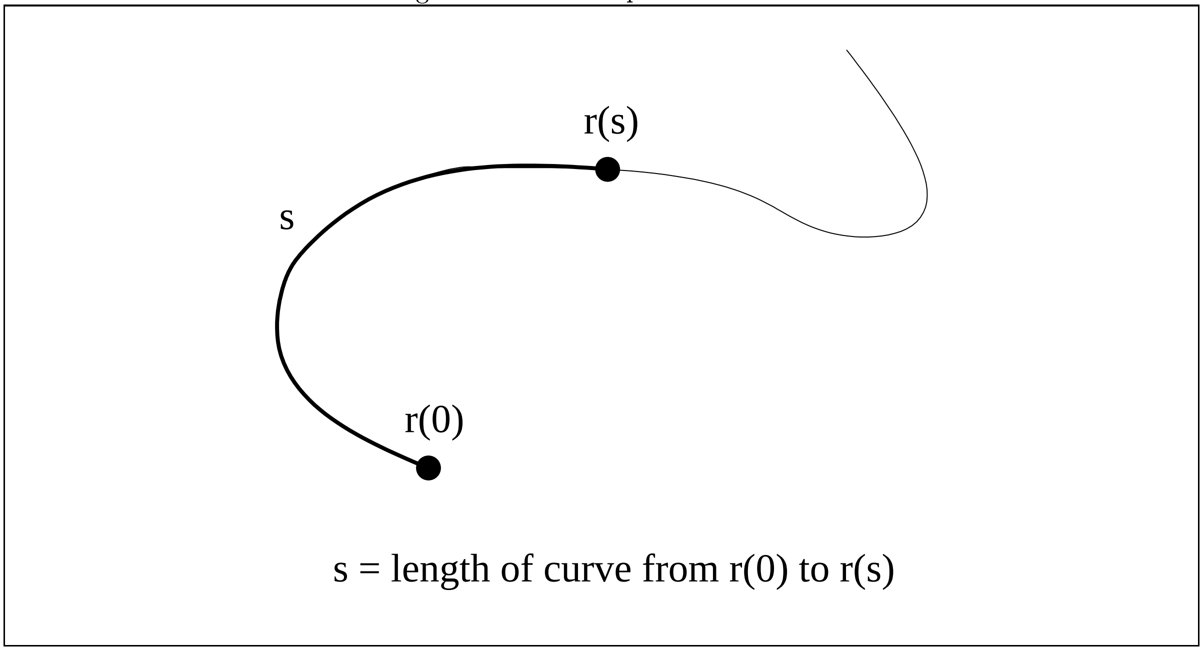
$$\mathbf{T}(t) = \frac{\mathbf{v}(t)}{|\mathbf{v}(t)|}$$

A specific curve generally has infinitely many possible parametrizations. Of particular interest is the parametrization where the length of the curve from the starting point at $\mathbf{r}(0)$ to $\mathbf{r}(s)$ equals $s$ for all $\mathbf{r}(s)$ on the curve. This is called an *intrinsic* or *natural* parametrization; see figure 3.4. Some properties of such a parametrization are $v(t) = |\mathbf{v}(t)| = 1$ and $\mathbf{v}(t) = \mathbf{T}(t)$. Natural parametrizations of curves are not always easy (or even possible) to find, but they always exist for continuous curves.

The *curvature* $\kappa$ of a curve $C$ expressed as a natural parametrization $\mathbf{r}(s)$ is defined as

$$\kappa(s) = i\left|\frac{d\mathbf{T}}{ds}\right|$$

Figure 3.4: Natural parametrization




where $i = 1$ if the curve is turning counterclockwise (the angle between $\mathbf{T}$ and $d\mathbf{T}/ds$ is 90°), and $-1$ otherwise (the angle is $-90°$)[2].

The amount $|1/\kappa(s)|$ is called the *radius of turning*, because the curve approximates a circle arc of this radius at $s$.

The fundamental theorem of 2D curves says that two curves with the same curvature are congruent. This means that a curve is defined completely by its curvature, starting point and direction.

### 3.2.2 Curvature and turtle graphics

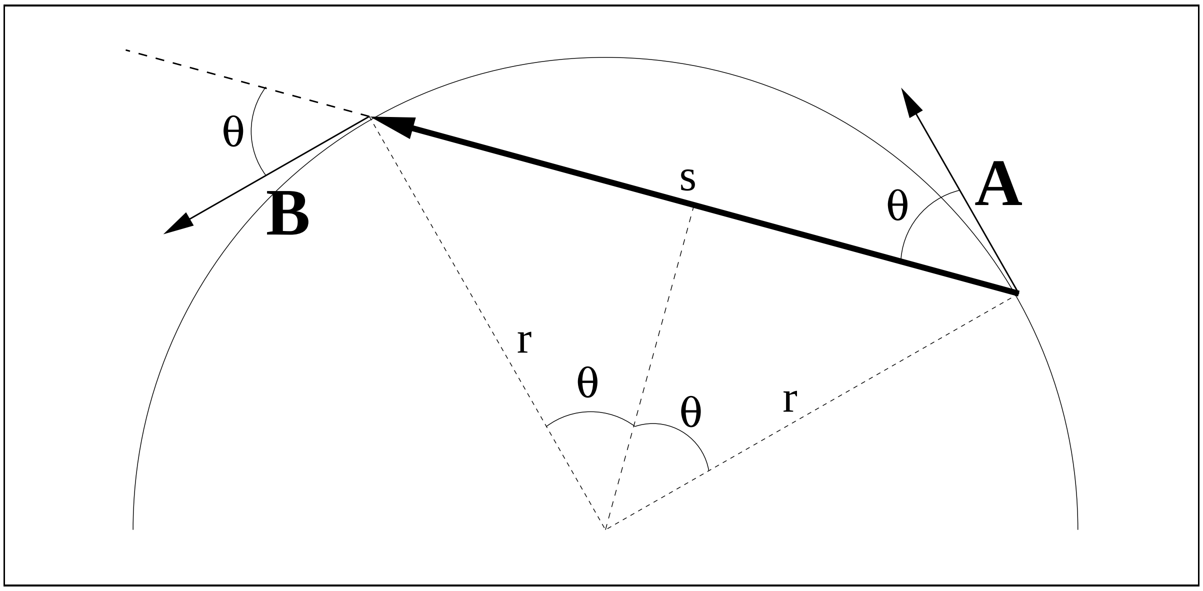
Turtle graphics can produce a sequence of line segments that approximates a curve, but it can obviously not represent a continuous curve exactly.

Consider the simple case where the curvature is constant; the curve will then be a circle arc (or a straight line if $\kappa(s) = 0$). The turtle string `F+(x)F+(x)+F+(x)...` will produce a sequence of segments that approximates a circle arc. However, the case where each line segment can be of different length is more interesting. We need to find out how much we should turn at each step to approximate an arc with curvature $\kappa$.

In figure 3.5, the turtle starts in position $\mathbf{A}$, turns an angle $\theta$, moves forward a distance $s$ (drawing a line), and finally turns an angle $\theta$ again, ending up in position $\mathbf{B}$. In both positions $\mathbf{A}$ and $\mathbf{B}$ the turtle's

[2] For 3D curves, curvature is usually defined so that it is always nonnegative, and a second property called *torsion* is used to describe the "twist" of the 3D curve. For 2D curves it makes sense to define that the curvature is positive when the curve turns in one direction, and negative in the other.

Figure 3.5: Curvature




heading is parallel with the arc's tangent. If we can calculate the turning angle $\theta$ from the distance $s$ and the radius $r = 1/|\kappa|$, then a sequence of these turtle commands will approximate an arc even if $s$ is different for each step.

As seen from the figure, the turning angle can be calculated as

$$\sin\theta = \frac{s/2}{r}$$

If $\theta$ is small, and as long as $s$ is short compared to $r$, which it will be (although it is not in the figure), then $\sin\theta \approx \theta$. So we get

$$\theta \approx \frac{s}{2r} = \frac{|s\kappa|}{2}$$

We actually want both the amount and direction of turning, and since the curvature's sign defines the direction of turning, we simply remove the absolute value and say

$$\theta \approx \frac{s\kappa}{2}$$

The numerical errors by using this method can be summarized in three points:

- The errors due to setting $\sin\theta = \theta$. This can of course be corrected by calculating the arcsine of the answer, but the error is usually small enough that it is not necessary.
- If the line segments are of total length $s$, the approximation will "reach" slightly longer than an arc of length $s$, because each line segment is a "short cut" compared to the arc.

- Floating point inaccuracy.

The two first error sources can be reduced by making the line segments shorter. But this can make the floating point inaccuracy worse, since a long series of turtle commands (or other floating point operations) will tend to create larger errors. But the floating point errors are probably negligible anyway. More important is the fact that more and shorter line segments will lead to slower computations.

Often the point is not to approximate an arc with a *specific* curvature and length; it doesn't matter if the arc is a little too big or too small, as long as it is in fact close to an arc. And if the length of the line segments are roughly equal, this is exactly what will happen. $\theta$ is slightly larger than $\sin\theta$, but if the segment lengths are almost equal, then the error for each step will be almost equal, and the approximation will be slightly larger than it should be, but still very close to an arc. Visually speaking it will usually not matter.

### 3.2.3 L-system examples

For L-systems, the turtle commands in the previous section are easily expressed using a homomorphism production. For instance, say that we have a sequence of modules on the form `A(s)` and we want each of these to represent a line segment on an arc approximation where $K = \kappa/2$[3]. Then we can simply use the homomorphism production

```
A(s) --> +(s*K)F(s)+(s*K)
```

A complete L-system of growth with constant curvature is given in figure 3.6. The L-system production specifies how the segments grow for each time step (rewrite), and the decomposition production how they split when they become longer than `MAXLENGTH`. The homomorphism production converts the segment to turtle commands. The image shows what the result would be after a few hundred steps, but this L-system would be better viewed as an animation.

### 3.2.4 Variable curvature

So far, we have assumed that the curvature is constant, so that the result is a circle arc. We can also use curvature mathematics to approach other kind of curves. We will look briefly at how to create an L-system with variable curvature defined by a *spline function* (see section 1.12, page 21), even though this approach will not be used in the final twining plant model.

With variable curvature, we can obviously not assume that the curve is a circle arc. But if we divide the curve in small segments, each segment will approximate a circle arc when the segment gets smaller. In other words, if the segments are small enough, we can assume that each segment has constant curvature and only get a small error.

See figure 3.7 for an example.

[3] If we are using degrees rather than radians, we should set $K = (180/\pi)(\kappa/2) = 90\kappa/\pi$, but usually we won't worry about the details, and just choose a $K$ that "looks good" instead.

Figure 3.6: Curvature L-system

```
#define K 10 /* A constant describing the curvature */
#define SPLITFACTOR 0.4
#define MAXLENGTH 1.5

Axiom: A(MAXLENGTH)

A(s) --> A(s*1.01)

decomposition
A(s) : s > MAXLENGTH --> A(s*SPLITFACTOR)A(s*(1-SPLITFACTOR))

homomorphism
A(s) --> +(s*K)F(s)+(s*K)
```

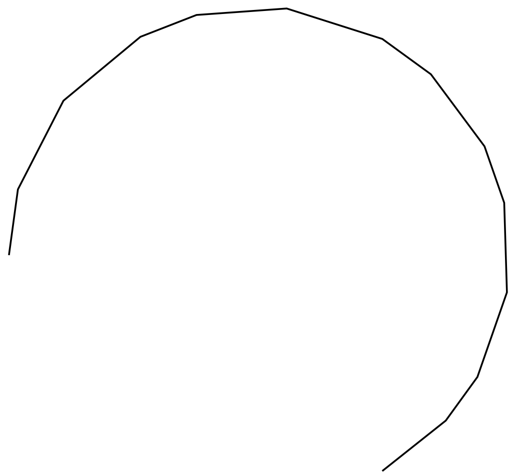
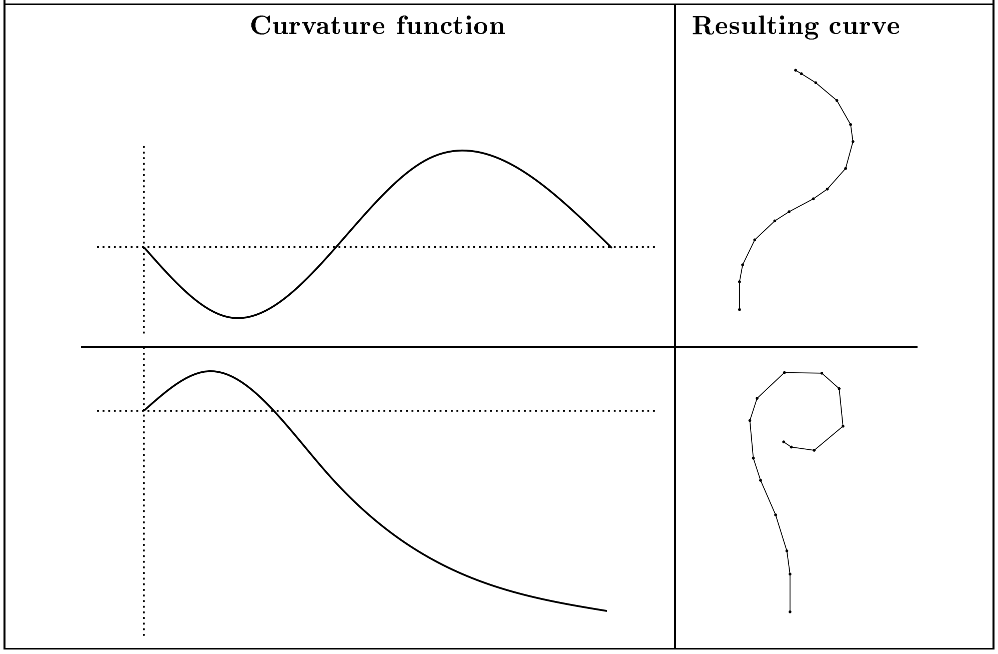


Figure 3.7: Variable Curvature L-system

```
#define SPLITFACTOR 0.4
#define MAXLENGTH 0.1
#define GROWTH 1.1
#define CURV_MULTIPLIER 100.0
#define spline_function(x) (...) /* system dependent */

Axiom: A(0.1, 0, 0)
StartEach: { spline_pos = 0.0; }

/* 1.0 is max length of the curve, so erase segments after that. */
A(x, c1, c2) : spline_pos >= 1.0 --> *
/* Grow the segment and find start and end curvatures. */
A(x, c1, c2) :
  { new_x = x * GROWTH;
    pos1 = spline_pos;
    pos2 = min(pos1 + new_x, 1.0);
    curv1 = spline_function(pos1);
    curv2 = spline_function(pos2);
    spline_pos = pos2; }
  --> A(new_x, curv1, curv2)

decomposition
A(x, c1, c2) : x > MAXLENGTH
  --> A(x*SPLITFACTOR, c1, c2)A(x*(1-SPLITFACTOR), c1, c2)

homomorphism
A(x, c1, c2)
  --> +(c1*x*CURV_MULTIPLIER)F(x)+(c2*x*CURV_MULTIPLIER)
```



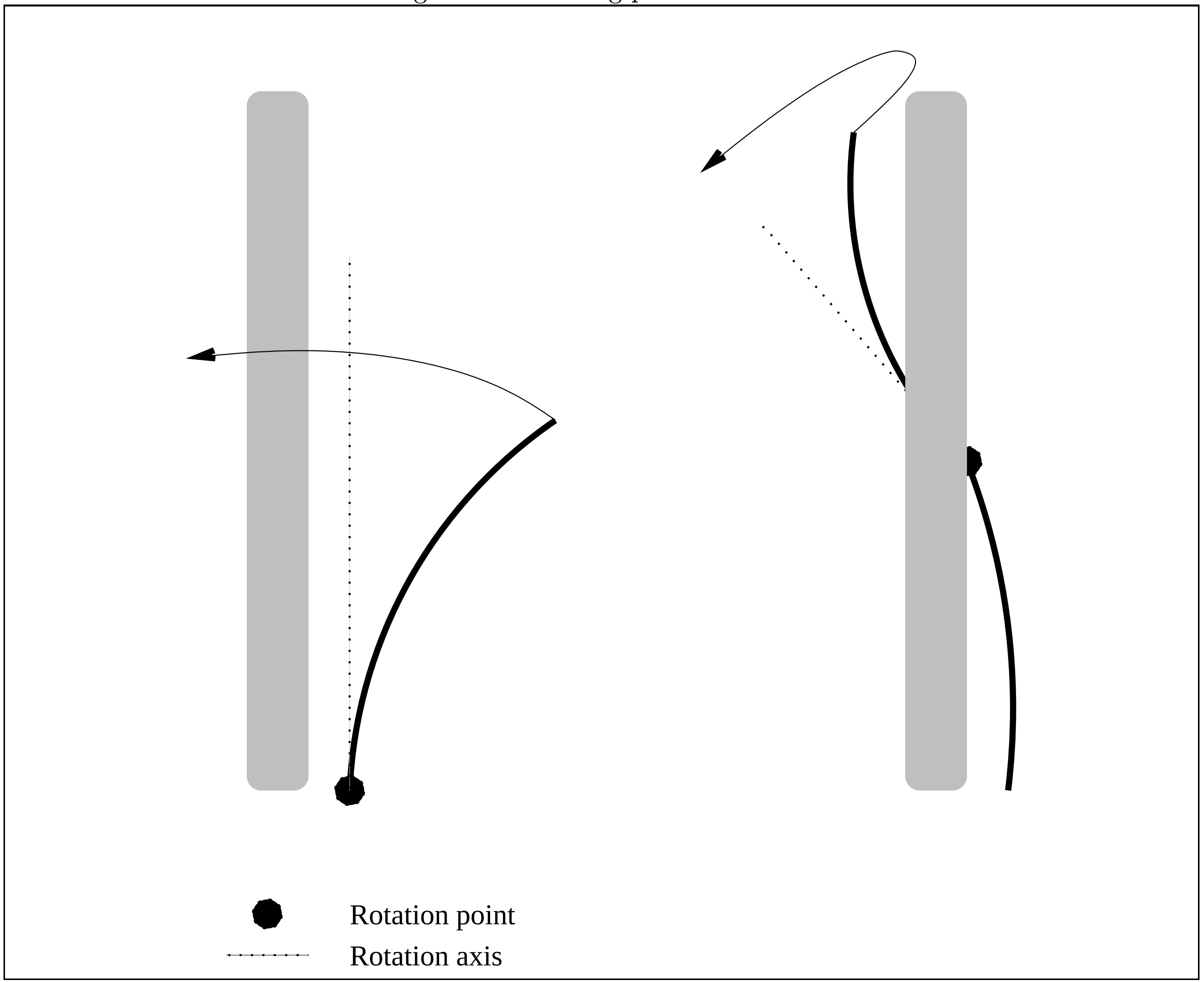
## 3.3 Searching movements

Many plants and plant organs show near-circular or elliptical movements when seen from above. This version of nutational movements is called *circumnutation*.

Twining plants perform wide "searching movements" that appear to be exaggerated circumnutations. See for instance [11]. When the stem hits a support, such as a tree branch or a vertical pole, it can end up winding around it in a helical pattern. [22]

Měch ([14], section 3.5.3) describes a twining plant simulation that is used as a basis for our model: The searching part of the stem is a rotating circle arc. When the stem collides with an object in the environment, a *contact point* is created, and the contact point becomes the new point of rotation. Originally the base of the stem is the point of rotation. Figure 3.3 illustrates this.

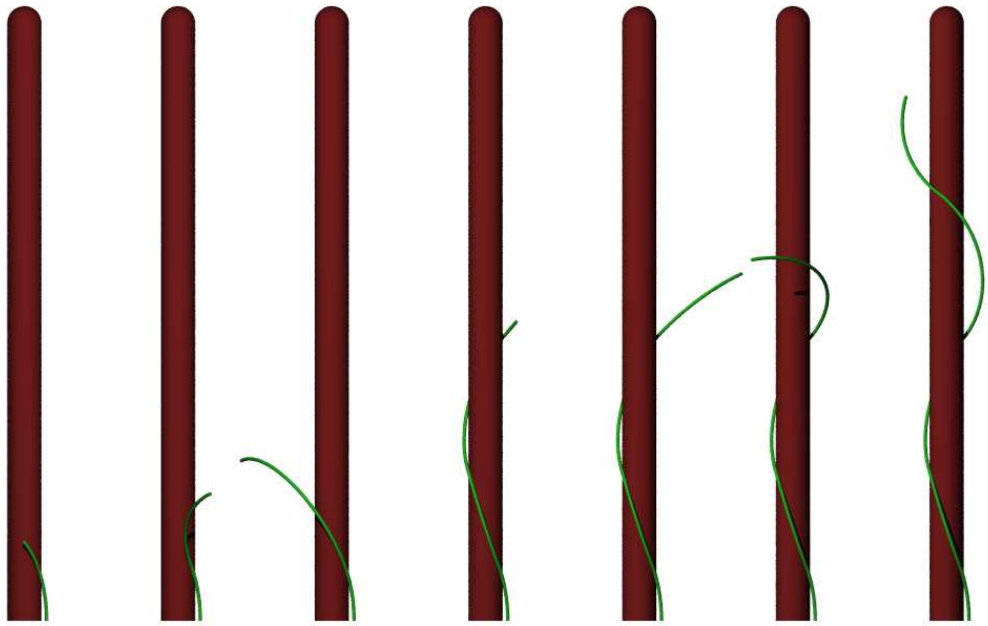
Figure 3.8: Twining plant schematic




The L-system in figure 3.9 is an implementation of this concept. The L-system toggles between two

Figure 3.9: Simple twining plant L-system

```
#define RSTEP 2 /* rotation step (degrees) */
#define ADEL 1  /* delay for apex growth */
#define PRAD 10 /* pole radius */
#define SRAD 1  /* segment radius */
#define in_obstacle(x,y,z) (x*x + z*z <= (PRAD+SRAD) * (PRAD+SRAD)
#define GROW 1  /* grow mode */
#define TEST 2  /* test mode */

Start:  mode = GROW; rotation_point = 0;
EndEach:  if (mode == GROW)  mode = TEST; else  mode = GROW;
Axiom: [!(PRAD)F(352)]!(SRAD)+(90)f(PRAD+SRAD+2)-(90)S(0,0)A(1,1)

A(ord,delay) : mode == GROW && delay > 0
  --> A(ord,delay-1)
A(ord,delay) : mode == GROW
  --> S(ord,0)?P(0,0,0,ord)A(ord+1,ADEL)
S(ord,rang)  : mode == GROW && ord == rotation_point
  --> S(ord,rang+RSTEP)
?P(x,y,z,ord) : mode == TEST && in_obstacle(x,y,z)
   rotation_point = ord;
  --> ?P(0,0,0,ord)

homomorphism
S(ord,rang) --> /(rang)+F
```



modes named “grow” and “test”. In grow mode, the stem both grows and rotates around the highest contact point. In test mode, environmental sensitivity is used to find the highest crashing point with the pole, and this point is set as the new rotation point.

The L-system is a simplification of an L-system created by Měch [14], the main differences being that Měch’s version contains a third mode called “backtrack” which after a collision with the stem rotates it backwards until the stem no longer collides with the pole. Also, leaves are created at fixed intervals.

## 3.4 Combining contact points and stem growth

Combining contact points (section 3.3) and stem elongation with an area of growth (section 3.1) is not completely trivial.

We want the segments of the stem to “flow” through a curve as outlined in section 3.2. We can write L-system productions that describe the growth of each segment on the stem without worrying about the speed at which the segment itself is moving compared to the root (base) of the stem. This is a *Lagrangian* description of the stem growth, which is easy to simulate by using turtle interpretation of L-system strings.

