Functional-Structural Plant Modelling with GroIMP and XL

Tutorial and Workshop at Agrocampus Ouest, Angers,
5-7 May, 2015

Winfried Kurth

University of Göttingen,
Department Ecoinformatics, Biometrics and Forest Growth

Introduction to rule-based programming, L-systems and the language XL

Paradigms of programming

Robert Floyd 1978:

Turing Award Lecture

"The Paradigms
of Programming"

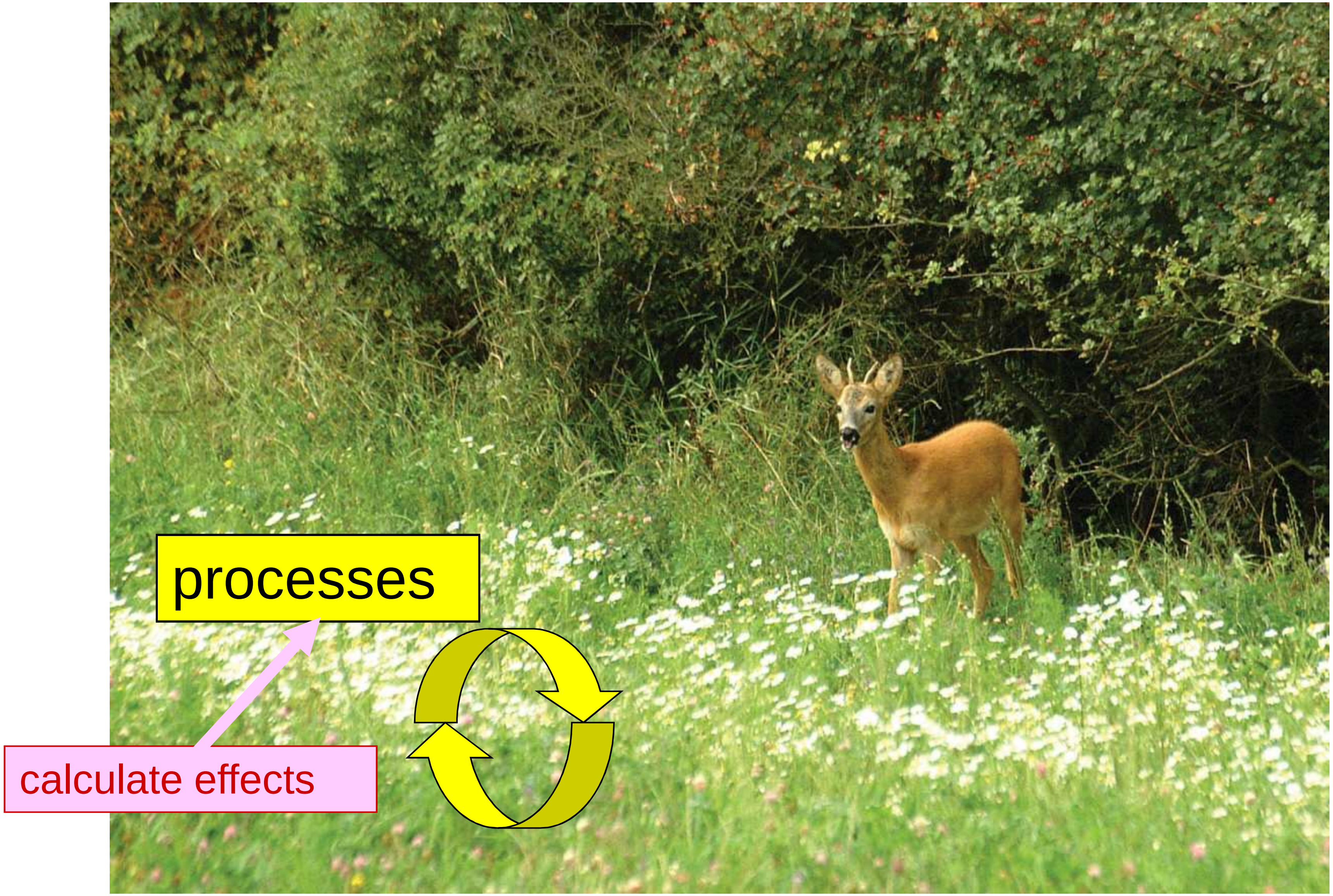
Robert W. Floyd (1936-2001)

Paradigms of programming

„Paradigm“: a basic model, a way of thinking,
or a philosophical approach towards reality,
often expressed by examples...