The problem is that the contact points cannot follow the flow of the stem; the stem distance from the root to each contact point must (in our model) be constant. We need to use Eulerian coordinates for the contact points and Lagrangian for the normal stem growth.

We create an L-system that demonstrates one way to solve this problem:

- Each segment is given as $A(s, p, c)$, where $s$ is the size of the segment, $p$ is the position of the segment (the distance from the root to the start of the segment), and $c$ is the position of a contact point within the segment, or $-1$ if it does not have a contact point.
- A global variable is used to keep track of segment positions.
- To make sure contact points do not follow the flow of the stem, context-sensitivity is used to move them between segments. As the segments flow towards the apex, the contact points will have to flow towards the root (or to the left) in the rewriting string.
- For simplicity, the stem is one-dimensional and contact points are created at fixed intervals.

The full L-system is given in figure 3.10. This is a fairly complex L-system that makes some assumptions about the underlying L-systems implementation:

- All variables are global.
- The rewriting string is processed sequentially from left to right. This allows us to use the variable `position` to keep track of each segment’s position on the “stem”.
- The productions are checked sequentially in the order they are listed, so that the variables calculated by the first production before the test expression can be used in subsequent productions.

Figure 3.10: Contact points and stem growth

```
#define GROWTH_RATE 1.01
#define CP_INTERVAL 5.0
#define MAX_LENGTH 1.0

Axiom: A(1,0,-1)X

Start: { next_contact_point = CP_INTERVAL; }
StartEach: { position = 0; this_position = 0; }

/* contact point exists and remains */
A(s,p,c)
    { /* calculate variables for all following productions */
      new_s = s * GROWTH_RATE;
      this_position = position;
      position = position + new_s; }
    : (c >= this_position)
    --> A(new_s, this_position, c)
/* contact point exists, but disappears (to the left) */
A(s,p,c) : (c > 0) && (c < this_position)
    --> A(new_s, this_position, -1)
/* new contact point from the right */
A(s,p,c) > A(sr,pr,cr) : (cr < position)
    --> A(new_s, this_position, cr)
/* new contact points at fixed intervals */
A(s,p,c) > X : (position > next_contact_point)
    { new_c = next_contact_point;
      next_contact_point = next_contact_point + CP_INTERVAL; }
    --> A(new_s, this_position, new_c)
/* no contact point */
A(s,p,c) --> A(new_s, this_position, c)

decomposition
A(s,p,c) : (s > MAX_LENGTH)
    { c1 = -1;
      c2 = -1;
      if (c >= 0) {
        if (c < p + s/2) { c1 = c; }
        else { c2 = c; }
      }
    }
    --> A(s/2, p, c1)A(s/2, p+s/2, c2)

homomorphism
A(s,p,c) : (c < 0) --> @O(3)F(s)
A(s,p,c) : (c >= 0) --> @O(3)F(c-p)@O(6)F(s+p-c)
```

Figure 3.11: Contact points

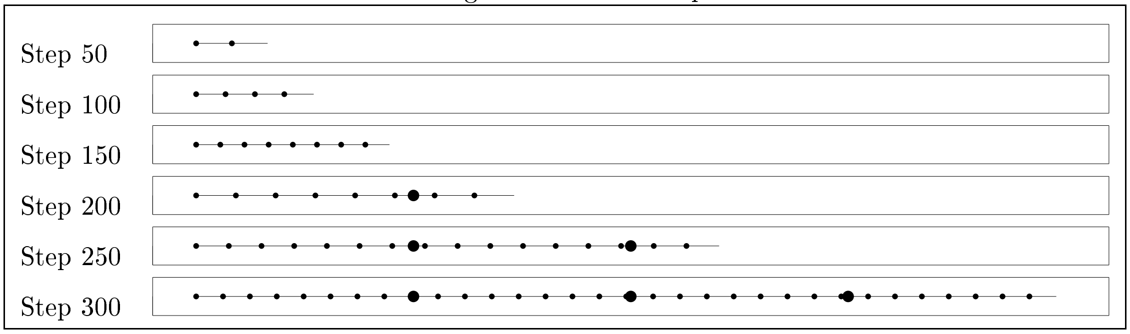



Figure 3.11 shows some images produced by the L-system. The segments are separated by small circles, and the big circles represent contact points. Circles are created by the turtle command `@O`. Note that the images are snapshots of an animation that would be better viewed in motion.

In this example, the segments grow at an exponential rate, and each of them split into two smaller segments when they reach a certain length, as seen by the decomposition production. Note that such splits have taken place between each pair of snapshots, except between steps 150 and 200, but this is not important. The important thing to notice is that the contact points (big circles) do not follow the flow of the segments.

## 3.5 The model

A twining plant model that combines many of the techniques from this chapter will now be described:

- Stem growth is described by an elongation function defined as a spline.
- The stem is shaped as a sequence of circle arcs between contact points. The arc curvature is constant for the whole stem.
- The last arc is rotating while it "searches" for support. Environmental sensitivity is used to check for support, and new contact points are created where the stem attaches.
- Contact points do not follow the flow of the stem: context-sensitivity is used to ensure this.
- Leaves are created near the apex and follow the stem's flow for a while.

The twining plant L-system is specified in figures 3.12, 3.13 and 3.14.

Since this model simulates the elongation of the stem, it is relatively easy to add leaves that follow the flow of the stem for a while.[4] As seen in figure 3.13, the apex `A(n)` produces the dummy symbol `X`

[4] In fact, from a visual point of view this is the whole point of the model; otherwise we could have just used the L-system in figure 3.9. But from a biological point of view the simulation of continuous growth might be interesting even without the visualization.

Figure 3.12: Twining plant L-system, part 1: definitions

```
#define TIMESTEPS_PER_HOUR 60
#define ROTATION_PER_HOUR 180
#define TIME_BETWEEN_LEAVES 5.0
#define MAX_LENGTH 2
#define GROW_AREA 20
#define SPLINE_FUNC(x) (...)
#define GROW(s,p,length) ((length-p > GROW_AREA) ? s : \
        s + s*SPLINE_FUNC((length-p)/GROW_AREA)/TIMESTEPS_PER_HOUR)
#define CURVATURE 2
#define ROTATION_STEP (ROTATION_PER_HOUR/TIMESTEPS_PER_HOUR)
#define TEST_MODE 0
#define GROW_MODE 1
#define POLE_RADIUS 1.5
#define STEM_RADIUS 0.5
#define CRASH(x,y,z) (...)

Start: {
  mode = GROW_MODE;
  rotation_point = 0;
  new_rotation_point = 0;
  pos = MAX_LENGTH;
}
StartEach: {
  length = position;
  this_position = 0;
  position = 0;
  new_s = 0;
}
EndEach: {
  if (mode == TEST_MODE) {
    mode = GROW_MODE;
  } else {
    mode = TEST_MODE;
  }
}

Consider: T
Axiom: !(SRAD*2)T(MAX_LENGTH,0,0,0)A(0)
```

Figure 3.13: Twining plant L-system, part 2: productions

```
/*** GROW_MODE ***/
/* contact point disappears to the left */
T(s,p,c,ra) :
    { new_s = GROW(s,p,length);
      this_position = position;
      position = position + new_s;
    }
    mode == GROW_MODE && c >= 0 && c < this_position
    --> T(new_s, this_position, -1, -1)
/* contact point remains */
T(s,p,c,ra) :
    mode == GROW_MODE && c >= 0
    { if (c == rotation_point) {
        new_ra = ra+ROTATION_STEP;
      } else {
        new_ra = ra; }}
    --> T(new_r, this_position, c, new_ra)
/* new contact point from the right */
T(s,p,c,ra) > T(s2,p2,c2,ra2) :
    mode == GROW_MODE && c2 >= 0 && c2<position
    { if (c2 == rotation_point) {
        new_ra = ra2 + ROTATION_STEP;
      } else {
        new_ra = ra2; }}
    --> T(new_s, this_position, c2, new_ra)
/* just growing */
T(s,p,c,ra) :
    mode == GROW_MODE
    --> T(new_s, this_position, c, ra)
/* leaves */
A(n) : mode == GROW_MODE && n>=TIME_BETWEEN_LEAVES --> XA(0)
A(n) : mode == GROW_MODE --> A(n+(1/TIMESTEPS_PER_HOUR))
X    : mode == GROW_MODE --> *
?P(x,y,z) > T(s,p,c,ra)X : mode == GROW_MODE --> ?P(x,y,z)L(0,0)
/* L(size,rotation) --> ... */ /* production not shown here */

/*** TEST_MODE ***/
/* ignore tests lower than rotation_point + MAX_LENGTH */
?P(x,y,z) > T(s,p,c,ra) :
    mode == TEST_MODE && p < rotation_point + (MAX_LENGTH * 1.1)
    --> ?P(0,0,0)
/* choose the highest crashing point as the new rotation point */
?P(x,y,z) > T(s,p,c,ra) :
    mode == TEST_MODE && CRASH(x,y,z)
    { new_rotation_point = p; }
    --> ?P(0,0,0)
```

Figure 3.14: Twining plant L-system, part 3: decomposition and homomorphism

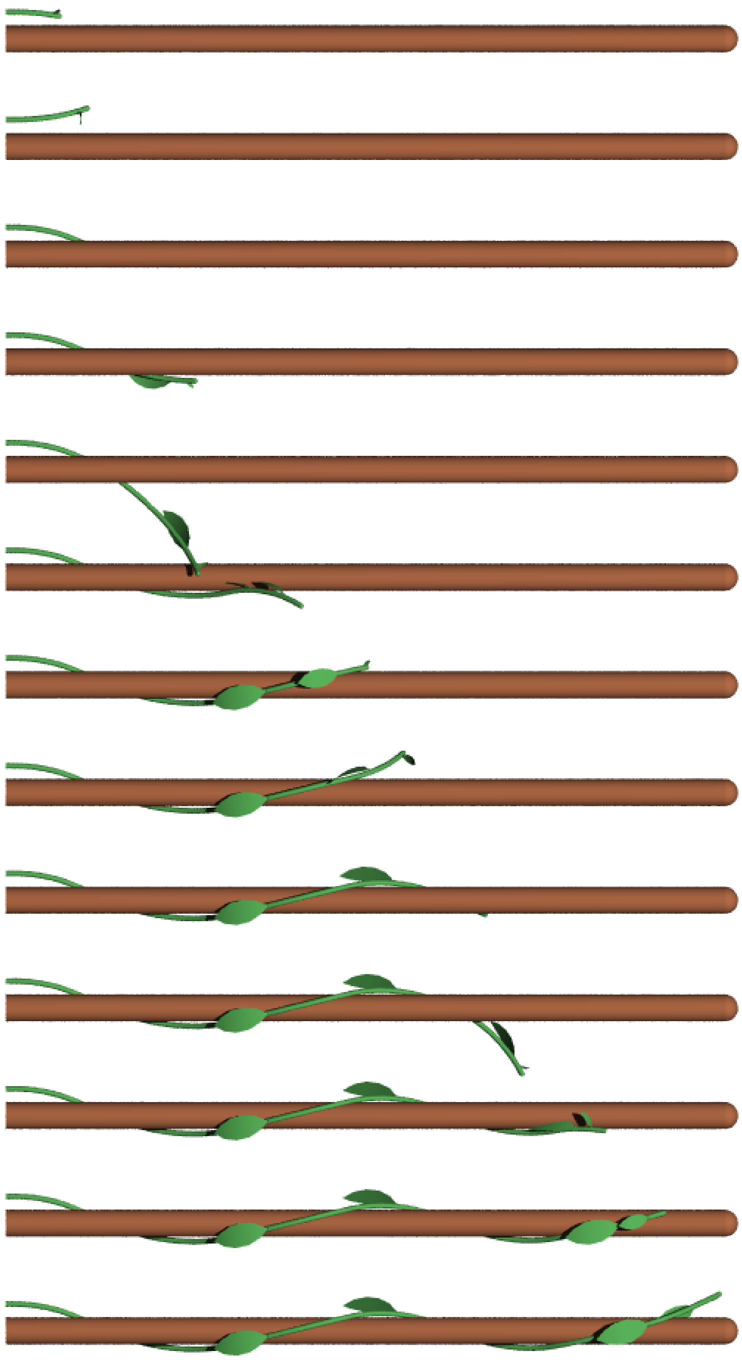
```
decomposition

T(s,p,c,ra) : s > MAX_LENGTH && c < 0
    --> T(s/2,p,c,ra)?P(0,0,0)T(s/2,p+s/2,c,ra)
T(s,p,c,ra) : s > MAX_LENGTH && c < p+s/2
    --> T(s/2,p,c,ra)?P(0,0,0)T(s/2,p+s/2,-1,-1)
T(s,p,c,ra) : s > MAX_LENGTH
    --> T(s/2,p,-1,-1)?P(0,0,0)T(s/2,p+s/2,c,ra)

homomorphism

T(s,p,c,ra) : c < 0
    --> +(CURVATURE*s)F(s)+(CURVATURE*s)
T(s,p,c,ra) : c >= 0
    { left_angle = CURVATURE*(c-p);
      right_angle = CURVATURE*(s+p-c); }
    --> +(left_angle)F(c-p)+(left_angle)/(ra)
        +(right_angle)F(s+p-c)+(right_angle)
/* L(size,rotation) --> ... */
/* homomorphism productions for leaves are not shown */
```

at fixed time intervals. This symbol disappears during the next growth step, and is replaced by `L(0,0)` one segment from the apex, which is a representation of a leaf.

The model simulates the curving and elongation of the stem fairly well, but no attempt has been made to accurately model the *torsion* (twist) of the stem. In fact, the stem torsion is exactly zero except at the contact points, where it is nonexistent (or infinite). Visually, this means that each leaf will not follow the stem's twist as well as it follows its elongation.

Exactly how a leaf grows and is rotated around the stem was not examined in detail, but implemented in an "ad hoc" manner, and so further leaf productions are not shown in the L-system. Homomorphism productions for leaves are also omitted. The important aspect is how the flow of the stem can be seen as each leaf follows the growth for a while, eventually ending up in a fixed position from the stem's base, as shown in figure 3.15.

Figure 3.15: Twining plant



# Chapter 4

# L-Lisp: L-systems in Common Lisp

## 4.1 L-systems and programming

Section 1.10 introduced some programming extensions to the L-systems language. Experience shows that the more complex L-system models get, the more extensions are needed. More support for functions, data structures and other programming concepts would be useful. Ideally our L-systems program should have the power of a real programming language.

It is possible to do this by adding more and more support for the needed extensions until we have a programming language built around the concept of L-systems. Another option is to create an L-systems framework *within* an existing programming language. This chapter describes L-Lisp, an L-systems framework within Common Lisp. L-Lisp was created by the author.

This chapter assumes some knowledge of Common Lisp as well as L-systems. For good introductory books on Common Lisp, see [10, 15].

## 4.2 Related work

The *cpfg* plant modeling program which is being developed by Prusinkiewicz and others at the University of Calgary features a powerful special-purpose programming language based upon L-systems. [20]

Goel and Rozehnal developed a programming language which is general-purpose with built-in support for L-systems and graphical applications. [8]

Borovikov presented a language with support for L-systems with inheritance, an object-oriented version of L-systems. [2]

L-system implementations within existing programming languages seem to be rare, but Prusinkiewicz and Hanan describe an approach for expressing context-free L-systems in C. [18]

## 4.3 Advantages and disadvantages

Using an existing programming language to create L-systems has some disadvantages:

- The language's syntax is not made especially for L-systems, so it will tend to get more verbose than a pure L-systems language, except possibly for complex L-systems[1].
- Users have to learn the language, or at least a subset of it.
- Error messages from the compiler will not be L-system-specific, and therefore potentially less informative than what you would get from an L-systems program.

But there are also many advantages:

- A full programming language.
- Flexibility; users can do things that just wouldn't work within the constraints of normal L-system syntax.
- Extensibility; adding new (experimental) features is relatively easy.

Working within a programming language could be an attractive choice for researchers who want to try out experimental new features without having to deal with the complexities of an existing implementation, such as parsing and compilation.

## 4.4 Why Common Lisp?

The simple answer is, to quote Peter Norvig [15]: "Lisp makes it easy to define new languages especially targeted to the problem at hand."

This is true because Lisp allows Lisp code to be manipulated easily. The Common Lisp dialect has a very powerful macro facility; a macro is just a special kind of function that returns transformed Lisp code. You might go as far as saying that for experienced Common Lisp programmers, writing new languages is trivial. A good book on advanced macro techniques is Paul Graham's *On Lisp* [9].

Macros turn out to be very useful for simplifying the syntax of L-system productions, especially parametric ones.

One of the reasons to create an L-systems framework, is to end up with something more flexible than a normal L-systems program. Lisp provides a lot of flexibility, not least because of its dynamic type system. This allows the parameters of L-system modules to be of any type, not just numerical ones. A parameter can be any kind of object or data structure.

[1] The last part of this statement might need an explanation; why would a framework within an existing language handle complex L-systems better than a special-purpose language for L-systems? The reason is that general programming techniques are often needed to solve complex problems, and *most* special-purpose languages do not support such general techniques particularly well.

Lisp is also interactive, which means that programming and compiling usually happens within the run-time Lisp environment. This eliminates the need to quit, save, recompile and restart every time you make a change to an L-system.

## 4.5 Design decisions

A short overview of some of the design decisions:

### 4.5.1 Syntax

Common Lisp allows the programmer to change the Lisp syntax quite easily using a facility called "read macros". However, since the goal is not to create an entire L-system language with its own syntax, but rather provide a framework, add some new operators, and yet keep the flexibility and freedom of Common Lisp, everything in L-Lisp uses normal Lisp syntax. In other words, L-Lisp uses parenthesized lists called *symbolic expressions*. For instance, a parametric L-system module such as $A(x, y)$ will be written as `(A x y)` in Lisp syntax.

### 4.5.2 Classes and methods

The implementation uses CLOS (the Common Lisp Object System, the object-oriented part of Common Lisp); a base class `l-system` provides methods for rewriting, context checking and so on, and L-systems are created by defining subclasses.[2]

Sets of productions should be defined in the methods `l-productions`, `homomorphism` and `decomposition`.

### 4.5.3 Macros

Macros are provided for simplifying productions and context-checking. It would be possible to create macros that hide the fact that CLOS is used, or even encapsulate the whole L-system declaration in one big `def-l-system` macro. This has not been done because it would make it harder to add new features, and basically move us further away from Lisp.

The rule-of-thumb for L-Lisp is that macros are used to simplify, *not* to hide details.

### 4.5.4 Turtle commands

Turtle commands are usually given the same names as used throughout this text, only in Lisp syntax. $+(x)$ is written as `(+ x)`, for instance. There are a few problems though:

[2]CLOS programmers might complain that this description is incorrect since methods do not belong to a class in CLOS. Which is true, but of little importance in this case.

- The Common Lisp reader by default converts all characters to upper case, so there is no difference between `f` and `F`. Adding a backspace before lower case characters works: `\f`.
- Some turtle commands have a special meaning in Common Lisp, such as `.` or `|`. Again, adding a backspace before the command works: `\.` or `\|`.
- Some turtle commands look exactly like numerical functions in Lisp syntax, and a single expression may contain both uses. For instance, $+(x+y)$ will look like `(+ (+ x y))`, which can be confusing to read for humans.

Because of these problems, most predefined turtle commands have both a short and a long name. For instance, `+` has the long name `:turn-left`, so $+(x+y)$ can be written as `(+ (+ x y))` or `(:turn-left (+ x y))`. In some cases the long name will be the more readable, especially for complicated expressions, but it is ultimately a matter of personal style.

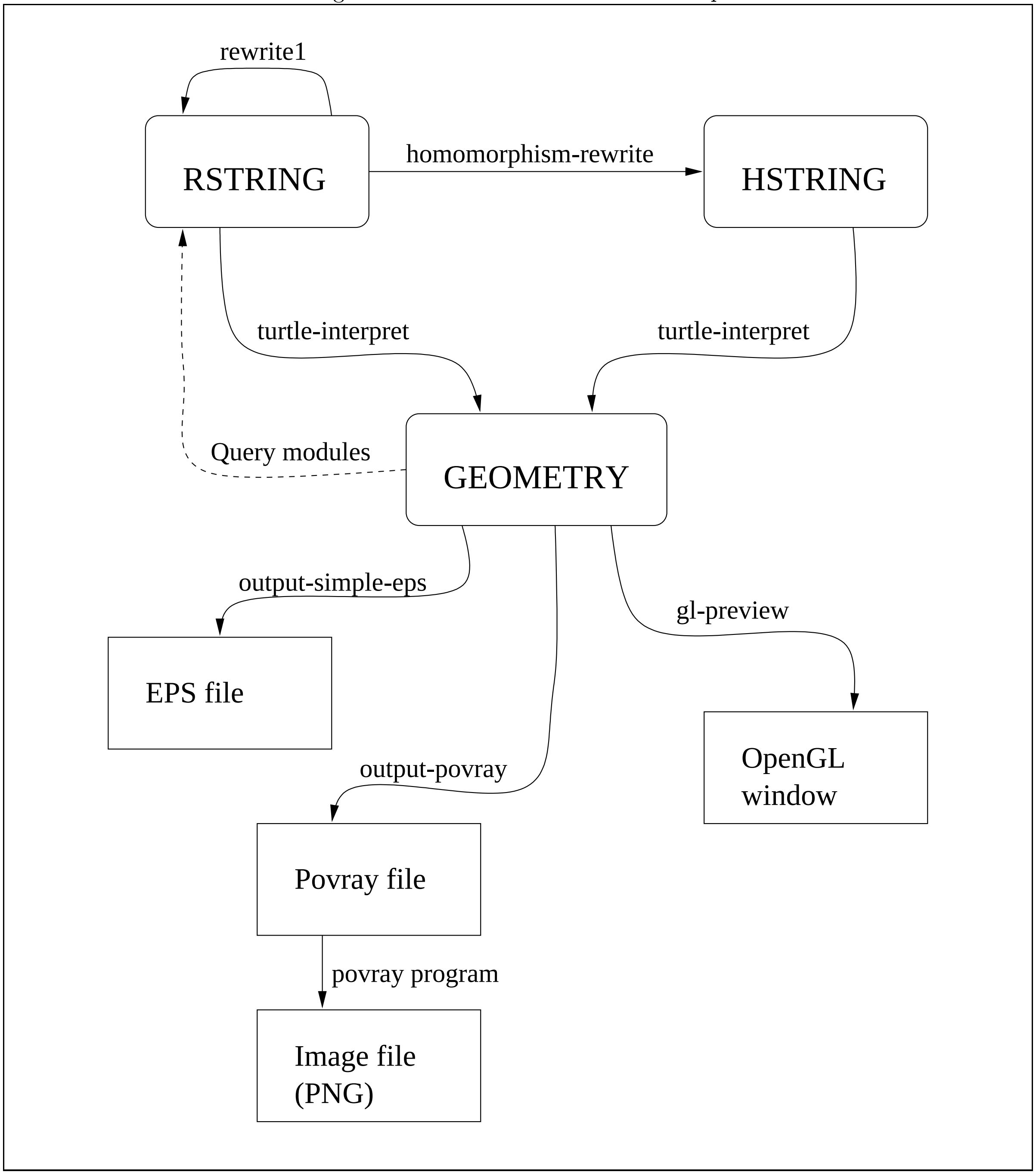
## 4.6 Introduction

This section gives a tutorial-style introduction to some of the features of the L-Lisp framework.

Before starting the tutorial, it is useful to get an understanding of the data flow in L-Lisp. For a schematic overview, see figure 4.1, which can be described as follows:

- The rewriting string is stored in a slot called `rstring`, which is updated each time the function `rewrite1` is called.
- The rewriting string can then be converted via turtle interpretation to a set of geometric shapes (such as lines) which is stored in the `geometry` slot.
- For L-systems with homomorphism (section 1.8), the rewriting string is converted by the function `homomorphism-rewrite` and stored in the `hstring` slot, which is used instead of `rstring` during turtle interpretation.
- Finally, the geometry can be visualized by creating:
  - EPS (Postscript) files, which can be used for creating good printable black-and-white images.
  - Povray (Persistence of Vision ray-tracer) files, which can produce 3D images with realistic colors, shadows and other effects.
  - OpenGL previews, which is fast and allows models to be interactively rotated and resized. Creating real-time animations of growing models is also possible.

Figure 4.1: Schematic overview of L-Lisp



### 4.6.1 Creating a simple L-system in L-Lisp

In this section we will use the Fibonacci L-system (page 2) as an example. In normal L-system syntax, it is written like this:

$$\begin{aligned} \omega &: a \\ a &\rightarrow b \\ b &\rightarrow ab \end{aligned}$$

In L-Lisp, L-systems are defined as subclasses of `l-system`:

```
(defclass fibonacci (l-system)
  ((axiom :initform '(a))
   (depth :initform 4)))
```

The axiom must be a sequence of modules, in this case it is a list of a single module `a`. The depth is the default number of rewrites to be performed.

L-system productions are defined in the method `l-productions`, which we *could* define like this:

```
(defmethod l-productions ((ls fibonacci))
  (case (current-module ls)
    (a (list 'b))
    (b (list 'a 'b))))
```

The method looks at the current module, and returns a list of modules to replace it. We could also define the method like this:

```
(defmethod l-productions ((ls fibonacci))
  (choose-production ls
    (a (--> b))
    (b (--> a b))))
```

This looks very similar to the above definition, except that the `case` statement has been replaced with a `choose-production` statement, `list` with `-->` and the modules are not quoted. `choose-production` and `-->` are macros that in this case may seem unnecessary, but in the case of more complex productions will simplify things a *lot*.

Now we can create an instance of the L-system and do some rewriting (the '>' is the Lisp prompt[3]):

```
> (setf x (make-instance 'fibonacci))
#<FIBONACCI 753C755>
> (rewrite x)
#(A B B A B)
```

The `rewrite` function does the default number of rewrites, as defined by the `depth` slot (in this case 4). We can do a different number of rewrites by supplying a second parameter:

```
> (rewrite x 6)
#(A B B A B B A B A B B A B)
```

[3]Common Lisp is an interactive language which includes a so-called read-eval-print loop which reads an expression, evaluates it, then prints the result.

Or we can do one rewrite at a time by calling the `rewrite1` function:

```
> (rewrite x 0)
#(A)
> (rewrite1 x)
#(B)
> (rewrite1 x)
#(A B)
> (rewrite1 x)
#(B A B)
> (rewrite1 x)
#(A B B A B)
> (dotimes (i 2) (rewrite1 x))
NIL
> (rstring x)
#(A B B A B B A B A B B A B)
```

Note that the rewriting string is represented by a vector of modules, as opposed to a Lisp string (which is a specialized vector of characters). It is stored in the `rstring` slot.

Also note that `(rewrite x 0)` initializes the rewriting string with the axiom, but does no rewriting.

### 4.6.2 The snowflake curve

We will use a simple fractal generated by an L-system, the snowflake curve (page 3), as an example. It can be defined like this in Lisp:

```
(defclass snowflake (l-system)
  ((axiom :initform '(F - - F - - F))
   (depth :initform 1)
   (angle-increment :initform 60.0)))

(defmethod l-productions ((ls snowflake))
  (choose-production ls
    (F (--> F + F - - F + F))))
```

To create a postscript (EPS) image step-by-step:

```
> (setf x (make-instance 'snowflake))
#<SNOWFLAKE F96CDE5>
> (rewrite x 3)
#(F + F - - F + F + F + F ...)
> (create-geometry x)
#(#S(LINE ...) ...)
> (output-simple-eps (geometry x) "snowflake.eps")
NIL
```

The exact output from the Lisp toplevel may be different, especially if it shows the whole vectors.

We create an instance of `snowflake` and rewrite it, then we call `create-geometry`, which passes the rewriting string (along with the angle increment defined in the snowflake class) on to a turtle interpreting function, which returns the snowflake geometry (a vector of geometric shapes, in this case lines). Finally we call `output-simple-eps` to create a postscript file.

Normally we won't go through all these steps. Instead we will just call `(rewrite-and-preview x "snowflake.eps" :depth 3)`, which rewrites, creates the geometry, outputs a postscript file, and finally previews the file by calling the ghostview program. Figure 4.2 shows an example of how an L-Lisp session might look; this setup uses Emacs with ILISP and CMU Common Lisp.

On some systems we can also call `(gl-preview x :depth 3)`, which also rewrites the L-system and creates the geometry, and then shows an OpenGL preview of the geometry. This is typically much faster than `rewrite-and-preview`.

### 4.6.3 A parametric L-system

As an example of a parametric L-system, we use the parametric version of the snowflake curve (page 8).

In normal L-system syntax, it can be written as

$$
\begin{array}{lcl}
\omega & : & \texttt{F(1)-(120)F(1)-(120)F(1)} \\
\texttt{F(x)} & \rightarrow & \texttt{F(x/3)+(60)F(x/3)-(120)F(x/3)+(60)F(x/3)}
\end{array}
$$

In L-Lisp, we can define it like this:

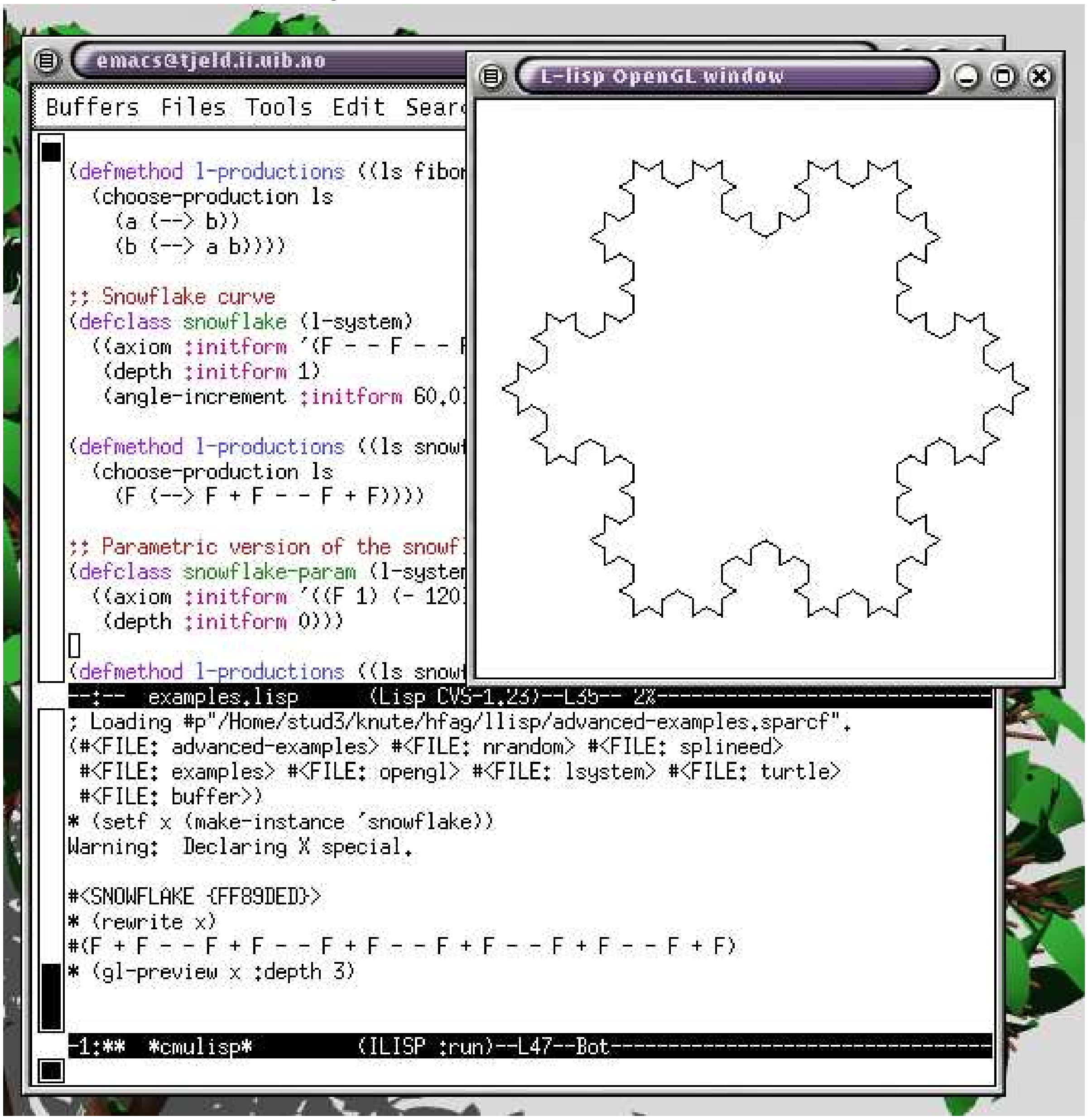
```
(defclass snowflake-param (l-system)
  ((axiom :initform '((F 1) (- 120) (F 1) (- 120) (F 1)))
   (depth :initform 1)))

(defmethod l-productions ((ls snowflake-param))
  (choose-production ls
    ((F x)
     (let ((newx (/ x 3)))
       (--> (F newx) (+ 60) (F newx) (- 120) (F newx)
            (+ 60) (F newx))))))
```

The current module matches `(F x)` only if it is a parametric module with `F` as the first element and has exactly one parameter. The `choose-production` macro checks all this, and also binds the variable `x` to the parameter. If no module match is found, it returns `t`, which symbolized the identity production.

The `-->` macro builds a list from its input, treating the first element of each parameter as a symbol and the rest as expressions. For instance, the expression `(let ((x 3)) (--> (x x) (x x (* x 2))))` will return the nested list `((X 3) (X 3 6))`.

It we were to write the production without these macros, we could use something like this:

Figure 4.2: An L-Lisp session in GNU Emacs



```
(defmethod l-productions ((ls snowflake-param))
  (let ((module (current-module ls)))
     (if (and (consp module) (eql (first module) 'F)
              (= (length module) 2))
       (let* ((x (second module))
              (newx (/ x 3)))
         `((F ,newx) (+ 60) (F ,newx) (- 120) (F ,newx)
           (+ 60) (F ,newx)))
       t)))
```

Even this relatively easy production is quite complex without the production macros. In this particular case we could simplify it a bit by omitting the length test, but still the first version would be much clearer.

## 4.7 User's guide

### 4.7.1 The `l-system` class

`l-system` is the base class for L-systems. There is normally no reason to create instances of it. L-systems are created as subclasses of `l-system`.

Figure 4.3 gives an overview of the `l-system` slots. All slots have accessors with the same name as the slot.

### 4.7.2 Rewriting

The basic idea behind rewriting is as follows: For each module in the old rewriting string, set the `current-module` slot, then call the `l-productions` method, defined by the user. `l-productions` looks at the current module (and possibly some other variables), and returns either a list of modules, which are added to the new rewriting string, or `t`, which means that the current module is added to the new rewriting string.

The above is implemented in the generic function `rewrite1`:

```
(rewrite1 lsystem)
```

The function `rewrite` is used for performing several rewrites at once:

```
(rewrite lsystem &optional depth)
```

`rewrite` will initialize the rewriting string with the axiom, then call `rewrite1` repeatedly. This means that in order to initialize the rewriting string without performing any rewrites, one can call `(rewrite lsystem 0)`.

Figure 4.3: The `l-system` class

Basic slots

| Slot | Default | Description |
|---|---|---|
| `axiom` | `nil` | Sequence of modules. |
| `depth` | 0 | Default number of rewrites. |
| `angle-increment` | 90.0 | Default angle increment; passed along to turtle. |
| `ignore-list` | `nil` | List of symbols to ignore in context. |
| `consider-list` | `nil` | List of symbols to consider in context. Overrides `ignore-list`. |
| `homomorphism-depth` | 0 | Max depth of homomorphism. If zero, homomorphism will be ignored (default). |
| `decomposition-depth` | 0 | Max depth of decomposition. If zero, decomposition will be ignored (default). |
| `sensitive` | `t` | Must be set to `t` for environmentally sensitive L-systems. Setting it to `nil` can result in faster rewriting. |

Graphical view/animation parameters (for OpenGL)

| Slot | Default | Description |
|---|---|---|
| `line-style` | `:lines` | Either `:lines` or `:cylinders`. |
| `cylinder-width` | `1.0` | Width multiplier for cylinders. |
| `limits` | `nil` | List of two position sequences `((x1 y1 z1) (x2 y2 z2))` that override the default values for the size and position of the geometry. This is useful for animations that grow in size. |
| `frame-delay` | 0.5 | Delay between animation frames, in seconds. |
| `frame-list` | nil | List of frames and frame intervals (nested lists) for animations. For details, see section 4.7.8, page 73. |

Slots that should not be set explicitly

| Slot | Default | Description |
|---|---|---|
| `rstring` | `nil` | Rewriting string; vector of modules. |
| `hstring` | `nil` | Homomorphism rewriting string; vector of modules. |
| `geometry` | `nil` | 3D geometry created by turtle interpretation; vector of geometric structures. |
| `current-depth` | `nil` | The number of the current/latest rewrite. |

Slots for internal use

| Slot | Default | Description |
|---|---|---|
| `current-module` | `nil` | Used during rewriting. |
| `pos` | 0 | Used during context checking. |
| `context-pos` | 0 | Used during context checking. |
| `warning-msg` | `nil` | Used to keep track of warnings. |

#### Auxiliary methods

CLOS supports a facility called *auxiliary methods* that allows code to be called before, after, or around specific methods. Since `rewrite` and `rewrite1` are generic functions, auxiliary methods can be defined for subclasses. Before- and after-methods for `rewrite` is the equivalent of *Start-* and *End*-blocks. before- and after-methods for `rewrite1` is the equivalent of *StartEach* and *EndEach*.

If a class `foo` is a subclass of `l-system`, then before- and after-methods can be defined as:

```
(defmethod rewrite :before ((ls foo) &optional (depth 0))
  ...)

(defmethod rewrite :after ((ls foo) &optional (depth 0))
  ...)

(defmethod rewrite1 :before ((ls foo))
  ...)

(defmethod rewrite1 :after ((ls foo))
  ...)
```

Note that you need to supply the optional parameter to `rewrite` even if you will ignore it.

### 4.7.3 Defining productions

L-system productions are defined in the method `l-productions`, which should be created for subclasses of `l-system`.

The only requirements for the method is that it must return either a list of modules, or `t`. The method will be called once for each module in the rewriting string; if the return value is a list of modules, the modules will be appended to the next rewriting string[4], and if `t` is returned, the current module will be added to the new rewriting string.

There are almost no restrictions for how to produce the return value within `l-productions`, but the recommended way is to use the forms `-->` and `choose-production`, described below.

#### The `-->` macro

The `-->` macro is used for producing lists of modules, and has the following form:

(`-->` $module_1$ ... $module_n$)

Each module must be either a symbol or a list of the form

($symbol$ $expr_1$ ... $expr_n$)

in which case the symbol will not be evaluated, while the succeeding expressions will be.

The return value is a list of modules, where each module is either a symbol or a list whose first element is a symbol (representing a parametric module).

[4] Except for modules that equal `nil`, which will be ignored.

**The `choose-production` macro**

The `choose-production` macro has the following form:

```
(choose-production lsystem (module1 exprs1) ... (modulen exprsn))
```

Each of the modules must be either a symbol or list of symbols. A symbol means a module without parameters, a list means a module with parameters. The parameter names of $module_i$ can be used in $expr_i$.

The macro returns the first expression whose

1. module-form matches the current module. The name and number of parameters must be the same.

2. expression does not return `nil`.

If none of the productions match, it returns `t`, which is interpreted as the identity production.

Note that it is not possible to return `nil` from within the macro (unless you use `return-from` or similar tricks), since this means "the production did not match, continue searching". If you need an erasing production, use `(--> nil)` (or `(list nil)`, or `'(nil)`, which is the same).

The macro does a few things in addition to matching modules and binding parameter variables:

- The set of productions is surrounded by a block, and every `-->` expression is returned from this block. In other words, a `(--> ...)` expression is guaranteed to return from the surrounding `choose-production` expression.

- Every `with-left-context`, `with-right-context`, `with-lc` and `with-rc` expression within the macro gets the L-system added as its first parameter. These expressions will be explained in section 4.7.4.

### 4.7.4 Context checking

In normal L-systems syntax a set of context-sensitive productions could be written as

$$\begin{array}{ccccccc} A & < & X & > & B & \rightarrow & Y_1 \\ A & < & X & > & CD & \rightarrow & Y_2 \\ A & < & X & & & \rightarrow & Y_3 \\ & & X & & & \rightarrow & Y_4 \end{array}$$

In L-Lisp, the same set of productions could be written as

```
(choose-production ls
  (X (with-left-context (A)
       (with-right-context (B) (--> Y1))
       (with-right-context (C D) (--> Y2))
       (--> Y3))
     (--> Y4)))
```

Ignoring the implementation details for now, what happens can be described like this:

```
if <current module matches X> then
    if <left context matches A> then
        if <right context matches B> then
            return Y1
        else if <right context matches C D> then
            return Y2
        else
            return Y3
    else
        return Y4
```

What looks like four separate productions in normal L-system syntax, is expressed more efficiently (although not necessarily more clearly) as *one* production with several context checks.

A parametric, context-sensitive production

$$A(x) < B(y) > C(z) \rightarrow Y(x + y + z)$$

can be expressed in L-Lisp like this:

```
(choose-production ls
  ((B y) (with-lc ((A x))
           (with-rc ((C z))
             (--> (Y (+ x y z)))))))
```

Here `with-lc` and `with-rc` are short forms of the `with-left-context` and `with-right-context` macros. They do exactly the same.

This example shows that the context-macros can bind variables in much the same way as `choose-production`.

The `with`-forms for context checking can only be used within a `choose-production` expression[5], and are given as:

```
(with-left-context (module1 ... modulen) exprs)
(with-right-context (module1 ... modulen) exprs)
(with-lc (module1 ... modulen) exprs)
(with-rc (module1 ... modulen) exprs)
```

As in `choose-production`, each module can be a symbol or a parametric module. The body is evaluated if and only if the left (or right) context matches the listed modules. Module parameters can be used within the body.

The slots `ignore-list` and `consider-list` define modules that will be ignored or considered in the context. See figure 4.3 for details.

[5] Or to be specific, the syntax is different outside of `choose-production` (see page 89).

**Dynamic ignore and consider**

In traditional L-system programs, the symbols that should be ignored or considered during context checks cannot be changed during the rewriting process. In L-Lisp, `ignore-list` and `consider-list` are slots that can be changed at any time. This turns out to be very useful in some cases; typically there are cases when it is convenient to ignore one kind of module most of the time, but we need to check for it in a few places. Working around this limitation can be a time-consuming task.

Setting the slots explicitly using for instance `(setf (ignore-list ls) ...)` is possible, but usually not recommended, since keeping track of what the `ignore-list` is at any time can be difficult. Instead, the macros `while-ignoring` and `while-considering` should be used:

```
(while-ignoring lsystem ignore-list
  ...)

(while-considering lsystem consider-list
  ...)
```

These macros change the slots temporarily, and guarantee that the old values are restored afterwards.[6] The expansion of these macros are pretty straightforward, and not described further here.

### 4.7.5 Cut symbol

L-Lisp supports cutting off branches using a *cut symbol* (see section 1.11), and like the turtle functions it has both a short and a long name; `%` and `:cut-symbol`.

The cut symbol is used exactly like a turtle command, except that it cannot take parameters.

### 4.7.6 Homomorphism and decomposition

In addition to `l-productions`, two other sets of productions named `homomorphism` and `decomposition` can be defined. See sections 1.8 and 1.9 for details on these L-system extensions.

These methods are defined in the same way as `l-productions`, except that context checking cannot be used. The use of `-->` and `choose-production` is recommended.

Since both homomorphism and decomposition rewriting are recursive, steps need to be taken to prevent infinite recursion. The L-system class has two slots `homomorphism-depth` and `decomposition-depth` that defines the maximum recursion depth of the two rewriting functions. If either of these values are zero (the default value), homomorphism or decomposition will not be performed at all.

### 4.7.7 The `stochastic-choice` macro

In section 1.7, we defined a set of L-system productions like this:

[6] This is guaranteed even if the block is exited by a `return-from`, `throw` or even an error. The special form `unwind-protect` ensures this; consult a Common Lisp reference for further details.

$$
\begin{array}{lcl}
\mathtt{F} & \rightarrow & \mathtt{F} & : & 2 \\
\mathtt{F} & \rightarrow & \mathtt{F[-F]} & : & 1 \\
\mathtt{F} & \rightarrow & \mathtt{F[+F]} & : & 1
\end{array}
$$

Notice that the left-hand side of each production is the same. In a way similar to what we did for context-checking, we will express this as *one* production which chooses its result randomly based upon the given random distribution.

Common Lisp has a function `random`; `(random` $x$`)` returns a pseudorandom number $y$ such that $0 \leq y < x$. Using this, we could define the productions like

```
(choose-production ls
  (F (let ((r (random 4)))
       (cond ((< r 2) (--> F))
             ((< r 3) (--> F [ - F ]))
             (t (--> F [ + F ]))))))
```

We generate one random number (integer to be precise), and choose one of the "`-->`"-expressions based on the result. In L-Lisp, there is a macro called `stochastic-choice` to simplify this, so we could more easily define the productions like this:

```
(choose-production ls
  (F (stochastic-choice
       (2 (--> F))
       (1 (--> F [ - F ]))
       (1 (--> F [ + F ])))))
```

The general form for the macro is

```
(stochastic-choice
  (weight_1 exprs_1)
  ...
  (weight_n exprs_n))
```

A few things to note about the macro:

- The weights do not have to be constants.
- The weights can be either integers or floats, but not ratios.
- The macro can in fact appear anywhere, and is not specific to L-systems.[7]

### 4.7.8 Creating images

The basis for all image generation in L-Lisp is the geometry vector stored in the `geometry` slot, which is filled by the function `create-geometry`:

[7]Choosing randomly between expressions is not a particularly common programming technique, but it can be used in some randomized algorithms.

```
(create-geometry lsystem)
```

which will do the necessary homomorphism rewriting and turtle interpretation to create the geometry vector. More details on turtle interpretation will be given in section 4.7.9.

`create-geometry` is usually not called directly. Instead, images are created directly by methods like `rewrite-and-preview`, `rewrite-and-raytrace` or `gl-preview`:

Postscript graphics are generated by the function `rewrite-and-preview`, which generates an EPS file, then previews it using Ghostview. Note that some of the geometry data is ignored when generating Postscript images, such as colors and polygon meshes.

```
(rewrite-and-preview lsystem filename
    &key depth width sphere-width)
```

**depth** : Number of rewrites.

**width** : Line width multiplier.

**sphere-width** : Width multiplier for spheres/points.

Ray-traced images can be created by the function `rewrite-and-raytrace`, which generates a Povray [26] file and launches the ray-tracer program. This can produce fancy 3D images, but keep in mind that the Povray file may need to be edited manually to get good camera angles, lighting effects and so forth.

```
(rewrite-and-raytrace lsystem filename
    &key depth width)
```

**depth** : Number of rewrites.

**width** : Width multiplier for cylinders/cones.

If OpenGL bindings are loaded and working, fast previewing is possible using the function `gl-preview`, which will show the geometry in an OpenGL window. It is possible to rotate or zoom the view by holding down the left and middle mouse buttons, respectively. The window can be closed by pressing the right button.

```
(gl-preview lsystem
    &key width line-style cylinder-slices depth lighting limits)
```

**width** : Line/cylinder width multiplier.

**line-style** : `:lines` or `:cylinders`.

**cylinder-slices** : Higher numbers give smoother cylinders.

**depth** : Number of rewrites.

**lighting** : Boolean, lighting on or off.

**limits** : `'((x1 y1 z1) (x2 y2 z2))` defines the geometric limits of the coordinate system. These are normally calculated automatically.