Usage for simulating an ecosystem:

Usage for simulating an ecosystem:



for numerical simulation of processes:

imperative paradigm
(also: von-Neumann paradigm,
control flow paradigm)

John von Neumann (1903-1957)

imperative programming:

computer = machine for the manipulation of values of variables

(these manipulations can have side effects).

programme = plan for the calculation process with specification of the commands and of the control flow (e.g. loops).

example:

```
x = 0;
while (x < 100)
   x = x + 1;
```

(one) drawback of the imperative paradigm:

simultaneous, parallel assignment is not supported

(one) drawback of the imperative paradigm:

simultaneous, parallel assignment is not supported

Example (Floyd 1978):

predator-prey system (population sizes A, B), described by

A_{new} = f(A, B),
B_{new} = g(A, B)

beginners' mistake in programming:

```
for (i = ... ) {
   A = f(A, B);
   B = g(A, B);
}
```

(one) drawback of the imperative paradigm:

simultaneous, parallel assignment is not supported

Example (Floyd 1978):

predator-prey system (population sizes A, B), described by

A_{new} = f(A, B),
B_{new} = g(A, B)

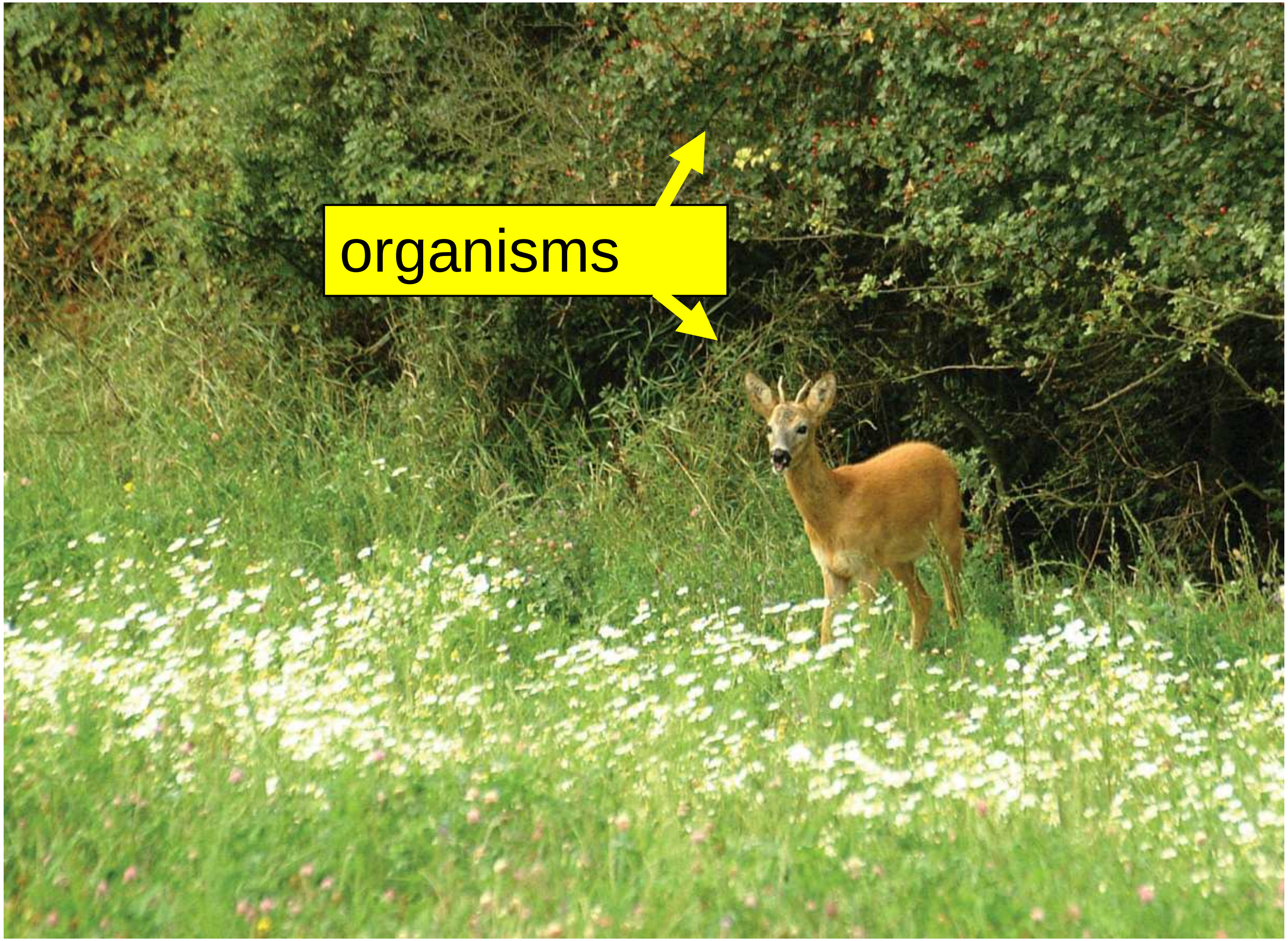
beginners' mistake in programming:

```
for (i = ... ) {
    A = f(A, B);
    B = g(A, B);
}
```