### Animations

A series of images of a growing L-system can be created by using the animation commands `eps-animation`, `povray-animation` and `gl-animation`.

One common aspect of the three functions is that they all take a frame-list argument. Frame lists tell which rewriting steps are part of the animation. Each element of a frame list is either:

- A single frame number.
- A pair ($a$ $b$), representing an interval of frames from $a$ to $b$, inclusive.
- A triplet ($a$ $b$ *step*), representing the frames $a, a + step, a + 2 * step \ldots, a + n * step$ where $n$ is the largest possible integer such that $a + n * step \leq b$.

Frames in the frame list must be listed in increasing order. For instance, the frame list `'(1 4 (6 10) (12 20 3))` represents the frames $1, 4, 6, 7, 8, 9, 10, 12, 15, 18$, and the frame list `((0 1000 2))` represents all even frame numbers from 0 to 1000.

Postscript animations are created by `eps-animation`:

```
(eps-animation lsystem &key filename-prefix frames border-percent)
```

**filename-prefix** : for frame number $n$, "$n$.eps" will be appended to this prefix to make a filename.

**frames** : frame list.

**border-percent** : percentage of border (whitespace) around image.

Povray animations are created by `povray-animation`:

```
(povray-animation lsystem
    &key filename-prefix frames width full-scene)
```

**filename-prefix** : for frame number $n$, "$n$.pov" will be appended to this prefix to make a filename.

**frames** : frame list.

**width** : width multiplier for cylinders/cones.

**full-scene** : boolean; a camera and light source will be generated if true.

Real-time[8] OpenGL animations can be created by the function `gl-animation`:

[8] There are obviously some limiting factors for "real-time" L-Lisp animations, such as the complexity of the model, the OpenGL drivers and the CPU speed. No matter what processing power is available, it will always be possible (even easy) to create models that bring down the frame rate.

(`gl-animation` *lsystem*
    `&key` *width line-style line-style cylinder-splices lighting limits*
        *recenter frames frame-delay instant-animate*)

**width, line-style, cylinder-splices, lighting, limits** : see `gl-preview`.

**recenter** : automatically center and resize each frame.

**frames** : frame list.

**frame-delay** : delay between frames, in seconds.

**instant-animate** : boolean; start animation right away if true, otherwise wait for “A” keypress.

Keypresses in OpenGL mode:

**A** : toggle animation/pause.

**R** : restart animation.

**P** : output Povray from current frame.

**Q** : quit.

### 4.7.9 Turtle interpretation

Turtle interpretation is handled automatically by L-Lisp, but it can be tuned and extended in various ways. This section describes the existing turtle commands and the mechanisms for adding new ones.

#### Predefined turtle commands

Most predefined turtle commands have both a short name (which is similar to the ones used in earlier chapters), and a long one, which is by convention a Lisp keyword.

Figure 4.4 lists both the short and long names of some common turtle commands together with the type of parameter expected. Parameters are either ignored, optional, required, or, in the case of brackets, internal[9].

The environmentally sensitive commands (such as `?P`) return a value by storing it as the first parameter. An L-system production will normally create the module `(?P nil)`, and during turtle interpretation the `nil` will be replaced by the current position, which is of the type `(simple-array double-float (3))`.

#### The `turtle` structure

In order to understand how the turtle commands are implemented, it is necessary to have a look at the `turtle` structure, which contains a local coordinate system, describing its position and orientation in 3D space. It also contains other values, and figure 4.5 gives an overview.

[9]For an explanation of how bracket parameters are used, see section 4.8.4

Figure 4.4: Turtle functions

<table>
<tr><th>Name</th><th>Long name</th><th>Parameters</th><th>Par. type</th></tr>
<tr><td>]</td><td></td><td></td><td>internal</td></tr>
<tr><td>[</td><td></td><td></td><td>internal</td></tr>
<tr><td>F</td><td>:forward</td><td>length</td><td>optional</td></tr>
<tr><td>\f</td><td>:forward-no-line</td><td>length</td><td>optional</td></tr>
<tr><td>+</td><td>:turn-left</td><td>angle</td><td>optional</td></tr>
<tr><td>-</td><td>:turn-right</td><td>angle</td><td>optional</td></tr>
<tr><td>&</td><td>:pitch-down</td><td>angle</td><td>optional</td></tr>
<tr><td>^</td><td>:pitch-up</td><td>angle</td><td>optional</td></tr>
<tr><td>\\</td><td>:roll-left</td><td>angle</td><td>optional</td></tr>
<tr><td>/</td><td>:roll-right</td><td>angle</td><td>optional</td></tr>
<tr><td>\|</td><td>:turn-around</td><td>angle</td><td>ignored</td></tr>
<tr><td>!</td><td>:set-width</td><td>new width</td><td>required</td></tr>
<tr><td>{</td><td>:start-polygon</td><td></td><td>ignored</td></tr>
<tr><td>\.</td><td>:add-vertex</td><td></td><td>ignored</td></tr>
<tr><td>f.</td><td>:forward-vertex</td><td>length</td><td>optional</td></tr>
<tr><td>}</td><td>:end-polygon</td><td></td><td>ignored</td></tr>
<tr><td>m{</td><td>:start-mesh</td><td></td><td>ignored</td></tr>
<tr><td>m.</td><td>:mesh-vertex</td><td></td><td>ignored</td></tr>
<tr><td>mf</td><td>:forward-mesh-vertex</td><td>length</td><td>optional</td></tr>
<tr><td>m/</td><td>:new-strand</td><td></td><td>ignored</td></tr>
<tr><td>m}</td><td>:end-mesh</td><td></td><td>ignored</td></tr>
<tr><td>?P</td><td>:get-position</td><td>returns position</td><td>required</td></tr>
<tr><td>?H</td><td>:get-heading</td><td>returns heading vector</td><td>required</td></tr>
<tr><td>?U</td><td>:get-up-vector</td><td>returns up vector</td><td>required</td></tr>
<tr><td>?L</td><td>:get-left-vector</td><td>returns left vector</td><td>required</td></tr>
<tr><td>?T</td><td>:get-turtle</td><td>returns turtle</td><td>required</td></tr>
<tr><td>@M</td><td>:set-position</td><td>position vector</td><td>required</td></tr>
<tr><td>@O</td><td>:sphere</td><td>radius</td><td>optional</td></tr>
<tr><td>@R</td><td>:rotate-towards</td><td>vector</td><td>optional</td></tr>
<tr><td></td><td>:color</td><td>color vector</td><td>required</td></tr>
</table>

Figure 4.5: The `turtle` structure

| **Slot** | **Default** | **Description** |
|---|---|---|
| `pos` | `#(0.0 0.0 0.0)` | 3D position. |
| `H` | `#(0.0 1.0 0.0)` | Heading vector (forward direction). |
| `L` | `#(-1.0 0.0 0.0)` | Left vector. |
| `U` | `#(0.0 0.0 -1.0)` | Up vector. |
| `angle` | 90.0 | Default angle increment. |
| `width` | 1.0 | Branch/segment width. |
| `prev-width` | `nil` | Previous width; used to create smooth width transitions for cylinders. |
| `color` | `nil` | Colors are given as RGB vectors (red, green and blue values between 0.0 and 1.0). |
| `texture` | `nil` | Name of a povray texture. *(not implemented)* |
| `shared-polygon-stack` | `(list nil)` | A stack of unfinished polygons. |
| `shared-mesh` | `(list nil)` | An unfinished polygon mesh. |
| `produce-spheres` | `t` | Boolean; decides whether cylinders should be joined by spheres in Povray. |

Each slot has an accessor named `turtle-`*slotname*.

The polygon and mesh slots are shared between copies, which means that popping the turtle stack does not change the values. Additional accessors called `turtle-polygon-stack` and `turtle-mesh` have been defined to simplify access.

### Geometric shapes

The result of a turtle interpretation is a vector of geometric shapes. The predefined shapes are implemented as structures rather than CLOS classes, and the following geometric shapes are predefined in L-Lisp:

`line` : represents a line or a cylinder.[10] Created by the turtle commands `:forward` (`F`) and `:line`.

`sphere` : a sphere. Created by the turtle command `:sphere` (`@O`).

`polygon` : a flat polygon. Created by the turtle commands `:start-polygon` (`{`), `:add-vertex` (`.`), `:forward-vertex` (`f.`) and `:end-polygon` (`}`).

`mesh` : a polygon mesh with smooth shading for Povray and OpenGL. See below for more details.

Figure 4.6 (also used at the title page of the thesis) shows off a ray-traced image of a berry bush where the leaves are polygon meshes and the berries are spheres created by L-Lisp.

[10]The cylinders produced by L-Lisp can actually have different widths at each end. In Povray this geometric shape is

Figure 4.6: Berry bush

### Polygon meshes

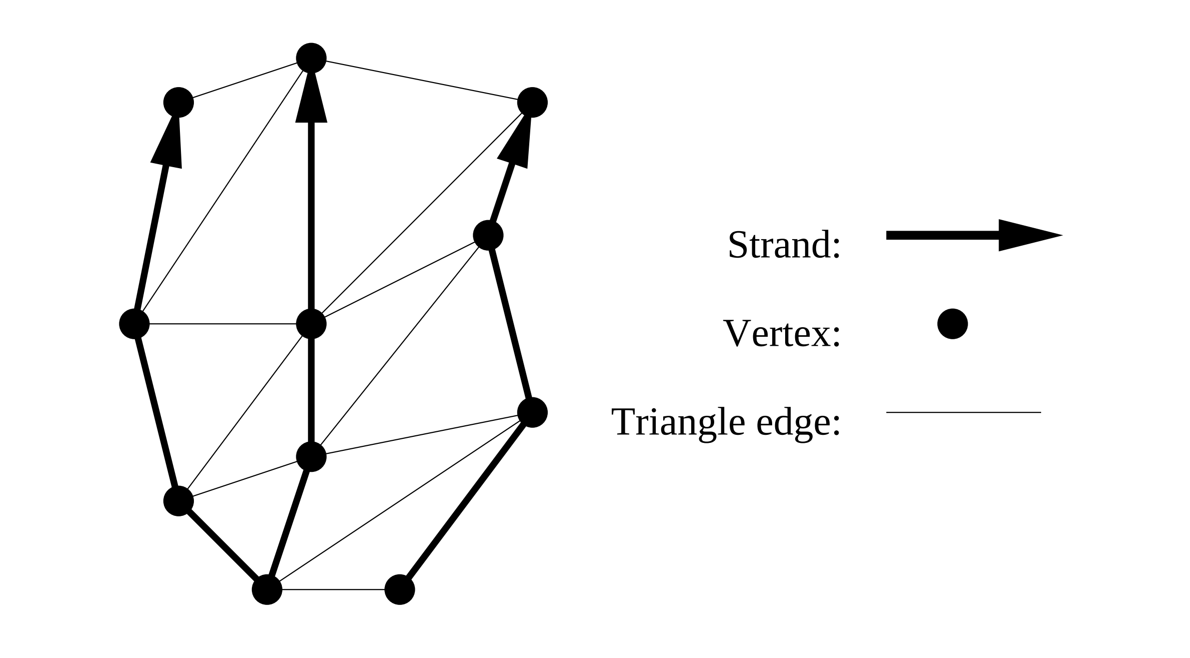
L-Lisp provides turtle commands for creating polygon meshes. A polygon mesh in L-Lisp is a surface consisting of smoothly shaded triangles that can give the impression of bent surfaces. This is useful for creating leaves, for instance.

Meshes are defined by creating "strands" of vertices. Figure 4.7 shows an example of how three strands of vertices can be used to create a surface.

The command `:start-mesh` creates a new mesh, and start the first strand. New vertices are added using `:mesh-vertex` or `:forward-mesh-vertex`. The latter is a combination of a `:forward-no-line` and `:mesh-vertex` command. A new strand is started using `:new-strand`. This does not move the turtle, so if you want each strand to start at the same point you must do so manually, for instance using brackets. The command `:end-mesh` converts the strands of vertices to a triangle mesh and returns the resulting mesh. This is done by creating a strip of triangles between each consecutive pair of strands.

called a cone, not a cylinder.

Figure 4.7: A schematic mesh example




Also, averaged normal vectors are calculated for each vertex; these are used to create smooth shadow effects for both Povray and OpenGL.

The short names for each of the above commands can be found in figure 4.4.

**Turtle functions**

Turtle functions are functions that implement specific turtle commands. The mappings between turtle commands and the functions are stored in the hash table `*turtle-functions*`.

As mentioned in section 4.5.4, each predefined turtle command (except brackets) has been given both a short and long name in L-Lisp. The long names are Lisp *keywords*, which in this context just means that they start with a colon.

The macro `def-turtle-function` is for defining turtle functions:

```
(def-turtle-function (name*) (turtle param*)
  body)
```

The parameter `turtle` is required, but can of course be ignored by the function or have a different name. For instance, a command with short name `S` and long name `:long-name`, that expects one optional parameter which defaults to 1.0 can be defined as

```
(def-turtle-function (:long-name S) (turtle &optional (param 1.0))
  ...)
```

You can supply as many command names as you wish for each function. If you only want one name, you can drop the parenthesis around the name:

```
(def-turtle-function :some-name (turtle &optional (param 1.0))
  ...)
```

Turtle functions that set the parameters cannot use the `def-turtle-function` form. Instead, they can be implemented by a `def-turtle-function-raw` form which gives full control over the list of parameters:

```
(def-turtle-function-raw (name*) (turtle params)
  body)
```

Within this function, the parameters can now be changed at will. For instance, the `?P` or `:get-position` command is implemented as

```
(def-turtle-function-raw (:get-position ?P) (turtle params)
  (setf (first params) (copy-seq (turtle-pos turtle)))
  nil)
```

### 4.7.10 Spline editor

L-Lisp includes an OpenGL-based spline editor. The functionality includes basic editing, saving, loading, and calculating spline values. A screenshot is given in figure 4.8.

The function `edit-spline` starts the graphical spline editor and allows basic save and load functionality:

```
(edit-spline &key spline filename eps-filename)
```

**spline** : The spline to edit. If `nil`, create a new spline or read from file.

**filename** : File where spline is to be loaded and saved.

**eps-filename** : Save Postscript graphics of the spline to this file.

The following key and mouse bindings are supported in the spline editor:

**Left mouse button** : Mark point (click) or move point (drag).

**Middle mouse button** : Add new point.

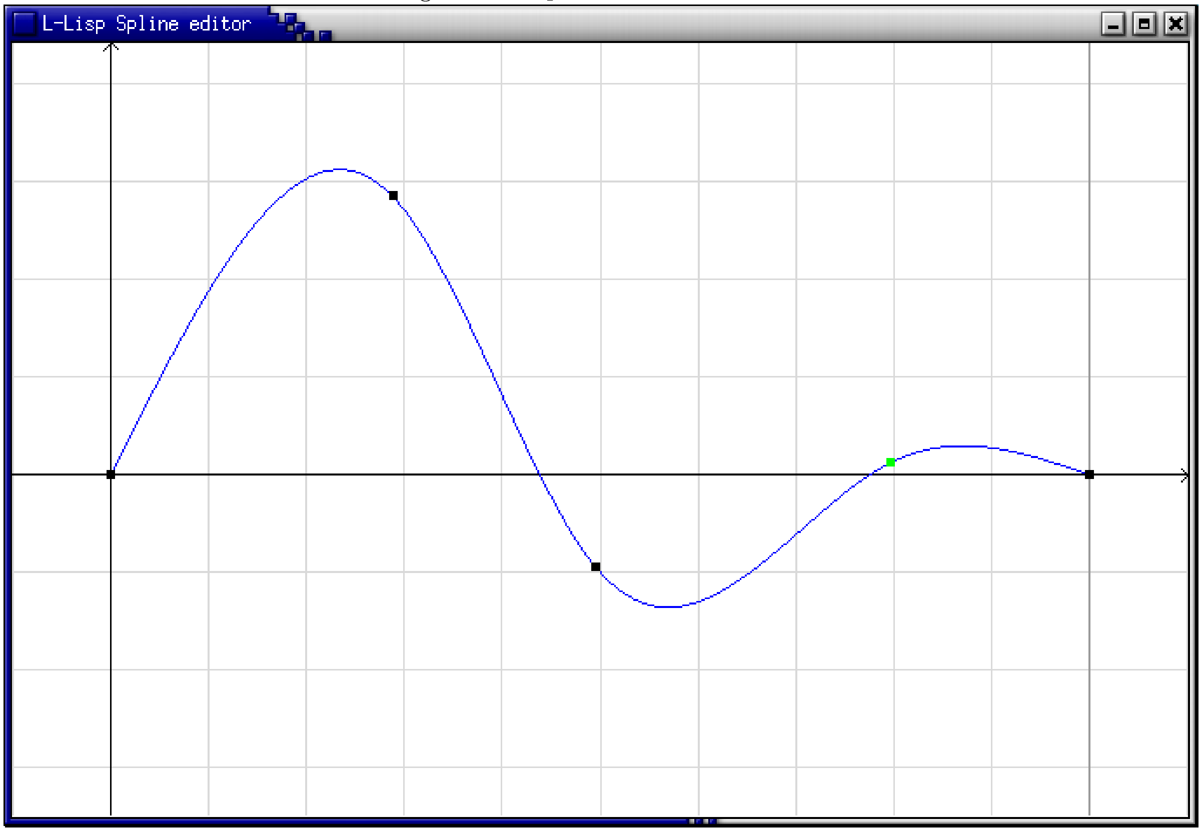
**Right mouse button or 'Q'** : Save and quit.

**'D'** : Delete marked point.

**'Escape'** : Quit without saving any files.

To get a single spline-value, the `spline-value` function can be used (although `spline-values` and `make-spline-function` are often better alternatives):

Figure 4.8: Spline editor screenshot



```
(spline-value spline x-value)
```

To get many spline values in one call, use `spline-values`, which returns a vector of y-values:

```
(spline-values spline &key start end steps)
```

**start** : Minimum x-value.

**end** : Maximum x-value.

**steps** : Number of values to generate in the (*start end*) interval.

To create a compiled function object, call `make-spline-function`, which will compile the function on-the-fly[11]:

```
(make-spline-function spline &optional number-type)
```

**number-type** : The internal floating-point numeric type, by default `double-float`. `single-float` is (probably) faster, but less accurate.

To save a spline, use the `output-spline` function:

```
(output-spline spline filename)
```

To load a spline from file, use the `input-spline` function:

```
(input-spline filename &optional make-function)
```

**make-function** : Create a function object which can be retrieved using the `spline-function` accessor. This is done using `make-spline-function`; see above for details.

The base function for creating splines is called `natural-cubic-spline` (normally there is no need to call it directly):

```
(natural-cubic-spline xvec yvec &key make-function)
```

**xvec** : Vector of x-values in ascending order.

**yvec** : Vector of y-values corresponding to *xvec*.

**make-function** : Boolean; create a compiled function-object by calling `make-spline-function` (see above).

[11] Compiling functions at "run-time" rather than "compile-time" (or perhaps "development-time" would be more accurate) is a feature of Common Lisp which is rarely used. Compiling spline functions on-the-fly is an example which shows that it can in fact be useful.

## 4.8 Implementation

This section describes the internals of the L-Lisp implementation, including important algorithms and explanations of code generated by macros. Those who just want to use L-Lisp as a framework can skip this section.

Part of the complete source code is given in appendix A.

### 4.8.1 Rewriting algorithms

The basic algorithm for a single rewrite is outlined in section 4.7.2. Figure 4.9 shows pseudocode[12] for the algorithm. Note that the implementation in `rewrite1` also contains some context-checking and bracket-specific code which is not shown here.

Figure 4.9: The `rewrite1` algorithm

```
function rewrite1(l-system)
    old-string := l-system.rstring
    new-string := <expandable vector>
    for i := 0 to old-string.length
        l-system.current-module := old-string[i]
        answer := l-productions(l-system)
        if <answer is a list>
            for element := <each element of answer>
               <append element to new-string>
        else if answer = t
            <append old-string[i] to new-string>
        else
            <error>
    l-system.rstring := new-string
    return new-string
```

The `rewrite` function is very simple; it initializes the rewriting string with the axiom, then calls `rewrite1` the specified number of times.

### 4.8.2 Homomorphism and decomposition algorithms

Homomorphism and decomposition can be expressed by a common algorithm that recursively expands each module of the rewriting string by calling either the `homomorphism` or `decomposition` methods. Like `l-productions`, these methods must return either a list of modules, or `t`.

To avoid infinite recursion, the slots `homomorphism-depth` and `decomposition-depth` define limits to how deep the recursion can go. The default depths are zero, which means no rewriting will happen.

The common algorithm is implemented using two functions `tree-rewrite` and `expand-module`.

[12]This pseudocode uses a mix of Pascal and Python-like syntax, and Lisp variable naming. Hopefully it is clear enough.

The production function is passed along as a parameter. This is possible in most programming languages, and trivial in Lisp. With these functions as a base, the code for the actual homomorphism and decomposition rewriting is simple. Pseudocode is given in figure 4.10.

Figure 4.10: Homomorphism and decomposition algorithms

```
function tree-rewrite(l-system, production-func, max-depth)
    old-string := l-system.rstring
    new-string := <expandable vector>
    for i := 0 to old-string.length
        l-system.current-module := old-string[i]
        expand-module(l-system, new-string, production-func,
                      max-depth)
    return new-string

function expand-module(l-system, new-string, production-func,
                       max-depth)
    if max-depth = 0
        <warn that maximum depth has been reached>
        <append l-system.current-module to new-string>
    else
        answer := production-func(l-system)
        if <answer is a list>
            for element := <each element of answer>
                l-system.current-module := element
                expand-module(l-system, new-string,
                              production-func, max-depth - 1)
        else if answer = t
            <append l-system.current-module to new-string>
        else
            <error>

function homomorphism-rewrite(l-system)
    new-string := tree-rewrite(l-system, homomorphism,
                               l-system.homomorphism-depth)
    l-system.hstring := new-string
    return new-string

function decomposition-rewrite(l-system)
    new-string := tree-rewrite(l-system, decomposition,
                               l-system.decomposition-depth)
    l-system.rstring := new-string
    return new-string
```

### 4.8.3 `choose-production` implementation

The implementation of `choose-production`, as most other Lisp macros, is very Lisp-specific, and hard to express in pseudocode. A more informal explanation is given by showing roughly what the macro is transformed (macroexpanded) into before compilation[13]:

A macro of the form

```
(choose-production lsystem (module1 exprs1) ... (modulen exprsn))
```

is transformed into something like

```
(block blockname
  (let ((cmodule (current-module lsystem)))
     (or (prod cmodule module1 exprs1)
         ...
         (prod cmodule modulen exprsn)
         t)))
```

The actual names (symbols) for *blockname* and *cmodule* are generated during macroexpansion, they are are so-called *gensyms*. This insures that there are no name clashes with other variables.

The `prod` macro represents a single production, and is explained below.

The macro uses `or`, which returns the first non-`nil` argument, which in this case is the first matching production.

Also, any "`(-->` ...`)`" forms in the expressions are replaced with "`(return-from` *blockname* `(-->` ...`))`", which explains the named block around the `or` clause.

A `prod` macro is transformed differently depending on whether the module is a parametric module or just a symbol. If it is a symbol, then

```
(prod cmodule symbol exprs)
```

is transformed to

```
(when (eq symbol cmodule) exprs)
```

If it is a parametric module, then

```
(prod cmodule (symbol param1 ... paramn) exprs)
```

is transformed to

```
(when (and (consp cmodule)
           (eq (first cmodule) symbol)
           (= (length cmodule) n))
  (destructuring-bind (param1 ... paramn) (rest cmodule)
    exprs))
```

The non-parametric version is simple, but the parametric version has to do several checks, then bind the variables. It uses `destructuring-bind` instead of `let`.

Together, the macros mean that, for instance, the expression

[13] Compilation in Lisp is different from most other languages. Depending on how you view the process you could say the macroexpansion happens *before* or *during* compilation (or evaluation).

```
(choose-production ls
  (a (--> b))
  ((a x) (--> (b x x)))
  ((b x y) (--> (a (+ x y)))))
```

is roughly equivalent to

```
(block block1
  (let ((mod (current-module ls)))
    (or (when (eq mod 'a)
          (return-from block1 (--> b))
        (when (and (consp mod)
                   (eq (first mod) 'a)
                   (= (length mod) 2))
          (destructuring-bind (x) (rest mod)
            (return-from block1 (--> (b x x)))))
        (when (and (consp mod)
                   (eq (first mod) 'b)
                   (= (length mod) 3))
          (destructuring-bind (x y) (rest mod)
            (return-from block1 (--> (a (+ x y))))))
        t)))
```

A few comments about this example:

- Using an `or` clause as opposed to `case` or `cond` may seem confusing, but it is the simplest way to get the first non-nil of the expressions.
- The `block` and `return-from` statements are not necessary in this particular case. They will be necessary in some context-sensitive L-systems though.
- The test if the current module is a list (Lisp cons) is done twice. It is possible to avoid this, and an optimization could be implemented in future versions of `choose-production`.

### 4.8.4 Context matching algorithms

The basic algorithm for context matching consists of going left or right through the rewriting string, ignoring some modules specified by the user, matching each symbol against the specified context. For parametric L-systems we also need to collect the parameters in the context, and for bracketed L-systems we need to implement the rules for skipping branch segments.

Since L-Lisp is meant to work for all L-systems, it implements all of the above.

Context matching is done by three functions for left and right contexts (six in all). Bracketed context matching is easier for left contexts than right contexts. Figure 4.11 shows the algorithm.

- `next-module-left` is a simple wrapper that returns the next module to the left, or a special symbol `EMPTY` if we are already at the leftmost module.

Figure 4.11: Left context matching algorithm

```
function next-module-left (l-system)
    if l-system.context-pos = 0 then
        return EMPTY
    else
        l-system.context-pos := l-system.context-pos - 1
        return l-system.rstring[l-system.context-pos]

function next-context-module-left (l-system, ignore-list)
    new-ignore-list := <ignore-list + '[' and ']'>
    repeat
        module := next-module-left(l-system)
        symbol := <module's symbol>
        if symbol = ']' then
            <skip to matching '['>
    until <symbol is not in new-ignore-list>
    return module

function match-context-left (l-system, context-pattern,
                             ignore-list)
    ;; CONTEXT-PATTERN is a list of symbols and lengths
    l-system.context-pos := l-system.pos
    param-list := <empty list>
    new-context-pattern := <reverse of context-pattern>
    for element := <each element of new-context-pattern>
        module := next-context-module-left(l-system,
                                           ignore-list)
        params := <list of module's parameters>
        symbol := <module's symbol>
        if (symbol = element.symbol) and
           (params.length = element.length) then
            <append params to param-list>
        else
            return false, <empty list>
    return true, param-list
```

- `next-context-module-left` returns the next module that should be matched against the context. It ignores the symbols given in `ignore-list`, and skips brackets as necessary. Since the left context should not contain brackets, this is easy. Note that the method for finding matching pairs of brackets is not given in the pseudocode; this will be discussed later.

- `match-context-left` matches a list of symbols and lengths (`context-pattern`) against the left context. If a match is found, it returns two values: true and a list of parameters (if any). If no match is found, it returns false and an empty list.

Right context matching is similar, but more complex since it is possible to either skip branches, or include parts of them in the context. Figure 4.12 shows the algorithm.

The `next-context-module-right` cannot simply skip all brackets as the left version did, this must be done in `match-context-right` instead. `match-context-right` is more complex than `match-context-left` because of the special rules for bracketed context-sensitivity; each branch can either be skipped or considered. For instance, the right context $A[BCD][EFG]HI$ is matched by the patterns $AH$, $A[BC]H$ and $A[B][EF]H$, but not by $A[E][B]H$.

**Optimizing bracket skipping**

Context matching without brackets usually means looking at a few modules to the left and right of the current module, which is relatively fast. If the maximum number of context modules checked for each step is $k$, and the length of the rewriting string is $n$, then the cost of context matching for one rewrite is $O(nk)$.

For bracketed L-systems, the context matching might involve skipping over much more than $k$ modules in some steps. If simple linear search is used for skipping branches, the cost is obviously no more than $O(n^2)$, and even though tighter limits will exist for most bracketed L-systems, it is clear that a lot more modules are being skipped than in the bracket-free case.

One possible strategy for improving performance would be to represent the bracketed rewriting string as a tree internally. Each element in the rewriting string would then be either a module or a branch, and a "branch" would be another rewriting string. For instance, the string $A[BCD]E$ could be represented as $A[p_1]E$, where $p_1$ is a pointer to the string $BCD$. This would complicate allocation and rewriting among other things, but skipping a branch would only use constant time.

Another strategy, the one used by L-Lisp, is to add special parameters to the bracket symbols `[` and `]`. Brackets are represented internally as `[(x)` and `](y)`,[14] where `x` is the position of the matching right bracket in the string, and `y` the position of the matching left bracket. The string $A[BCD]E$ will then be represented as $A[(5)BCD](1)E$, assuming that indexes start at zero. Skipping a branch obviously takes constant time. The following changes needs to be done to rewriting and context matching:

- After each rewriting step, the function `add-bracket-params` is called. The algorithm for adding bracket parameters is trivial.

[14] Or `([ x)` and `(] y)` in Lisp syntax

Figure 4.12: Right context matching algorithm

```
function next-module-right (l-system)
    if l-system.context-pos >= (l-system.rstring.length - 1) then
        return EMPTY
    else
        l-system.context-pos := l-system.context-pos + 1
        return l-system.rstring[l-system.context-pos]

function next-context-module-right (l-system, ignore-list)
    repeat
        module := next-module-right(l-system)
        symbol := <module's symbol>
    until <symbol is not in ignore-list>
    return module

function match-context-right (l-system, context-pattern,
                              ignore-list)
    l-system.context-pos := l-system.pos
    param-list := <empty list>
    for element := <each element of context-pattern>
        try-again := true
        while try-again = true
            module := next-context-module-right(l-system,
                                                ignore-list)
            symbol := <module's symbol>
            params := <module's parameter list>
            try-again := false
            if element.symbol = ']' then
                ;; attempt to skip to ']'
                if <there is an unmatched ']' in rstring> then
                    <skip to the unmatched ']'>
                else return false, <empty list>
            else if (symbol = element.symbol) and
                    (params.length = element.length) then
                ;; module matches
                <append params to param-list>
            else if (symbol = '[') then
                ;; no match, but skip brackets and try again
                try-again := true
                <skip right to unmatched ']'>
            else
                ;; no match
                return false, <empty list>
        ;; end while
    ;; end for
    return true, param-list
```

- Skipping brackets in `next-context-module-left` is done by using the "`]`" parameter. This obviously takes constant time.
- In `match-context-right`, we sometimes need to skip from the current position to the next unmatched "`]`". By maintaining a stack of any brackets we pass by, this, too can be done in constant time. A modified version of `match-context-right` is given in figure 4.13. This algorithm is implemented in L-Lisp.

### Context checking macros

The context checking macros are given in the form

```
(with-left-context lsystem (module_1 ... module_n) exprs)
(with-right-context lsystem (module_1 ... module_n) exprs)
```

However, when used within a `choose-production` macro, the *lsystem* argument is added automatically, so the macros must be given as described in section 4.7.4, page 69.

The macros are in a sense "wrappers" for the `match-context-left` and `match-context-right` functions. If none of the modules have parameters, macroexpansion works as follows:

```
(with-left-context lsystem (symbol_1 ... symbol_n) exprs)
```

is transformed to

```
(when (match-context-left
        lsystem '((symbol_1 . 0) ... (symbol_n . 0)))
  exprs)
```

The parametric case is more complex. Like `choose-production`, the module parameters should be bound to variables by the macro. The matching functions return two values; a boolean that tells whether a match was found, and a list of parameter values. The macro works by transforming

```
(with-left-context lsystem (module_1 ... module_n) exprs)
```

to

```
(multiple-value-bind (match parameters)
    (match-context-left lsystem '((symbol_1 l_1) ... (symbol_n l_n)))
  (when match
    (destructuring-bind (param_1 ... param_k) parameters
      exprs)))
```

where $l_i$ is the number of parameters of $module_i$ in the pattern, and $param_i$ is the $i$th of all parameter names. *match* and *parameters* are gensyms.

For instance, the expression

```
(with-left-context lsys ((a x) b (c y z)) (--> ...))
```

is equivalent to

Figure 4.13: Modified right context matching algorithm

```
function match-context-right (l-system, context-pattern,
                              ignore-list)
    l-system.context-pos := l-system.pos
    param-list := <empty list>
    mstack := <empty stack>
    for element := <each element of context-pattern>
        try-again := true
        while try-again = true
            module := next-context-module-right(l-system,
                                                ignore-list)
            symbol := <module's symbol>
            params := <module's parameter list>
            try-again := false
            if element.symbol = ']' then
                ;; attempt to skip to ']'
                if <mstack is empty>
                    return false, <empty list>
                else
                    l-system.context-pos := pop(mstack)
            else if element.symbol = '[' then
                ;; if brackets match, then push end position on
                ;; mstack, else fail
                if symbol = '[' then
                    push(<first element of params>, mstack)
                else
                    return false, <empty list>
            else if (symbol = element.symbol) and
                    (params.length = element.length) then
                ;; module matches
                <append params to param-list>
            else if (symbol = '[') then
                ;; no match, but skip brackets and try again
                try-again := true
                <skip right to unmatched ']'>
            else
                ;; no match
                return false, <empty list>
        ;; end while
    ;; end for
    return true, param-list
```

```
(multiple-value-bind (match parameters)
    (match-context-left lsys '((a . 1) (b . 0) (c . 2)))
  (when match
    (destructuring-bind (x y z) parameters
      (--> ...))))
```

The `with-right-context` macro is exactly like `with-left-context`, except that it calls `match-context-right` instead of `match-context-left`.

`with-lc` and `with-rc` are just shorter names for the same macros.

### 4.8.5 Turtle interpretation

The function `turtle-interpret` does the basic turtle interpretation. It takes the rewrite string as input, and returns a vector of geometry shapes such as lines or polygons.

The basic algorithm consists of maintaining a stack of "turtles", where a turtle is a local coordinate system. Turtles also contain some other information like line width and color. For each module in the rewriting string, we try to find the turtle function associated with the symbol. If a turtle function is found, it is called, and the result, if non-nil, is added to a geometry vector. When we have gone through the whole rewrite string, we return the geometry vector.

The idea behind turtle functions is to move as much as possible of the turtle functionality away from `turtle-interpret` and into separate functions. In addition to making the code more readable, this makes it possible to add new functionality without changing any of the predefined functions, although it will sometimes be necessary to add new slots to the turtle structure. In this implementation, all turtle commands except for the brackets `[` and `]` uses turtle functions.

With these decisions in mind, the algorithm can be described by the pseudocode in figure 4.14.

## 4.9 Further work

L-Lisp provides a solid basis for L-systems in Lisp, but there are many features that should be added to make a more professional package for graphical or biological purposes. For instance, support for more 3D shapes, better integration with the ray-tracer or better support for generating landscapes would be nice.

From a theoretical viewpoint, there are more fundamental issues to be investigated:

**Tree transformations** : Rewriting bracketed L-systems is a form of tree transformation. However, the context matching in L-systems is a rather limited model, even with the L-Lisp extensions in section 4.7.4. Other approaches for tree transformations have been used in XML transformations [27] and compilers [3, 13]. Lisp macros uses yet another approach for tree transformations. Integrating similar ideas in L-Lisp could result in a more powerful rewriting model.

**Interaction with the environment** : In open L-systems [14], there are two different programs that simulate a plant and its environment. In L-Lisp, there could potentially be a tighter integration

Figure 4.14: Turtle interpretation algorithm

```
function turtle-interpret (rstring, angle-increment)
    geometry := <empty vector>
    turtle-stack := <empty stack>
    turtle := <new turtle>
    turtle.angle-increment := angle-increment
    push(turtle, turtle-stack)
    for module := <each module of rstring>
        symbol := <module's symbol>
        params := <module's parameters>
        if symbol = '[' then
            turtle := copy(turtle)
            push(turtle, turtle-stack)
        else if symbol = ']' then
            pop(turtle-stack)
            turtle := top(turtle-stack)
        else
            function := lookup-turtle-function(symbol)
            if <function exists> then
                returnval := function(turtle, params)
                if returnval <> NIL then
                    <append returnval to geometry>
    ;; end for
    return geometry
```

between the two processes, since the whole simulation would be a single Lisp program. However, open L-systems in L-Lisp is an unresearched area.

**Different kinds of parameters** : L-Lisp supports parameters of any type already, but the only non-numerical types in use thus far are position vectors for query modules; `(?P pos)` is used instead of `(?P x y z)`. The possibilities which arise from using e.g. lists or other data structures as parameters has yet to be examined.

While L-Lisp isn't necessary to investigate these aspects, it will allow suggested solutions to be implemented and tested quickly. In fact, one of Lisp's great strengths is as a vehicle for testing experimental language features, and this is also true for L-Lisp.

# Appendix A

# L-Lisp source code

Parts of the L-Lisp source code is included here. The following files are included:

**packages.lisp** : Defines the `l-systems`, `spline-editor` and `l-system-examples` packages.

**buffer.lisp** : Implements a dynamically growing sequence, useful both for rewriting and turtle interpretation.

**lsystem.lisp** : The `l-system` class, rewriting, context checking, miscellaneous L-system macros.

**turtle.lisp** : 3D vector and matrix operations, turtle interpretation, generating Postscript and Povray.

The following files were omitted for space reasons, or because they are not directly relevant for L-systems:

**opengl.lisp** : OpenGL previews and animations for the X Window System.

**nrandom.lisp** : Generating random numbers from normal distributions.

**splineed.lisp** : Spline editor.

**examples.lisp** : Examples of L-systems.

**advanced-examples.lisp** : Advanced examples of L-systems.

## A.1 packages.lisp

```
(in-package :cl-user)

(defpackage l-systems
  (:nicknames lsystem lsys ls)
  (:use common-lisp)
  (:export
   ;; trigonometry
```

```
   deg-to-rad rad-to-deg cosd sind tand
   ;; 3D vectors
   v3 vec3 vlength normalize equalvec almost-equalvec zerovec vec- vec+
   dot-product cross-product
   ;; turtle
   turtle turtle-pos turtle-H turtle-L turtle-U turtle-angle turtle-width
   turtle-prev-width turtle-color turtle-texture copy-turtle rotate
   rotate-towards turn pitch roll move-forward turtle-interpret
   def-turtle-function-raw def-turtle-function turtle-function
   ;; geometric shapes
   line line-p1 line-p2 line-width
   box box-pos box-size
   ;; short names of turtle functions
   F \f + - \& \^ \\ / \| ! @O \{ \. f. \} m{ m. m/ m} mf @M @R
   ?P ?H ?U ?L ?T \[ \]
   ;; creating images
   output-simple-eps output-povray
   ;; l-system class
   l-system axiom depth angle-increment ignore-list consider-list
   homomorphism-depth decomposition-depth sensitive
   rstring hstring geometry current-depth current-module
   line-style cylinder-width limits frame-delay frame-list
   ;; rewriting
   l-productions homomorphism decomposition
   rewrite rewrite1 homomorphism-rewrite decomposition-rewrite
   ;; geometry methods
   create-geometry rewrite-and-preview rewrite-and-raytrace
   timed-raytrace timed-preview lsys2eps lsys2pov
   eps-animation povray-animation
   top-of-framelist next-framelist extract-framelist
   ;; l-system macros
   choose-production --> stochastic-choice
   with-left-context with-right-context with-lc with-rc
   while-ignoring while-considering
   ;; OpenGL stuff
   gl-preview gl-animation
   ;; Random number generation
   nrandom
   ))

(defpackage spline-editor
  (:nicknames splineed spline)
  (:use common-lisp)
  (:export
   spline spline-x spline-y spline-a spline-b spline-b spline-c spline-d
   spline-function natural-cubic-spline spline-value spline-values
   make-spline-function output-spline input-spline output-spline-to-eps
   edit-spline))

(defpackage l-system-examples
  (:nicknames lsx)
  (:use common-lisp l-systems spline-editor))
```

## A.2 buffer.lisp

```
;;; *** buffer.lisp ***
;;;
;;; This file is part of L-Lisp by Knut Arild Erstad.
;;; Implements a growing sequence as a list of vectors.
```

```
(in-package :l-systems)

;; *** Buffer data structure ***
(defstruct (buffer (:constructor internal-make-buffer))
  (inc-size 0 :type fixnum)     ; size of each "minibuffer"
  (list nil)                    ; list of minibuffers
  (current nil)                 ; current minibuffer (for fast access)
  (size 0 :type fixnum)         ; total number of elements
  (current-size 0 :type fixnum) ; number of elements in current buffer
  (list-size 0 :type fixnum))   ; length of list

(defun make-buffer (&optional (inc-size 4096))
  (internal-make-buffer :inc-size inc-size
                        :current-size inc-size))

(declaim (inline buffer-push))
(defun buffer-push (elt buffer)
  "Add a new element to the end of the buffer (like VECTOR-PUSH-EXTEND)."
  (declare (optimize (speed 3) (safety 0)))
  (incf (buffer-size buffer))
  (if (>= (buffer-current-size buffer) (buffer-inc-size buffer))
    (let ((newbuf (make-array (buffer-inc-size buffer))))
      (setf (buffer-current buffer) newbuf)
      (push newbuf (buffer-list buffer))
      (incf (buffer-list-size buffer))
      (setf (buffer-current-size buffer) 1)
      (setf (svref newbuf 0) elt))
    (let ((pos (buffer-current-size buffer)))
      (incf (buffer-current-size buffer))
      (setf (svref (buffer-current buffer) pos) elt))))

(defun buffer->vector (buffer)
  "Returns simple-vector of the buffer elements."
  (declare (optimize (speed 3) (safety 0)))
  (let ((vector (make-array (buffer-size buffer)))
        (i 0)
        (list-count (buffer-list-size buffer)))
    (declare (type fixnum i list-count))
    (dolist (v (reverse (buffer-list buffer)))
      (let ((vsize (if (zerop (decf list-count))
                     (buffer-current-size buffer)
                     (buffer-inc-size buffer))))
        (dotimes (vpos vsize)
          (setf (svref vector i) (svref v vpos))
          (incf i))))
    vector))

(defmacro dobuffer ((var buffer &optional (result nil)) &body body)
  (let ((vec (gensym))
        (vec-size (gensym))
        (vec-pos (gensym))
        (list-count (gensym))
        (buf (gensym)))
    `(let* ((,buf ,buffer)
            (,list-count (buffer-list-size ,buf)))
       (dolist (,vec (reverse (buffer-list ,buf)) ,result)
         (let ((,vec-size (if (zerop (decf ,list-count))
                            (buffer-current-size ,buf)
                            (buffer-inc-size ,buf))))
           (dotimes (,vec-pos ,vec-size)
             (let ((,var (svref ,vec ,vec-pos)))
               ,@body)))))))
```

## A.3 lsystem.lisp

```
;;; *** lsystem.lisp ***
;;;
;;; This file is part of L-Lisp by Knut Arild Erstad.
;;; Contains the basic L-systems framework, including:
;;; - L-system rewriting
;;; - Homomorphism/decomposition rewriting
;;; - Context matching
;;; - Macros for productions and context sensitivity

(in-package :l-systems)

;; CMUCL does not define RUN-SHELL-COMMAND:
#+cmu
(defun run-shell-command (str)
  (ext:run-program "sh" (list "-c" str)
                   :output t))

(defclass l-system ()
  ;;"The base class for L-systems--not to be used directly."
  ((axiom
    :accessor axiom
    :initform nil
    :documentation "Axiom/initiator: sequence of modules/symbols")
   (depth
    :accessor depth
    :initform 0
    :documentation "Depth: (max) number of rewrites.")
   (angle-increment
    :accessor angle-increment
    :initform 90.0
    :documentation "Default angle increment (passed along to turtle).")
   (ignore-list
    :accessor ignore-list
    :initform nil
    :documentation "List of symbols to ignore in context.")
   (consider-list
    :accessor consider-list
    :initform nil
    :documentation
    "List of symbols to consider in context (overrides ignore-list).")
   (homomorphism-depth
    :accessor homomorphism-depth
    :initform 0
    :documentation "Max depth of the homomorphism tree.")
   (decomposition-depth
    :accessor decomposition-depth
    :initform 0
    :documentation "Max depth of the decomposition tree.")
   (sensitive
    :accessor sensitive
    :initform t
    :documentation "Boolean: set to nil to optimize non-environmentally
sensitivite L-systems.")
   ;; The rewriting strings RSTRING and HSTRING (vectors, actually) should
   ;; not be set explicitly.  Neither should GEOMETRY.
   (rstring
```

```
    :accessor rstring
    :initform nil
    :documentation "Rewrite string: vector of modules.")
   (hstring
    :accessor hstring
    :initform nil
    :documentation "Homomorphism rewriting string: vector of modules.")
   (geometry
    :accessor geometry
    :initform nil
    :documentation "3D Geometry created by turtle interpretation.")
   (current-depth
    :accessor current-depth
    :initform nil
    :documentation "The number of the current/latest rewrite.")
   ;; Graphical view/animation parameters (for OpenGL)
   (line-style
    :accessor line-style
    :initform :lines
    :documentation "Either :LINES or :CYLINDERS.")
   (cylinder-width
    :accessor cylinder-width
    :initform 1.0d0
    :documentation "Width multiplier for cylinders.")
   (cylinder-slices
    :accessor cylinder-slices
    :initform 8
    :documentation "Number of slices around cylinders.")
   (limits
    :accessor limits
    :initform nil
    :documentation "List of two positions vectors/sequences.")
   (frame-delay
    :accessor frame-delay
    :initform 0.5
    :documentation "Delay between animation frames, in seconds.")
   (frame-list
    :accessor frame-list
    :initform nil
    :documentation "List of frames and frame intervals (nested lists).")
   ;; The rest of the slots are for internal use
   (current-module
    :accessor current-module
    :initform nil
    :documentation "Current module: used internally during rewriting.")
   (pos
    :accessor pos
    :type fixnum
    :initform 0
    :documentation "Position: used internally during context checking.")
   (context-pos
    :accessor context-pos
    :type fixnum
    :initform 0
    :documentation "Used internally during context checking.")
   (warning-msg
    :accessor warning-msg
    :initform nil
    :documentation "Used internally to keep track of warnings.")
   ))

;; *** Methods that should be overridden ***
```

```
(defmethod l-productions ((ls l-system))
  "Override this when creating L-systems of greater depth than 0."
  t)

(defmethod homomorphism ((ls l-system))
  "Override this when creating L-systems with homomorphism."
  t)

(defmethod decomposition ((ls l-system))
  "Override this when creating L-systems with decomposition."
  t)

;; *** Helper functions ***
(defun add-bracket-params (vec)
  "Add positions of matching brackets as parameters."
  (declare (type simple-vector vec))
  (let ((bracket-positions nil))
    (dotimes (pos (length vec) vec)
      (declare (optimize (speed 3) (safety 0))
               (type fixnum pos))
      (let* ((module (svref vec pos))
             (symbol (if (consp module) (first module) module)))
        (case symbol
          (\[ (push pos bracket-positions))
          (\] (let ((bpos (pop bracket-positions)))
                (setf (svref vec bpos) (list '\[ pos)
                      (svref vec pos) (list '\] bpos))))))))))

(defun rlist->vector (rlist length)
  "Convert a reverse list RLIST of given length LENGTH to a simple-vector."
  (declare (optimize (speed 3) (safety 0))
           (fixnum length))
  (let ((vec (make-array length)))
    (dolist (elt rlist)
      (setf (svref vec (decf length)) elt))
    vec))

;; *** Rewriting ***
(defmethod rewrite ((ls l-system) &optional (depth (depth ls)))
  "Initialize the rewriting string and rewrite DEPTH times.
If DEPTH is 0, the rewriting string is initialized with the axiom,
but no rewriting is done."
  ;; intialize some slots
  (setf (geometry ls) nil)
  (setf (current-depth ls) 0)
  ;; init rstring with the axiom
  ;;(setf (rstring ls) (apply #'vector (axiom ls)))
  (setf (rstring ls) (map 'simple-vector #'identity (axiom ls)))
  (add-bracket-params (rstring ls))
  ;; for enviro-sensitive systems, create geometry
  (if (sensitive ls) (create-geometry ls))
  ;; do the rewrites
  (dotimes (i depth)
    (rewrite1 ls))
  ;;(if (sensitive ls) (create-geometry ls)))
  (rstring ls))

(defun skip-to-right-bracket (rstring pos)
  "Skip to next unmatched ']', and return the new position,
or NIL if there is no unmatched ']'."
  ;; Note: this could be optimized, but is probably fast enough
  (let ((i pos)
```

```
        (len (length rstring)))
    (loop
     (when (>= i len)
       (return nil))
     (let* ((module (svref rstring i))
            (symbol (if (consp module) (first module) module)))
       (cond ((eql symbol '\[)
              (setq i (1+ (skip-to-right-bracket rstring (1+ i)))))
             ((eql symbol '\])
              (return i))
             (t
              (incf i)))))))

(defmethod rewrite1 ((ls l-system))
  "Do a single rewrite of the L-system.
This includes a decomposition rewrite if DECOMPOSITION-DEPTH > 0."
  ;; If geometry exists, remove it
  (setf (geometry ls) nil)
  ;; Update current-depth
  (incf (current-depth ls))
  ;; Use a buffer to fill in new string
  (let* ((oldstr (rstring ls))
         ;;(strlength (length oldstr))
         (newstr (make-buffer)))
    (dotimes (pos (length oldstr))
      (declare (optimize (speed 3) (safety 0))
               (type simple-vector oldstr)
               (type fixnum pos))
      (setf (pos ls) pos)
      (let ((module (svref oldstr pos)))
        ;; check for cut symbol
        (when (member module '(\% :cut-symbol) :test #'eq)
          (let ((jump-pos (skip-to-right-bracket oldstr pos)))
            (if jump-pos
              (setf pos jump-pos
                    (pos ls) jump-pos
                    module (svref oldstr pos))
              (return))))
        (setf (current-module ls) module)
        (let ((answer (l-productions ls)))
          (if (eq answer t)
            ;; T means identity production (no change)
            (buffer-push module newstr)
            ;; otherwise we should have a list
            (dolist (obj answer)
              ;; add all non-nil objects to new rstring
              (unless (null obj)
                (buffer-push obj newstr)))))))
    ;; convert buffer to simple-vector
    (setf (rstring ls) (buffer->vector newstr))
    ;; set some other slots
    (setf (pos ls) 0
          (current-module ls) nil)
    ;; do decomposition if it exists
    (when (> (decomposition-depth ls) 0)
      (decomposition-rewrite ls))
    ;; add bracket parameters (for context checking in next step)
    (add-bracket-params (rstring ls))
    ;; create geometry if it is environmentally sensitive
    (when (sensitive ls)
      (create-geometry ls))
    ;; return new rstring
```

```
    (rstring ls)))

(defun tree-rewrite (ls production-func max-depth)
  "Common algorithm for homomorphism and decomposition (a bit like
Chomsky grammar rewriting)."
  (let* ((oldstr (rstring ls))
         (strlength (length oldstr))
         (newstring (make-buffer)))
    (labels ((expand-module (max-depth)
               (declare (optimize (speed 3) (safety 0))
                        (type fixnum max-depth)
                        (type compiled-function production-func))
               (if (= max-depth 0)
                 (progn (when (not (warning-msg ls))
                          (setf (warning-msg ls) "Maximum depth reached."))
                        (buffer-push (current-module ls) newstring))
                 (let ((x (funcall production-func ls)))
                   (if (eq x t)
                     (buffer-push (current-module ls) newstring)
                     (dolist (obj x)
                       (setf (current-module ls) obj)
                       (expand-module (1- max-depth))))))))
      (dotimes (pos strlength)
        (declare (optimize (speed 3) (safety 0))
                 (type fixnum pos))
        (setf (current-module ls) (svref oldstr pos))
        (expand-module max-depth)))
    (buffer->vector newstring)))

;; *** Homomorphism rewriting ***
(defmethod homomorphism-rewrite ((ls l-system))
  "Do a homomorphism (visual) rewrite from the current rewriting string."
  (let ((newstr (tree-rewrite ls #'homomorphism
                              (homomorphism-depth ls))))
    (when (warning-msg ls)
      (format t "Homomorphism warning: ~A" (warning-msg ls))
      (setf (warning-msg ls) nil))
    (setf (hstring ls) newstr)))

;; *** Decomposition rewriting ***
(defmethod decomposition-rewrite ((ls l-system))
  "Do a decomposition rewrite on the current rewriting string."
  (let ((newstr (tree-rewrite ls #'decomposition
                              (decomposition-depth ls))))
    (when (warning-msg ls)
      (format t "Decomposition warning: ~A" (warning-msg ls))
      (setf (warning-msg ls) nil))
    (setf (rstring ls) newstr)))

;; *** Context checking ***
(declaim (inline next-module-left))
(defun next-module-left (ls)
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (if (= (the fixnum (context-pos ls)) 0)
    :empty
    (svref (rstring ls) (decf (the fixnum (context-pos ls))))))

(defun next-context-module-left (ls ignore-list consider-list)
  "Find next context module to the left, ignoring symbols and skipping
brackets as necessary."
  (if consider-list
    (let ((new-consider-list (cons :empty consider-list)))
```

```
      (do* ((module (next-module-left ls)
                    (next-module-left ls))
            (symbol (if (consp module) (first module) module)
                    (if (consp module) (first module) module)))
          ((member symbol new-consider-list) module)
        (if (eq symbol '\]) ;; skip brackets
          (setf (context-pos ls) (second module)))))
    (let ((new-ignore-list (append '(\[ \]) ignore-list)))
      (do* ((module (next-module-left ls)
                    (next-module-left ls))
            (symbol (if (consp module) (first module) module)
                    (if (consp module) (first module) module)))
          ((not (member symbol new-ignore-list)) module)
        (if (eq symbol '\]) ;; skip brackets
          (setf (context-pos ls) (second module)))))))

(defun match-context-left (ls context-pattern ignore consider)
  "If the symbols and lengths in context-pattern match, return t and a
parameter list."
  (setf (context-pos ls) (pos ls))
  (let ((param-list nil))
    (dolist (obj (reverse context-pattern) (values t param-list))
      (let* ((module (next-context-module-left ls ignore consider))
             (params (if (consp module) (rest module) nil))
             (symbol (if params (first module) module)))
        (if (and (eql symbol (car obj)) (= (length params) (cdr obj)))
          (setf param-list (append params param-list))
          (return (values nil nil)))))))

(declaim (inline next-module-right))
(defun next-module-right (ls)
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (if (>= (the fixnum (context-pos ls))
          (the fixnum (1- (length (the simple-array (rstring ls))))))
    :empty
    (svref (rstring ls) (incf (the fixnum (context-pos ls))))))

(defun next-context-module-right (ls ignore-list consider-list)
  "Find next context module to the right, ignoring symbols as necessary,
but not skipping brackets."
  (if consider-list
    (let ((new-consider-list (append '(:empty \[ \]) consider-list)))
      (do* ((module (next-module-right ls)
                    (next-module-right ls))
            (symbol (if (consp module) (first module) module)
                    (if (consp module) (first module) module)))
          ((member symbol new-consider-list) module)))
    (do* ((module (next-module-right ls)
                  (next-module-right ls))
          (symbol (if (consp module) (first module) module)
                  (if (consp module) (first module) module)))
        ((not (member symbol ignore-list)) module))))

(defun match-context-right (ls context-pattern ignore consider)
  "If the symbols, lengths and brackets in context-pattern match, return a
parameter list and t."
  (setf (context-pos ls) (pos ls))
  (let ((param-list nil)
        (mstack nil))
    ;; local mstack function
    (flet ((mstack-empty? () (null mstack))
           (mstack-push (elt) (push elt mstack))
```

```
         (mstack-pop () (pop mstack)))
    (dolist (obj context-pattern (values t (nreverse param-list)))
      (do ((try-again t))
          ((not try-again))
        (let* ((module (next-context-module-right ls ignore consider))
               (params (if (consp module) (rest module) nil))
               (symbol (if params (first module) module)))
          (setq try-again nil)
          (cond ((eql (car obj) '\])
                 ;; skip to unmatched ], or fail if there is none
                 (if (mstack-empty?)
                   (return-from match-context-right (values nil nil))
                   (setf (context-pos ls) (mstack-pop))))
                ((eql (car obj) '\[)
                 ;; if brackets match, then push end position on
                 ;; mstack, else fail
                 (if (eql symbol '\[)
                   (mstack-push (first params))
                   (return-from match-context-right (values nil nil))))
                ((and (eql symbol (car obj))
                      (= (length params) (cdr obj)))
                 ;; found matching module, add params to list
                 (dolist (param params) (push param param-list)))
                ((eql symbol '\[)
                 ;; no match, but skip brackets and try again
                 (setq try-again t)
                 (setf (context-pos ls) (first params)))
                (t ;; no match, no brackets, fail
                 (return-from match-context-right (values nil nil))))))))))

;; *** Geometry methods ***
(defmethod create-geometry ((ls l-system))
  "Create geometry from current rewriting string."
  (let ((turtle-string (if (> (homomorphism-depth ls) 0)
                           (homomorphism-rewrite ls)
                           (rstring ls))))
    (setf (geometry ls)
      (turtle-interpret turtle-string
                        :angle-increment (angle-increment ls)))))

(defmethod rewrite-and-preview ((ls l-system) filename
                                &key
                                (depth (depth ls))
                                (width 0.3)
                                (sphere-width 0.1))
  "Rewrite, create geometry, output EPS file and view it in Ghostview."
  (rewrite ls depth)
  (if (null (geometry ls)) (create-geometry ls))
  (output-simple-eps (geometry ls) filename
                     :ps-width width
                     :sphere-width sphere-width)
  (run-shell-command (format nil "gv ~A -scale 10" filename)))

(defmethod rewrite-and-raytrace ((ls l-system) filename
                                 &key
                                 (depth (depth ls))
                                 (width (cylinder-width ls)))
  "Rewrite, create geometry, output POV file and raytrace it."
  (rewrite ls depth)
  (if (null (geometry ls)) (create-geometry ls))
  (output-povray (geometry ls) filename :width-multiplier width)
  (run-shell-command (format nil "povray +i~A" filename)))
```

```
(defmethod timed-raytrace ((ls l-system)
                           &key (filename "foo.pov")
                                (depth (depth ls))
                                (width-multiplier 0.01))
  (format t "~%REWRITE:~%")
  (time (rewrite ls depth))
  (format t "~%CREATE-GEOMETRY:~%")
  (if (geometry ls)
    (write-line "(already exists)")
    (time (create-geometry ls)))
  (format t "~%OUTPUT-POVRAY:~%")
  (time (output-povray (geometry ls) filename
                       :width-multiplier width-multiplier))
  (format t "~%Raytracing:~%")
  (time (run-shell-command (format nil "povray +i~A" filename))))

(defmethod timed-preview ((ls l-system)
                          &key (filename "foo.eps")
                               (depth (depth ls))
                               (width-multiplier 0.2))
  (format t "~%REWRITE:~%")
  (time (rewrite ls depth))
  (format t "~%CREATE-GEOMETRY:~%")
  (if (geometry ls)
    (write-line "(already exists)")
    (time (create-geometry ls)))
  (format t "~%OUTPUT-SIMPLE-EPS:~%")
  (time (output-simple-eps (geometry ls) filename
                           :ps-width width-multiplier))
  (format t "~%Ghostview:~%")
  (time (run-shell-command (format nil "gv ~A -scale 10" filename))))

(defun lsys2eps (classname filename &key (depth nil)
                                         (width-multiplier 0.3))
  (let ((lsys (make-instance classname)))
    (rewrite lsys (or depth (depth lsys)))
    (if (null (geometry lsys)) (create-geometry lsys))
    (output-simple-eps (geometry lsys) filename
                       :ps-width width-multiplier)))

(defun lsys2pov (classname filename &key (depth nil)
                                         (width-multiplier 0.02))
  (let ((lsys (make-instance classname)))
    (rewrite lsys (or depth (depth lsys)))
    (if (null (geometry lsys)) (create-geometry lsys))
    (output-povray (geometry lsys) filename
                   :width-multiplier width-multiplier)))

;; *** Animations ***

;; extracting frame numbers from a list of numbers and intervals
;; each element is a number, a (FROM TO) interval or a (FROM TO STEP) interval
;; ((1 3) 5) => (1 2 3 5)
;; (1 (20 50 10)) => (1 20 30 40 50)
(defun next-framelist (list)
  (let ((elt (first list))
        (rest (rest list)))
    (if (atom elt)
        (values elt rest)
        (let* ((step (if (null (cddr elt)) 1 (third elt)))
               (new-num (min (+ step (first elt))
```

```
                            (second elt)))
              (newelt (list new-num (second elt) step)))
         (if (>= (first newelt) (second newelt))
             (values (first elt) (cons (first newelt) rest))
             (values (first elt) (cons newelt rest)))))))

(defun top-of-framelist (list)
  (let ((x (car list)))
    (cond ((atom x) x)
          ((atom (car x)) (car x))
          (t (error "Invalid framelist ~A." list)))))

(defmacro extract-framelist (list)
  (let ((num-sym (gensym))
        (newlist-sym (gensym)))
    `(multiple-value-bind (,num-sym ,newlist-sym)
         (next-framelist ,list)
       (setf ,list ,newlist-sym)
       ,num-sym)))

(defmethod eps-animation ((ls l-system)
                          &key
                          (filename-prefix (string-downcase
                                            (class-name (class-of ls))))
                          (frames (or (frame-list ls)
                                      `((0 ,(depth ls)))))
                          (border-percent 10.0)
                          )
  (rewrite ls 0)
  (do* ((framelist (copy-tree frames)))
       ((null framelist) :done)
    (let* ((frame (extract-framelist framelist))
           (filename (format nil "~A~A.eps" filename-prefix frame)))
      (loop until (<= frame (current-depth ls))
            do (rewrite1 ls))
      (create-geometry ls)
      (format t "Outputting '~A'..." filename)
      (force-output)
      (output-simple-eps (geometry ls) filename
                         :border-percent border-percent)
      (format t "done.~%"))))

(defmethod povray-animation ((ls l-system)
                             &key
                             (filename-prefix (string-downcase
                                               (class-name (class-of ls))))
                             (frames (or (frame-list ls)
                                         `((0 ,(depth ls)))))
                             (width (cylinder-width ls))
                             (full-scene t)
                             )
  (rewrite ls 0)
  (do* ((framelist (copy-tree frames)))
       ((null framelist) :done)
    (let* ((frame (extract-framelist framelist))
           (filename (format nil "~A~A.pov" filename-prefix frame)))
      (loop until (<= frame (current-depth ls))
            do (rewrite1 ls))
      (create-geometry ls)
      (format t "Outputting '~A'..." filename)
      (force-output)
      (output-povray (geometry ls) filename
```

```
                     :width-multiplier width
                     :full-scene full-scene)
      (format t "done.~%"))))


;; *** Macros ***
(defun expand-prods-body (expr ls blockname)
  "Helper function for the CHOOSE-PRODUCTION macro; makes the necessary
changes to the WITH-*-CONTEXT and --> macros within the expression."
  (cond ((atom expr) expr)
        ;; expand with-contexts with ls
        ((member (first expr)
                 '(with-left-context with-right-context with-lc with-rc))
         `(,(first expr) ,ls ,@(expand-prods-body (rest expr) ls blockname)))
        ;; expand --> to return-->
        ((eq (first expr) '-->)
         `(return-from ,blockname
            (--> ,@(expand-prods-body (rest expr) ls blockname))))
        ;; don't expand nested choose-production expressions
        ((eq (first expr) 'choose-production)
         expr)
        ;; keep searching
        (t (cons (expand-prods-body (first expr) ls blockname)
                 (expand-prods-body (rest expr) ls blockname)))))

(defmacro prod (module module-form &rest body)
  "This macro makes some testing and variable-binding automatic.
Used by CHOOSE-PRODUCTION, don't call it directly."
  (if (consp module-form)
    `(when (and (consp ,module)
                (eq (first ,module) ',(first module-form))
                (= (length ,module) ,(length module-form)))
       (destructuring-bind ,(rest module-form)
           (rest ,module)
         (declare (ignorable ,@(rest module-form)))
         ,@body))
    `(when (eq ',module-form ,module)
       ,@body)))

(defmacro choose-production (ls &rest prods)
  "Choose and evaluate the first matching production.
Productions are given as: (sym expr-returning-list)
or with parameters:       ((sym param-1 ... param-n) expr-returning-list)"
  (let* ((ls-sym (gensym))
         (module-sym (gensym))
         (blockname (gensym))
         (expanded-prods (expand-prods-body prods ls-sym blockname)))
    `(block ,blockname
       (let* ((,ls-sym ,ls)
              (,module-sym (current-module ,ls-sym)))
         (or ,@(mapcar #'(lambda (x) (if (consp x)
                                         `(prod ,module-sym ,@x)
                                         `(prod ,module-sym ,x)))
                       expanded-prods)
             t)))))

(defmacro --> (&rest args)
  "Macro which can be used to specify right hand productions with
parameters as expressions.  Use this within CHOOSE-PRODUCTIONS."
  `(list ,@(mapcar #'(lambda (x) (if (consp x)
                                     `(list ',(car x) ,@(cdr x))
                                     `',x))
```

```
                  args)))

(defmacro stochastic-choice (&rest args)
  (let ((sym-list (mapcar #'(lambda (x) (declare (ignore x)) (gensym))
                          args))
        (random-sym (gensym))
        (n (length args)))
    `(let* ((,(first sym-list) ,(caar args))
            ,@(maplist
               #'(lambda (x y) `(,(second x) (+ ,(first x) ,(caar y))))
               sym-list (rest args))
            (,random-sym (random ,(nth (1- n) sym-list))))
       (cond ,@(mapcar
                #'(lambda (x y z) (declare (ignore z))
                          `((< ,random-sym ,x) ,(second y)))
                sym-list args (rest sym-list))
             (t ,(second (nth (1- n) args)))))))

(defmacro with-context (ls match-func-name context-form &body body)
  (let ((context-params-sym (gensym))
        (bool-sym (gensym))
        (bind-params nil)
        (context-pattern (mapcar #'(lambda (x)
                                     (let* ((params
                                             (if (consp x) (rest x) nil))
                                            (symbol
                                             (if params (first x) x)))
                                       (cons symbol (length params))))
                                 context-form)))
    (dolist (obj context-form)
      (when (consp obj) (setf bind-params (append bind-params (rest obj)))))
    (if bind-params
        `(multiple-value-bind (,bool-sym ,context-params-sym)
             (,match-func-name ,ls ',context-pattern
                               (ignore-list ,ls) (consider-list ,ls))
           (when ,bool-sym
             (destructuring-bind ,bind-params ,context-params-sym
               (declare (ignorable ,@bind-params))
               ,@body)))
        `(when (,match-func-name ,ls ',context-pattern
                                 (ignore-list ,ls) (consider-list ,ls))
           ,@body))))

(defmacro with-left-context (ls context-form &body body)
  `(with-context ,ls match-context-left ,context-form ,@body))

(defmacro with-right-context (ls context-form &body body)
  `(with-context ,ls match-context-right ,context-form ,@body))

(defmacro with-lc (&rest args)
  `(with-left-context ,@args))

(defmacro with-rc (&rest args)
  `(with-right-context ,@args))

;; ignore/consider macros
(defmacro while-ignoring (ls ignore-list &body body)
  (let ((save-ignore (gensym))
        (save-consider (gensym)))
    `(let ((,save-ignore (ignore-list ,ls))
           (,save-consider (consider-list ,ls)))
       (unwind-protect
```

```
          (progn (setf (consider-list ,ls) nil
                       (ignore-list ,ls) ',ignore-list)
                 ,@body)
        (setf (consider-list ,ls) ,save-consider
              (ignore-list ,ls) ,save-ignore)))))

(defmacro while-considering (ls consider-list &body body)
  (let ((save-consider (gensym)))
    `(let ((,save-consider (consider-list ,ls)))
       (unwind-protect
           (progn (setf (consider-list ,ls) ',consider-list)
                  ,@body)
         (setf (consider-list ,ls) ,save-consider)))))
```

## A.4 turtle.lisp

```
;;; *** turtle.lisp ***
;;;
;;; This file is part of L-Lisp by Knut Arild Erstad.
;;; Contains routines for:
;;; - 3D vector/matrix functions for double-floats.
;;; - Turtle graphics and interpretation.
;;; - Output geometry to Postscript and Povray

(in-package :l-systems)

;; make sure pi is double-float, no larger
(defconstant d-pi (coerce pi 'double-float))

;; Helper trigonometric functions
(defun deg-to-rad (n)
  (* n (/ d-pi 180.0d0)))

(defun rad-to-deg (n)
  (* n (/ 180.0d0 d-pi)))

(defun cosd (n)
  (cos (deg-to-rad n)))

(defun sind (n)
  (sin (deg-to-rad n)))

(defun tand (n)
  (tan (deg-to-rad n)))

;; Helper 3d vector/matrix functions
(declaim (inline vec3 vlength normalize equalvec zerovec vec- vec+
                 dot-product cross-product))

(defun v3 (elt1 elt2 elt3)
  "Create a 3D vector.  Slow, but works for all real numbers."
  (make-array 3 :element-type 'double-float
              :initial-contents
              (list (coerce elt1 'double-float)
                    (coerce elt2 'double-float)
                    (coerce elt3 'double-float))))

(defun vec3 (elt1 elt2 elt3)
  "Create a 3D vector.  All arguments must be double-floats."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
```

```
          (type double-float elt1 elt2 elt3))
  (let ((vec (make-array 3 :element-type 'double-float)))
    (declare (type (simple-array double-float (3)) vec))
    (setf (aref vec 0) elt1)
    (setf (aref vec 1) elt2)
    (setf (aref vec 2) elt3)
    vec))

(defun vlength (vec)
  "Calculate the length of a 3D vector."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec))
  (let ((sum 0.0d0))
    (declare (double-float sum))
    (dotimes (i 3)
      (let ((elt (aref vec i)))
        (declare (double-float elt))
        (incf sum (* elt elt))))
    (the double-float (sqrt sum))))

(defun normalize (vec)
  "Change the length of the 3D vector to 1."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec))
  (let ((len (vlength vec)))
    (declare (double-float len))
    (dotimes (i 3 vec)
      (setf (aref vec i) (/ (aref vec i) len)))))

(defun equalvec (vec1 vec2)
  "Returns T if two 3D vectors are equal."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec1 vec2))
  (and (= (aref vec1 0) (aref vec2 0))
       (= (aref vec1 1) (aref vec2 1))
       (= (aref vec1 2) (aref vec2 2))))

(defun almost-equalvec (vec1 vec2)
  "Returns T if two 3D vectors are almost equal; this is defined to be
true if all distances along axes are smaller than a certain epsilon.
Epsilon = 1.0d-10."
  (declare (optimize (speed 3) (safety 0))
           (type (simple-array double-float (3)) vec1 vec2))
  (let ((epsilon 1d-10))
    (and (< (abs (- (aref vec1 0) (aref vec2 0))) epsilon)
         (< (abs (- (aref vec1 1) (aref vec2 1))) epsilon)
         (< (abs (- (aref vec1 2) (aref vec2 2))) epsilon))))

(defun zerovec (vec)
  "Returns T if the vector is equal to #(0 0 0)."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec))
  (and (zerop (aref vec 0))
       (zerop (aref vec 1))
       (zerop (aref vec 2))))

(defun vec- (vec1 vec2)
  "Returns VEC1 - VEC2."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec1 vec2))
  (let ((vec (make-array 3 :element-type 'double-float)))
    (dotimes (i 3 vec)
```

```
      (setf (aref vec i) (- (aref vec1 i) (aref vec2 i))))))

(defun vec+ (vec1 vec2)
  "Returns VEC1 + VEC2."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec1 vec2))
  (let ((vec (make-array 3 :element-type 'double-float)))
    (dotimes (i 3 vec)
      (setf (aref vec i) (+ (aref vec1 i) (aref vec2 i))))))

(defun dot-product (vec1 vec2)
  "Returns the dot product VEC1 * VEC2."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec1 vec2))
  (let ((sum 0.0d0))
    (declare (double-float sum))
    (dotimes (i 3 sum)
      (incf sum (* (aref vec1 i) (aref vec2 i))))))

(defun cross-product (vec1 vec2)
  "Returns the cross product VEC1 x VEC2."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3)) vec1 vec2))
  (let ((u1 (aref vec1 0))
        (u2 (aref vec1 1))
        (u3 (aref vec1 2))
        (v1 (aref vec2 0))
        (v2 (aref vec2 1))
        (v3 (aref vec2 2))
        (r (make-array 3 :element-type 'double-float)))
    (declare (double-float u1 u2 u3 v1 v2 v3)
             (type (simple-array double-float (3)) r))
    (setf (aref r 0) (- (* u2 v3) (* u3 v2)))
    (setf (aref r 1) (- (* u3 v1) (* u1 v3)))
    (setf (aref r 2) (- (* u1 v2) (* u2 v1)))
    r))

(defun matrix-vector-mult (mat vec)
  "Multiply a 3x3 (rotation) matrix with a vector."
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type (simple-array double-float (3 3)) mat)
           (type (simple-array double-float (3)) vec))
  (let ((r (make-array 3 :element-type 'double-float
                       :initial-element 0.0d0)))
    (declare (type (simple-array double-float (3)) r))
    (dotimes (j 3 r)
      (dotimes (i 3)
        (incf (aref r j) (* (aref mat j i) (aref vec i)))))))

;; turtle struct; a "moving coordinate system" with some extra parameters
(defstruct (turtle (:copier nil))
  (pos (make-array 3 :element-type 'double-float
                   :initial-contents '#(0.0d0 0.0d0 0.0d0))
       :type (simple-array double-float (3)))
  (H (make-array 3 :element-type 'double-float
                 :initial-contents '#(0.0d0 1.0d0 0.0d0))
     :type (simple-array double-float (3)))
  (L (make-array 3 :element-type 'double-float
                 :initial-contents '#(-1.0d0 0.0d0 0.0d0))
     :type (simple-array double-float (3)))
  (U (make-array 3 :element-type 'double-float
                 :initial-contents '#(0.0d0 0.0d0 -1.0d0))
```

```
    :type (simple-array double-float (3)))
  angle
  width
  prev-width
  color
  texture
  (scale 1.0d0) ;; unused
  (shared-polygon-stack (list nil))
  (shared-mesh (list nil))
  (produce-spheres t)
  )


;; Unlike the other turtle values, the polygon and mesh stuff must be
;; shared between copies.  Some accessors to make this easier:
(defun turtle-polygon-stack (turtle)
  (car (turtle-shared-polygon-stack turtle)))

(defun (setf turtle-polygon-stack) (val turtle)
  (setf (car (turtle-shared-polygon-stack turtle)) val))

(defun turtle-polygon (turtle)
  (car (turtle-polygon-stack turtle)))

(defun (setf turtle-polygon) (val turtle)
  (setf (car (turtle-polygon-stack turtle)) val))

(defun turtle-mesh (turtle)
  (car (turtle-shared-mesh turtle)))

(defun (setf turtle-mesh) (val turtle)
  (setf (car (turtle-shared-mesh turtle)) val))

(defun copy-turtle (turtle)
  "Deep copy of turtle."
  (make-turtle :pos (copy-seq (turtle-pos turtle))
               :H (copy-seq (turtle-H turtle))
               :L (copy-seq (turtle-L turtle))
               :U (copy-seq (turtle-U turtle))
               :angle (turtle-angle turtle)
               :width (turtle-width turtle)
               :prev-width (turtle-prev-width turtle)
               :color (turtle-color turtle)
               :texture (turtle-texture turtle)
               :scale (turtle-scale turtle)
               :shared-polygon-stack
               (turtle-shared-polygon-stack turtle)
               :shared-mesh (turtle-shared-mesh turtle)
               :produce-spheres (turtle-produce-spheres turtle)
               ))

(defun rotate (turtle vec angle)
  "Rotate turtle an angle around a unit vector."
  ;; the algorithm is taken from the comp.graphics.algorithms FAQ:
  ;; "How do I rotate a 3D point?" (works for vectors, too)
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type turtle turtle)
           (type (simple-array double-float (3)) vec))
  (let* ((rad-angle (deg-to-rad angle))
         (t-ang (/ rad-angle 2.0d0))
         (cost (cos t-ang))
         (sint (sin t-ang))
```

```
         ;; [ x y z w ] quaternation
         (x (* (aref vec 0) sint))
         (y (* (aref vec 1) sint))
         (z (* (aref vec 2) sint))
         (w cost)
         ;; multiples of x, y, z, w
         (wx (* w x)) (wy (* w y)) (wz (* w z))
         (xx (* x x)) (xy (* x y)) (xz (* x z))
         (yy (* y y)) (yz (* y z)) (zz (* z z))
         ;; rotation matrix
         (mat (make-array '(3 3) :element-type 'double-float)))
    (declare (double-float rad-angle t-ang cost sint x y z w
                           wx wy wz xx xy xz yy yz zz)
             (type (simple-array double-float (3 3)) mat))
    ;; fill in matrix
    (setf (aref mat 0 0) (- 1.0d0 (* 2.0d0 (+ yy zz)))
          (aref mat 0 1) (* 2.0d0 (- xy wz))
          (aref mat 0 2) (* 2.0d0 (+ xz wy))
          (aref mat 1 0) (* 2.0d0 (+ xy wz))
          (aref mat 1 1) (- 1.0d0 (* 2.0d0 (+ xx zz)))
          (aref mat 1 2) (* 2.0d0 (- yz wx))
          (aref mat 2 0) (* 2.0d0 (- xz wy))
          (aref mat 2 1) (* 2.0d0 (+ yz wx))
          (aref mat 2 2) (- 1.0d0 (* 2.0d0 (+ xx yy))))
    ;; rotate the three turtle vectors
    (setf (turtle-H turtle) (matrix-vector-mult mat (turtle-H turtle))
          (turtle-U turtle) (matrix-vector-mult mat (turtle-U turtle))
          (turtle-L turtle) (matrix-vector-mult mat (turtle-L turtle)))
    nil))

(defun rotate-towards (turtle vec angle)
  "Rotate the turtle an angle towards a vector."
  ;;(declare (optimize (speed 3) (safety 0)))
  ;;(declare (type (simple-array double-float (3)) vec))
  ;; Find current angle between vec and turtle heading
  (let* ((hvec (turtle-H turtle))
         (dot (dot-product hvec vec))
         (current-rad-angle (acos (/ dot (vlength vec) (vlength hvec))))
         (current-deg-angle (rad-to-deg current-rad-angle))
         (rotate-angle (min angle current-deg-angle)))
    ;;(declare (type (simple-array double-float (3)) hvec))
    ;;(declare (double-float dot current-rad-angle current-deg-angle))
    (when (> rotate-angle 0d0)
      (let ((rvec (cross-product hvec vec)))
        (when (zerovec rvec)
          (let ((newvec (copy-seq vec)))
            (incf (aref newvec 0) 1d-10)
            (setq rvec (cross-product hvec newvec))))
        (rotate turtle (normalize rvec) angle)))))

;; The definitions of turn, pitch and roll has been optimized
;; to avoid slow matrix multiplication.
;; Pencil and paper were used to calculate formulas. :-)
(defun turn (turtle angle)
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type turtle turtle))
  (let* ((rad-angle (deg-to-rad (coerce angle 'double-float)))
         (cosa (cos rad-angle))
         (sina (sin rad-angle))
         (H (turtle-H turtle))
         (L (turtle-L turtle)))
    (declare (double-float rad-angle cosa sina)
```

```
             (type (simple-array double-float (3)) H L))
    (dotimes (i 3)
      (let ((hi (aref H i))
            (li (aref L i)))
        (declare (double-float hi li))
        (setf (aref H i) (- (* hi cosa) (* li sina))
              (aref L i) (+ (* hi sina) (* li cosa)))))))

(defun pitch (turtle angle)
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type turtle turtle))
  (let* ((rad-angle (deg-to-rad (coerce angle 'double-float)))
         (cosa (cos rad-angle))
         (sina (sin rad-angle))
         (H (turtle-H turtle))
         (U (turtle-U turtle)))
    (declare (double-float rad-angle cosa sina)
             (type (simple-array double-float (3)) H U))
    (dotimes (i 3)
      (let ((hi (aref H i))
            (ui (aref U i)))
        (declare (double-float hi ui))
        (setf (aref H i) (+ (* hi cosa) (* ui sina))
              (aref U i) (- (* ui cosa) (* hi sina)))))))

(defun roll (turtle angle)
  (declare (optimize (speed 3) (safety 0) #+cmu (ext:inhibit-warnings 3))
           (type turtle turtle))
  (let* ((rad-angle (deg-to-rad (coerce angle 'double-float)))
         (cosa (cos rad-angle))
         (sina (sin rad-angle))
         (L (turtle-L turtle))
         (U (turtle-U turtle)))
    (declare (double-float rad-angle cosa sina)
             (type (simple-array double-float (3)) L U))
    (dotimes (i 3)
      (let ((li (aref L i))
            (ui (aref U i)))
        (declare (double-float li ui))
        (setf (aref L i) (+ (* li cosa) (* ui sina))
              (aref U i) (- (* ui cosa) (* li sina)))))))

(defun move-forward (turtle &optional (distance 1.0d0))
  (let ((pos (turtle-pos turtle))
        (h (turtle-H turtle)))
    (dotimes (i 3 pos)
      (incf (aref pos i) (* distance (aref h i))))))

;; line struct, for 3d geometry
;; (line/segment/cylinder depending on context)
(defstruct line
  (p1 (make-array 3 :element-type 'double-float
                  :initial-contents '(0.0d0 0.0d0 0.0d0))
      :type (simple-array double-float (3)))
  (p2 (make-array 3 :element-type 'double-float
                  :initial-contents '(0.0d0 0.0d0 0.0d0))
      :type (simple-array double-float (3)))
  width
  prev-width
  sphere
  color
  texture)
```

```
;; sphere struct
(defstruct sphere
  pos
  radius
  color
  texture)

;; polygon struct
(defstruct polygon
  points ;; list of points
  normal ;; normal vector
  color
  texture)

;; triangle mesh
(defstruct mesh
  vertices  ;; list of unique vertices
  triangles ;; list of triangles
  (strands (list nil)) ;; nested list of vertices used during contruction
  color
  texture)

(defstruct vertex
  pos       ;; 3d position
  normal)   ;; average of triangles' normal vectors

(defstruct triangle
  vertices ;; three of them
  normal)  ;; normal vector

;; box struct (independent of turtle orientation)
(defstruct box
  pos     ;; #(x y z) position of near lower left corner
  size    ;; #(xsize ysize zsize)
  color
  texture)

(defun mesh-add-vertex (mesh point)
  "Add new vertex to mesh and return it, or if one exists at the same
position, return that one."
  (let ((vertex nil))
    (dolist (obj (mesh-vertices mesh))
      (if (almost-equalvec point (vertex-pos obj))
        (setf vertex obj)))
    (unless vertex
      (setf vertex (make-vertex :pos (copy-seq point)
                                :normal
                                (make-array 3 :element-type 'double-float
                                            :initial-element 0d0)))
      (push vertex (mesh-vertices mesh)))
    (let* ((strands (mesh-strands mesh))
           (strand (first strands)))
      (if (or (null strand)
              (not (eql vertex (first strand))))
        (push vertex (first strands))))
    vertex))

(defun mesh-add-triangle (mesh vertex1 vertex2 vertex3)
  "If all vertices are distinct, create a triangle, add it to the mesh
and return it.  Otherwise, return NIL."
  (if (or (eql vertex1 vertex2)
```

```
          (eql vertex1 vertex3)
          (eql vertex2 vertex3))
    nil
    (let* ((pos1 (vertex-pos vertex1))
           (pos2 (vertex-pos vertex2))
           (pos3 (vertex-pos vertex3))
           (v1 (vec- pos2 pos1))
           (v2 (vec- pos3 pos1))
           (normal (normalize (cross-product v1 v2)))
           (triangle (make-triangle
                      :vertices (list vertex1 vertex2 vertex3)
                      :normal normal)))
      ;; add normal to vertices
      (dolist (vertex (list vertex1 vertex2 vertex3))
        (let ((vnormal (vertex-normal vertex)))
          (dotimes (i 3)
            (incf (aref vnormal i) (aref normal i)))))
      ;; add it to the mesh
      (push triangle (mesh-triangles mesh)))))

;; *** Turtle functions ***
(defparameter *turtle-functions*
  (make-hash-table :test 'eq :size 211))

(defmacro def-turtle-function-raw (names parameters &body body)
  "This macro is for defining turtle functions.  The NAMES is
either a single name or a list of names of the turtle command.
The function is stored in the *TURTLE-FUNCTIONS* hash table.
All non-NIL return values will be added to the geometry vector during
turtle interpretation.

The first parameter is the turtle and the second is the list of
parameters.  The elements of this list can be changed, which is unseful
for environmentally sensitive commands.
"
  `(let ((func (lambda ,parameters ,@body)))
    ,@(mapcar (lambda (x)
                `(setf (gethash ',x *turtle-functions*) func))
              (if (atom names) (list names) names))))

(defmacro def-turtle-function (names parameters &body body)
  "This macro is for defining turtle functions.  The NAMES is
either a single name or a list of names of the turtle command.
The function is stored in the *TURTLE-FUNCTIONS* hash table.
All non-NIL return values will be added to the geometry vector during
turtle interpretation.

The first parameter is the turtle (and is required), the rest of the
parameters are module parameters, which can be declared &OPTIONAL.
A &REST parameter is added (and declared to be ignored) unless you supply
one explicitly.

  (def-turtle-command (:foo s) (turtle &optional (angle 90.0))
    (do-something turtle angle)
    nil)
"
  (let* ((temp-param (gensym))
         (junk-param (gensym))
         (turtle-param (first parameters))
         (has-rest (find '&rest parameters))
         (lambda-func (if has-rest
                          `(lambda (,turtle-param &optional ,temp-param)
```

```
                            (destructuring-bind ,(rest parameters)
                                ,temp-param
                              ,@body))
                          `(lambda (,turtle-param &optional ,temp-param)
                            (destructuring-bind (,@(rest parameters)
                                                 &rest ,junk-param)
                                ,temp-param
                              (declare (ignore ,junk-param))
                              ,@body)))))
    `(let ((func ,lambda-func))
      ,@(mapcar (lambda (x)
                  `(setf (gethash ',x *turtle-functions*) func))
                (if (atom names) (list names) names)))))

(defun turtle-function (name)
  (gethash name *turtle-functions*))

;;
;; *** Predefined turtle functions ***
;;

(def-turtle-function (:forward F) (turtle &optional (length 1.0))
  (let ((oldpos (copy-seq (turtle-pos turtle))))
    (move-forward turtle length)
    (let ((line (make-line :p1 oldpos
                           :p2 (copy-seq (turtle-pos turtle))
                           :color (turtle-color turtle)
                           :texture (turtle-texture turtle)
                           :width (turtle-width turtle)
                           :prev-width (turtle-prev-width turtle)
                           :sphere (turtle-produce-spheres turtle))))
      ;; since we moved forward, set prev-width to width
      (setf (turtle-prev-width turtle) (turtle-width turtle))
      line)))

(def-turtle-function :produce-spheres (turtle produce-spheres)
  (setf (turtle-produce-spheres turtle) produce-spheres)
  nil)

(def-turtle-function (:forward-no-line \f) (turtle &optional (length 1.0))
  (move-forward turtle length)
  nil)

(def-turtle-function (:turn-left +)
    (turtle &optional (angle (turtle-angle turtle)))
  (turn turtle (- angle))
  nil)

(def-turtle-function (:turn-right -)
    (turtle &optional (angle (turtle-angle turtle)))
  (turn turtle angle)
  nil)

(def-turtle-function (:pitch-down &)
    (turtle &optional (angle (turtle-angle turtle)))
  (pitch turtle angle)
  nil)

(def-turtle-function (:pitch-up ^)
    (turtle &optional (angle (turtle-angle turtle)))
  (pitch turtle (- angle))
  nil)
```

```
(def-turtle-function (:roll-left \\)
    (turtle &optional (angle (turtle-angle turtle)))
  (roll turtle angle)
  nil)

(def-turtle-function (:roll-right /)
    (turtle &optional (angle (turtle-angle turtle)))
  (roll turtle (- angle))
  nil)

(def-turtle-function (:turn-around \|) (turtle)
  (let ((H (turtle-H turtle))
        (L (turtle-L turtle)))
    (dotimes (i 3)
      (setf (aref H i) (- (aref H i))
            (aref L i) (- (aref L i))))))

(def-turtle-function (:set-width !) (turtle width) ; required parameter
  (setf (turtle-prev-width turtle) (turtle-width turtle))
  (setf (turtle-width turtle) width)
  nil)

(def-turtle-function (:sphere @O)
    (turtle &optional (radius (turtle-width turtle)))
  (make-sphere :pos (copy-seq (turtle-pos turtle))
               :radius (coerce radius 'double-float)
               :color (turtle-color turtle)
               :texture (turtle-texture turtle)))

(def-turtle-function :box (turtle in-pos in-size)
  (let ((pos (map '(simple-array double-float (3))
                  (lambda (x) (coerce x 'double-float))
                  in-pos))
        (size (map '(simple-array double-float (3))
                   (lambda (x) (coerce x 'double-float))
                   in-size)))
    (make-box :pos pos :size size
              :color (turtle-color turtle)
              :texture (turtle-texture turtle))))

;; Polygon functions
(def-turtle-function (:start-polygon {) (turtle)
  (push (make-polygon :points (list (copy-seq (turtle-pos turtle)))
                      :color (turtle-color turtle)
                      :texture (turtle-texture turtle))
        (turtle-polygon-stack turtle))
  nil)

(def-turtle-function (:add-vertex \.) (turtle)
  (let* ((polygon (turtle-polygon turtle))
         (last-vertex (first (polygon-points polygon)))
         (pos (turtle-pos turtle)))
    (if (not (equalp pos last-vertex))
      (push (copy-seq pos)
            (polygon-points polygon))))
  nil)

(def-turtle-function (:forward-vertex f.) (turtle &optional (length 1.0))
  (move-forward turtle length)
  (funcall (turtle-function :add-vertex) turtle)
  nil)
```

```
(def-turtle-function (:end-polygon }) (turtle)
  (funcall (turtle-function :add-vertex) turtle)
  ;; save normal
  (let* ((polygon (turtle-polygon turtle))
         (points (polygon-points polygon))
         (p1 (first points))
         (p2 (second points))
         (p3 (third points))
         (v1 (vec- p2 p1))
         (v2 (vec- p3 p1))
         (normal (cross-product v1 v2)))
    (setf (polygon-normal polygon) (normalize normal)))
  ;; pop polygon
  (pop (turtle-polygon-stack turtle)))

(def-turtle-function :color (turtle in-color)
  ;; prepare color for OpenGL's glColor or glMaterial (use single-floats)
  ;; array [ r g b 1.0 ]
  (let ((color (make-array 4 :element-type 'single-float)))
    (setf (aref color 3) 1.0)
    (dotimes (i 3)
      (setf (aref color i) (coerce (aref in-color i) 'single-float)))
    (setf (turtle-color turtle) color))
  nil)

(def-turtle-function :texture (turtle texture)
  (setf (turtle-texture turtle) texture)
  nil)

;; Mesh functions
(def-turtle-function (:start-mesh m{) (turtle)
  (let ((mesh (make-mesh)))
    (mesh-add-vertex mesh (turtle-pos turtle))
    (setf (turtle-mesh turtle) mesh))
  nil)

(def-turtle-function (:mesh-vertex m.) (turtle)
  (mesh-add-vertex (turtle-mesh turtle) (turtle-pos turtle))
  nil)

(def-turtle-function (:new-strand m/) (turtle)
  (let ((mesh (turtle-mesh turtle)))
    ;; start an empty strand
    (push nil (mesh-strands mesh))
    ;; add current position
    (mesh-add-vertex mesh (turtle-pos turtle)))
  nil)

(defun strands-to-triangles (mesh strand1 strand2)
  (when (and strand1 strand2)
    (mapc (lambda (x y z) (mesh-add-triangle mesh x y z))
          strand1 strand2 (rest strand2))
    (mapc (lambda (x y z) (mesh-add-triangle mesh x y z))
          strand1 (rest strand2) (rest strand1))))

(def-turtle-function (:end-mesh m}) (turtle)
  (let* ((mesh (turtle-mesh turtle))
         (strands (mesh-strands mesh)))
    ;; create triangles
    (if (first strands)
      (mapc #'(lambda (x y) (strands-to-triangles mesh x y))
```

```
            strands (cdr strands)))
    ;; calculate average normal vectors
    (dolist (vertex (mesh-vertices mesh))
      (let ((normal (vertex-normal vertex)))
        (unless (zerovec normal)
          (normalize normal))))
    ;; set color
    (setf (mesh-color mesh) (turtle-color turtle))
    ;; mesh is finished, delete redundant fields
    (setf (mesh-vertices mesh) nil)
    (setf (mesh-strands mesh) nil)
    ;; return mesh
    mesh))

(def-turtle-function (:forward-mesh-vertex mf) (turtle &optional (length 1.0))
  (move-forward turtle length)
  (funcall (turtle-function :mesh-vertex) turtle)
  nil)

;; Set position
(def-turtle-function (:set-position @M) (turtle pos)
  (setf (turtle-pos turtle)
        (map '(simple-array double-float 1)
             (lambda (x) (coerce x 'double-float))
             pos))
  nil)

;; Rotate toward a vector
(def-turtle-function (:rotate-towards @R)
    (turtle angle &optional (vec #(0d0 1d0 0d0)))
  (let ((d-angle (coerce angle 'double-float))
        (d-vec (map '(vector double-float)
                    (lambda (x) (coerce x 'double-float))
                    vec)))
    (rotate-towards turtle d-vec d-angle)
    nil))

;; Create lines independently of turtle position
(def-turtle-function :line (turtle p1 p2
                                   &optional (width (turtle-width turtle)))
  (make-line :p1 (map '(simple-array double-float (3))
                      (lambda (x) (coerce x 'double-float))
                      p1)
             :p2 (map '(simple-array double-float (3))
                      (lambda (x) (coerce x 'double-float))
                      p2)
             :width width
             :color (turtle-color turtle)
             :texture (turtle-texture turtle)))

;; Environmentally sensitive functions
(def-turtle-function-raw (:get-position ?P) (turtle params)
  (setf (first params) (copy-seq (turtle-pos turtle)))
  nil)

(def-turtle-function-raw (:get-heading ?H) (turtle params)
  (setf (first params) (copy-seq (turtle-H turtle)))
  nil)

(def-turtle-function-raw (:get-up-vector ?U) (turtle params)
  (setf (first params) (copy-seq (turtle-U turtle)))
  nil)
```

```
(def-turtle-function-raw (:get-left-vector ?L) (turtle params)
  (setf (first params) (copy-seq (turtle-L turtle)))
  nil)

(def-turtle-function-raw (:get-turtle ?T) (turtle params)
  (setf (first params) (copy-turtle turtle))
  nil)

;; *** Turtle interpretation ***
(defun turtle-interpret (rstring &key (angle-increment 90.0d0))
  "Create geometry by turtle-interpreting a rewriting string."
  (declare (optimize (speed 3) (safety 0))
           (type simple-vector rstring))
  (let ((geometry (make-buffer))
        (turtle-stack nil)
        (turtle (make-turtle :angle angle-increment)))
    (push turtle turtle-stack)
    (dotimes (pos (length rstring))
      (let* ((module (svref rstring pos))
             (params (if (listp module) (rest module) nil))
             (symbol (if params (first module) module)))
        (cond
         ((eq symbol '\[)
          (setq turtle (copy-turtle turtle))
          (push turtle turtle-stack))
         ((eq symbol '\])
          (pop turtle-stack)
          (setq turtle (car turtle-stack)))
         (t
          (let ((func (gethash symbol *turtle-functions*)))
            (when (functionp func)
              (let ((returnval (funcall func turtle params)))
                (when returnval
                  (buffer-push returnval geometry)))))))))
    (buffer->vector geometry)))

;; Geometry limits
(defun find-limits (geometry)
  "Find the min- and max-values of the geometry coordinates."
  (let* ((bg most-positive-double-float)
         (sm most-negative-double-float)
         (minv (vector bg bg bg))
         (maxv (vector sm sm sm)))
    (dotimes (pos (length geometry))
      (let ((elt (aref geometry pos)))
        (case (type-of elt)
          (line
           (let* ((p1 (line-p1 elt))
                  (p2 (line-p2 elt)))
             (dotimes (i 3)
               (setf (aref minv i) (min (aref minv i)
                                        (aref p1 i)
                                        (aref p2 i)))
               (setf (aref maxv i) (max (aref maxv i)
                                        (aref p1 i)
                                        (aref p2 i))))))
          (polygon
           (dolist (p (polygon-points elt))
             (dotimes (i 3)
               (setf (aref minv i) (min (aref minv i)
                                        (aref p i)))
```

```
              (setf (aref maxv i) (max (aref maxv i)
                                       (aref p i))))))
          (box
           (let* ((p1 (box-pos elt))
                  (p2 (vec+ p1 (box-size elt))))
             (dotimes (i 3)
               (setf (aref minv i) (min (aref minv i)
                                        (aref p1 i)
                                        (aref p2 i)))
               (setf (aref maxv i) (max (aref maxv i)
                                        (aref p1 i)
                                        (aref p2 i))))))
          (mesh
           (dolist (triangle (mesh-triangles elt))
             (dolist (vertex (triangle-vertices triangle))
               (let ((pos (vertex-pos vertex)))
                 (dotimes (i 3)
                   (setf (aref minv i) (min (aref minv i)
                                            (aref pos i)))
                   (setf (aref maxv i) (max (aref maxv i)
                                            (aref pos i))))))))))))
    (values minv maxv)))

(defun find-simple-limits (geometry &key (border-percent 10.0))
  "Find 2d limits (min, max) of 3d geometry, ignoring z coordinate."
  (multiple-value-bind (minv maxv) (find-limits geometry)
    (let* ((xmin (aref minv 0))
           (xmax (aref maxv 0))
           (ymin (aref minv 1))
           (ymax (aref maxv 1))
           (x (- xmax xmin))
           (y (- ymax ymin))
           (border (* (max x y) border-percent 0.01)))
      (decf xmin border)
      (decf ymin border)
      (incf xmax border)
      (incf ymax border)
      (values xmin xmax ymin ymax))))

;; *** Outputting Postscript ***
(defun output-simple-eps (geometry filename
                          &key
                          (ps-size 100) (ps-width 0.3) (border-percent 10.0)
                          (sphere-width 0.1))
  "Output simplest possible 2d projection (ignoring z coordinate) as EPS."
  (with-open-file (file filename :direction :output
                        :if-exists :supersede)
    (multiple-value-bind (xmin xmax ymin ymax)
        (find-simple-limits geometry :border-percent border-percent)
      (let* ((xsize (- xmax xmin))
             (ysize (- ymax ymin))
             (line-width 1.0)
             (ps-size-x ps-size)
             (ps-size-y ps-size))
        (if (> xsize ysize)
          (setq ps-size-y (floor (* ps-size ysize) xsize))
          (setq ps-size-x (floor (* ps-size xsize) ysize)))
        (format file "%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 ~A ~A
%%Creator: L-Lisp (L-systems for CL) by Knut Arild Erstad (knute@ii.uib.no)
%%EndComments~%" ps-size-x ps-size-y)
        (loop for obj across geometry do
```

```
              (case (type-of obj)
                (line
                 (let* ((line obj)
                        (p1 (line-p1 line))
                        (x1 (/ (- (aref p1 0) xmin) xsize))
                        (y1 (/ (- (aref p1 1) ymin) ysize))
                        (p2 (line-p2 line))
                        (x2 (/ (- (aref p2 0) xmin) xsize))
                        (y2 (/ (- (aref p2 1) ymin) ysize))
                        (w (line-width line)))
                   (when w
                     (setq line-width w))
                   (format file "~,9F setlinewidth ~,9F ~,9F moveto~%"
                           (* line-width ps-width)
                           (* x1 ps-size-x) (* y1 ps-size-y))
                   (format file "~,9F ~,9F lineto stroke~%"
                           (* x2 ps-size-x) (* y2 ps-size-y))))
                (polygon
                 (let* ((polygon obj)
                        (points (polygon-points polygon))
                        (fpoint (first points))
                        (xf (/ (- (aref fpoint 0) xmin) xsize))
                        (yf (/ (- (aref fpoint 1) ymin) ysize)))
                   (format file "0.01 setlinewidth~%")
                   (format file "newpath ~,9F ~,9F moveto~%"
                           (* xf ps-size-x) (* yf ps-size-y))
                   (dolist (point (rest points))
                     (let ((x (/ (- (aref point 0) xmin) xsize))
                           (y (/ (- (aref point 1) ymin) ysize)))
                       (format file "~,9F ~,9F lineto~%"
                               (* x ps-size-x) (* y ps-size-y))))
                   (format file "closepath fill~%")))
                (sphere
                 (let* ((sphere obj)
                        (pos (sphere-pos sphere))
                        (x (/ (- (aref pos 0) xmin) xsize))
                        (y (/ (- (aref pos 1) ymin) ysize))
                        (radius (sphere-radius sphere)))
                   (format file
                           "newpath ~,9F ~,9F ~,9F 0 360 arc closepath fill~%"
                           (* x ps-size-x) (* y ps-size-y)
                           (* radius sphere-width))))
                ))))))

;; *** Outputting POV code ***
(defmethod povcode (obj stream &rest junk)
  (declare (ignore junk stream))
  (warn "Unknown type ~A passed to POVCODE." (type-of obj))
  nil)

(defun povcode-vector-long (vec stream)
  (format stream "<~,15F,~,15F,~,15F>"
          (aref vec 0) (aref vec 1) (aref vec 2)))

(defun povcode-vector-short (vec stream)
  (format stream "<~,5F,~,5F,~,5F>"
          (aref vec 0) (aref vec 1) (aref vec 2)))

(defun povcode-color (colvec stream)
  (when colvec
    (princ "pigment{ color rgb " stream)
    (povcode-vector-short colvec stream)
```

```
    (princ " }" stream)))

(defmethod povcode ((line line) stream &rest args)
  (unless (equalvec (line-p1 line) (line-p2 line))
    (let* ((width (or (line-width line) 1.0))
           (prev-width (or (line-prev-width line) width))
           (width-multiplier (if args (first args) 1.0))
           (width-x-mult (* width width-multiplier))
           (sphere (line-sphere line)))
      (when sphere
        (format stream "union { "))
      (princ "cone{ " stream)
      (povcode-vector-short (line-p1 line) stream)
      (format stream ", ~,5F, " (* prev-width width-multiplier))
      (povcode-vector-short (line-p2 line) stream)
      (format stream ", ~,5F " width-x-mult)
      (when sphere
        (format stream "} sphere {")
        (povcode-vector-short (line-p2 line) stream)
        (format stream ", ~,5F} " width-x-mult))
      (povcode-color (line-color line) stream)
      (write-line "}" stream))))

(defun povcode-points (list stream)
  (povcode-vector-long (car list) stream)
  (dolist (elt (cdr list))
    (princ ", ")
    (povcode-vector-long elt stream)))


(defmethod povcode ((sphere sphere) stream &rest args)
  (let ((radius (or (sphere-radius sphere) 1.0))
        (width-multiplier (if args (first args) 1.0)))
    (princ "sphere { " stream)
    (povcode-vector-short (sphere-pos sphere) stream)
    (format stream ", ~,5F " (* radius width-multiplier))
    (povcode-color (sphere-color sphere) stream)
    (write-line "}" stream)))

(defmethod povcode ((polygon polygon) stream &rest junk)
  (declare (ignore junk))
  (let ((points (polygon-points polygon)))
    (format stream "polygon{ ~A, " (length points))
    (povcode-points points stream)
    (princ " " stream)
    (povcode-color (polygon-color polygon) stream)
    (write-line "}" stream)))

(defun povcode-smooth-triangle (triangle stream)
  (let* ((vertices (triangle-vertices triangle))
         (v1 (first vertices))
         (v2 (second vertices))
         (v3 (third vertices)))
    (princ "smooth_triangle { " stream)
    (povcode-vector-short (vertex-pos v1) stream)
    (princ ", " stream)
    (povcode-vector-short (vertex-normal v1) stream)
    (princ ", " stream)
    (povcode-vector-short (vertex-pos v2) stream)
    (princ ", " stream)
    (povcode-vector-short (vertex-normal v2) stream)
    (princ ", " stream)
```

```
    (povcode-vector-short (vertex-pos v3) stream)
    (princ ", " stream)
    (povcode-vector-short (vertex-normal v3) stream)
    (princ " }" stream)))

(defmethod povcode ((mesh mesh) stream &rest junk)
  (declare (ignore junk))
  (princ "mesh { " stream)
  (dolist (elt (mesh-triangles mesh))
    (povcode-smooth-triangle elt stream)
    (princ " " stream))
  (povcode-color (mesh-color mesh) stream)
  (write-line "}" stream))

(defmethod povcode ((box box) stream &rest junk)
  (declare (ignore junk))
  (princ "box { " stream)
  (let* ((pos (box-pos box))
         (pos2 (vec+ pos (box-size box))))
    (povcode-vector-short pos stream)
    (princ ", " stream)
    (povcode-vector-short pos2 stream))
  (povcode-color (box-color box) stream)
  (write-line "}" stream))

(defun output-povray (geometry filename
                      &key (object-name "Lsystem")
                           (full-scene t)
                           (width-multiplier 0.01))
  "Output 3d geometry as an object for the POV ray-tracer."
  (with-open-file (file filename :direction :output
                   :if-exists :supersede)
    (format file "// ~A object generated by L-Lisp L-system framework
// by Knut Arild Erstad (knute@ii.uib.no)~%" object-name)
    (when full-scene
      ;; Put some "dumb" defaults for camera and light source
      (format file "
#include \"colors.inc\"
#include \"woods.inc\"
background { color LightBlue }
camera {
  location <0.7, 1, -1>
  look_at <0, 0.5, 0>
}
light_source { <5, 50, -20> color White }
plane { <0,1,0>, 0 pigment {color White} finish {ambient .3} }~%"))
    (format file "#declare ~A = union {~%" object-name)
    (dotimes (i (length geometry))
      (povcode (aref geometry i) file width-multiplier))
    ;;(format file "~A~%" (povcode (aref geometry i) width-multiplier)))
    (when full-scene
      (multiple-value-bind (minv maxv) (find-limits geometry)
        (declare (ignore minv))
        (let ((yscale (/ 1.0 (aref maxv 1))))
          (format file "scale ~,6F~%" yscale))))
    (format file "} // end of ~A object~%" object-name)
    (when full-scene
      (format file "object { ~A texture {T_Wood1 finish {ambient .5}} }~%"
              object-name))))
```

# Glossary

**1L-system** : Context-sensitive L-system in which either the successor or the predecessor module is considered.

**2L-system** : Context-sensitive L-system in which both the successor and the predecessor module is considered.

**acropetal** : Information flow from the root towards the branches of a plant.

**alphabet** : Set of symbols that can be used in a specific L-system. The alphabet is usually not defined explicitly.

**auxiliary methods** : CLOS mechanism to augment methods by adding code that is called before, after or around a specific method. See also CLOS.

**axiom** : The initial rewriting string of an L-system.

**basipetal** : Information flow from the branches towards the root of a plant.

**bracketed L-system** : L-system which uses brackets to represent branches. The rewriting string can then represent a tree.

**circumnutation** : Near-circular movements of a growing stem.

**climbing plant** : Plant that can use surfaces or other plants for support.

**CLOS** : The Common Lisp Object System, a standard object-oriented framework for Common Lisp. CLOS has support for multi-methods and auxiliary methods, features lacking in many other object-oriented languages.

**CMU Common Lisp** : A public domain Common Lisp implementation. See Common Lisp.

**Common Lisp** : A high-level programming language that is multi-paradigm (supports object-oriented, imperative and functional programming) and is extremely flexible and extensible.

**context-sensitive L-system** : L-system in which each production can consider modules to the left or right of the source module.

**cpfg** : Continuous Plant and Fractal Generator. An L-system implementation (language) with many extensions.

**curvature** : A measurement of curving.

**decomposition** : Set of productions that are usually used for splitting modules into "smaller" modules. Decomposition productions are applied recursively.

**deterministic** : Deterministic L-systems always produce the same sequence of strings. All L-systems without stochastic productions are deterministic.

**DOL-system** : A deterministic, context-free L-system. (The "O" is actually the number zero, which says the neither left nor right contexts is considered.)

**elongation function** : A function describing the continuous elongation of a stem or root.

**Emacs** : An extensible text editor.

**environmentally sensitive L-system** : L-system that is affected by the environment.

**erasing production** : A production which produces an empty string of modules. Written as $X \rightarrow \epsilon$.

**Eulerian motion** : Description of continuous motion seen from a global perspective. See also Lagrangian motion.

**fractal** : A fractal can be loosely defined as a complex-looking image or model that:

1. is generated by a (relatively) simple mathematical formula
2. contains self-similarity; parts of the fractal can be scaled and reoriented so that they are similar to other parts

Attempts at more precise mathematical definitions have been made, but no such definition has managed to capture all images that are normally viewed as fractals. L-systems can be used to generate many fractals.

**generic function** : In CLOS, a function which can be specialized based upon the types of its parameters. Each specialized version is called a *method*.

**grammar** : Set of productions.

**homomorphism** : Set of productions which are usually used for converting modules to turtle commands. Homomorphism productions are applied recursively.

**IL-system** : Context-sensitive L-systems in which a fixed number of left and right context modules are considered.

**ILISP** : An Emacs interface to many dialects of Lisp. See Emacs, Lisp.

**initiator** : See *axiom*.

**keyword** : In Common Lisp, a symbol belonging to a special keyword package. Keywords are printed with a colon first, and always evaluate to themselves.

**kinematic** : Motion described as a continuous flow.

**L-Lisp** : An extensible L-systems framework in Common Lisp.

**L-system** : Lindenmayer system; a parallel rewriting system.

**Lagrangian motion** : Description of continuous motion seen from a specific particle's perspective. The description will be different depending on which particle we focus on. See also Eulerian motion.

**Lisp** : A family of programming languages. Originally, Lisp was spelled LISP which is an acronym for LISt Processor. The best known Lisp dialects in use today are Common Lisp, Scheme and special-purpose languages like Emacs Lisp or AutoLisp. See Common Lisp.

**macro** : In Lisp, a mechanism for arbitrary code transformations. Macros are the main reason why Lisp is so easy to extend.

**mesh** : See polygon mesh.

**method** : In CLOS, methods define operations on objects of specific types. In CLOS, unlike most other object-oriented languages, methods belong to generic functions rather than classes.

**module** : Symbol with a (possibly empty) list of parameters. A rewriting string generally consist of modules.

**open L-system** : L-system that interacts with the environment.

**OpenGL** : Portable 2D/3D graphics library.

**parameter** : Numeric value attached to a module.

**parametric L-system** : L-system in which each module of the rewriting string can have parameters.

**polygon mesh** : A three-dimensional geometric shape consisting of several connected polygons (usually triangles).

**polygon** : A two-dimensional geometric shape enclosed by three or more straight lines.

**production** : Rewriting rule which defines how a single L-system module is replaced by a string of modules.

**pruning** : Shaping of plants by cutting off branches.

**radius of curving** : A measurement of curving that equals $1/\kappa$, where $\kappa$ is the curvature.

**read-eval-print loop** : A loop used by many interactive programming languages. It reads an expression, evaluates it and prints the result.

**rewriting** : Transformation of an L-system string to another.

**slot** : In CLOS, a data field belonging to an object. See also CLOS.

**spline function** : A function defined by a set of points rather than a formula. Coordinates between points are calculated by an interpolation algorithm.

**stochastic L-system** : Non-deterministic L-system in which productions are chosen based upon a random distribution.

**string** : A sequence of modules.

**symbolic expression** : parenthesized lists of atoms (symbols, numbers and other non-list objects) or nested lists. Symbolic expressions are the base syntax for most languages in the Lisp family.

**symbol** : Identifier that is part of an L-system's alphabet. Some symbols are also names of turtle commands. In L-Lisp, L-system symbols are also Lisp symbols. See also module, turtle command.

**tropism** : Directional growth caused by the environment.

**turtle command** : A command to a turtle, such as "move forward and draw a line" or "turn 90 degrees to the left". See turtle graphics, turtle function.

**turtle function** : In L-Lisp, a function that implements a turtle command.

**turtle graphics** : Turtle graphics are created by simulating a local coordinate system ("turtle") that can move through a global coordinate system.

**twining plant** : Climbing plant that seeks support by twining around poles or other plants.

**vertex** : A point at either end of a polygon edge.

# Bibliography

[1] L. Baillaud. Les mouvements d'exploration et d'enroulement des plantes volubiles. *Handb. Pflanzenphysiol.*, 12: 635–708, 1962.

[2] I. A. Borovikov. L-systems with inheritance: an object-oriented extension of L-systems. *ACM SIGPLAN notices*, 30(5): 43–60. 1995.

[3] M. Boshernitsan. Design and implementation of tree-transformations in Ensemble, UCB CS Honors Program Project Report. 1997. http://www.cs.berkeley.edu/ maratb/cs264/paper/paper.html

[4] R. O. Erickson and K. B. Sax. Elemental growth rate of the primary root of Zea Mays. *Proceedings of the American Philosophical Society*, 100(5): 487–498, 1956.

[5] R. O. Erickson and K. B. Sax. Rates of cell division and cell elongation in the growth of the primary root of *Zea Mays. Proceedings of the American Philosophical Society*, 100(5): 499–514, 1956.

[6] R. O. Erickson and W. K. Silk. The kinematics of plant growth. *Scientific American*, 242(5): 134–151, 1980.

[7] P. W. Gandar. Growth in root apices. I. The kinematic description of growth. *Botanical Gazette*, 144(1): 1–10, 1983.

[8] N. S. Goel and I. Rozehnal. A high-level Language for L-systems and its applications. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: impact on theoretical computer science, computer graphics, and developmental biology*, 231–251. Springer-Verlag, Berlin, 1992.

[9] P. Graham. *On Lisp; Advanced techniques for Common Lisp.* Prentice-Hall, New Jersey, 1994.

[10] P. Graham. *ANSI Common Lisp.* Prentice-Hall, New Jersey, 1996.

[11] J. W. Hart. *Plant tropisms and other growth movements.* Unwin Hyman, London 1990.

[12] A. Lindenmayer. Mathematical Models for Cellular Interactions in Development. Parts I and II. *Journal of Theoratical Biology*, 18(1): 280–315, 1968.

[13] B. Luttik, E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing.* Springer-Verlag, Berlin, November 1997.

[14] R. Měch. *Modeling and Simulation of the Interaction of Plants with the Environment using L-systems and their Extensions.* Calgary, Alberta, 1997.

[15] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers, Inc. 1992.

[16] P. Prusinkiewicz. Graphical Applications of L-systems. *Proceedings of Graphics Interface '86 - Vision Interface '86*, 247–253, Vancouver, 1986.

[17] P. Prusinkiewicz, A. Lindenmayer et al. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York Inc. 1990.

[18] P. Prusinkiewicz, J. Hanan. L-Systems: from formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: impact on theoretical computer science, computer graphics, and developmental biology*, 193–211. Springer-Verlag, Berlin, 1992.

[19] P. Prusinkiewicz, M. Hammel, J. Hanan and R. Mech. Visual Models of Plant Development. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages, vol. III: Beyond words*, 535–597. Springer-Verlag, Berlin, 1997.

[20] P. Prusinkiewicz, J. Hanan and R. Mech. An L-system-based plant modeling language. In M. Nagl, A. Schürr, M. Münch, editors, *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE 1999 proceedings*, 395–410. Kerkrade, The Netherlands, 1999.

[21] W. K. Silk and S. Abou Haidar. Growth of the stem of *Pharbitis nil*: Analysys of longitudinal and radial components. *Physiologie Végétale*, 24(1): 109–116, 1986.

[22] W. K. Silk. Growth rate patterns which maintain a helical tissue tube. *Journal of Theoretical Biology*, 128: 311–327, 1989.

[23] W. K. Silk and M. Hubbard. Axial forces and normal distributed loads in twining stems of morning glory. *J. Biomechanics*, 24(7): 559–606, 1991.

[24] CMU Common Lisp—a public domain Common Lisp implementation.
http://www.cons.org/cmucl/

[25] ILISP—a generalized (X)Emacs interface to an underlying Lisp system.
http://ilisp.cons.org/

[26] The POV-Ray team. POV-Ray—the Persistence of Vision Raytracer.
http://www.povray.org/

[27] J. Clark. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999. http://www.w3.org/TR/xslt

# Index