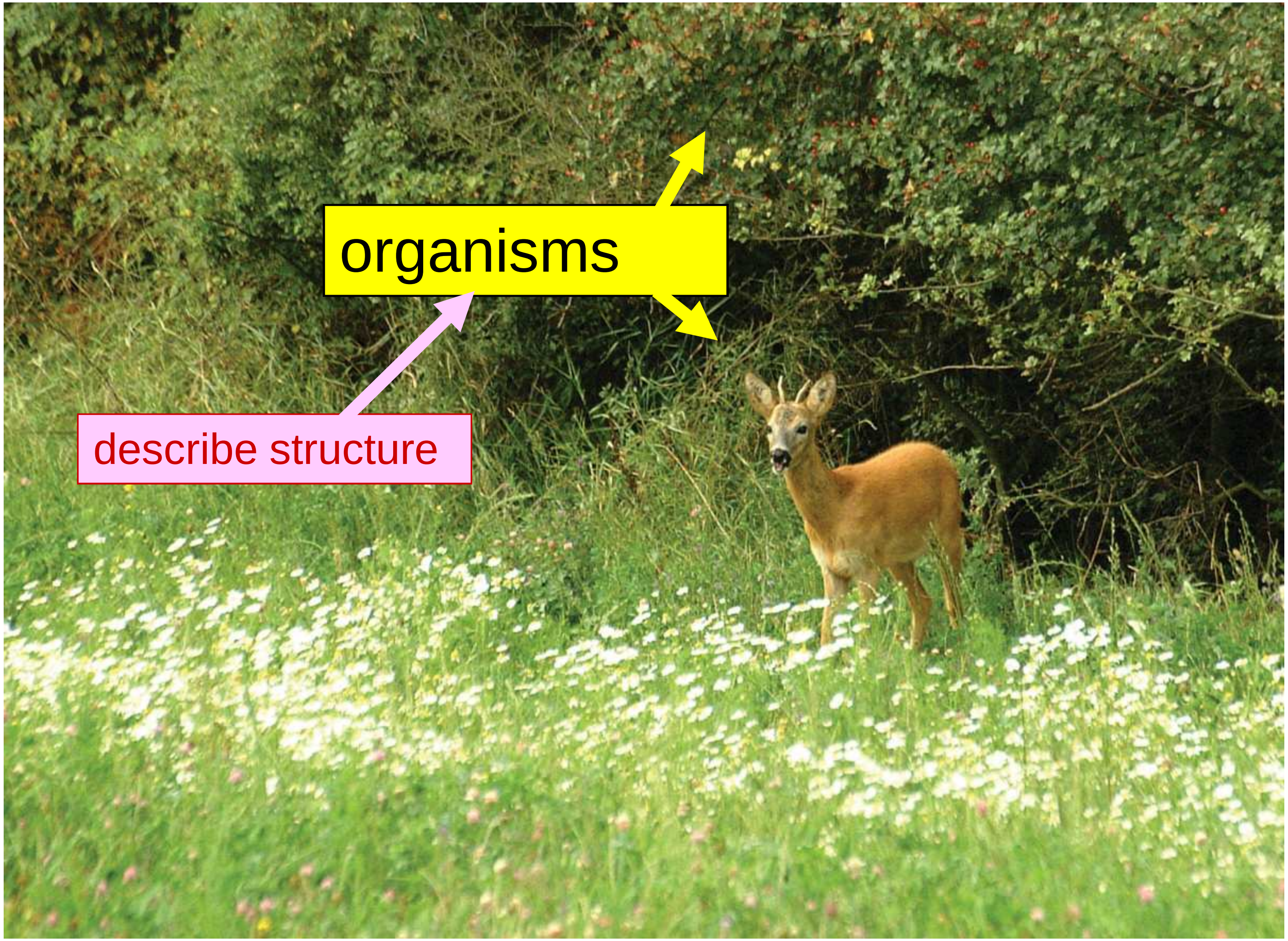
programming languages which support imperative programming:

Fortran, Pascal, C, ..., parts of Java, ...,
command language of turtle geometry

Simulating an ecosystem:



Simulating an ecosystem:



object-oriented paradigm

computer = environment for virtual objects

programme = list of (object) *classes*, i.e. general specifications of objects, which can be created and destroyed at runtime.

programming languages: Smalltalk, Simula, C++, Java, ...

example:

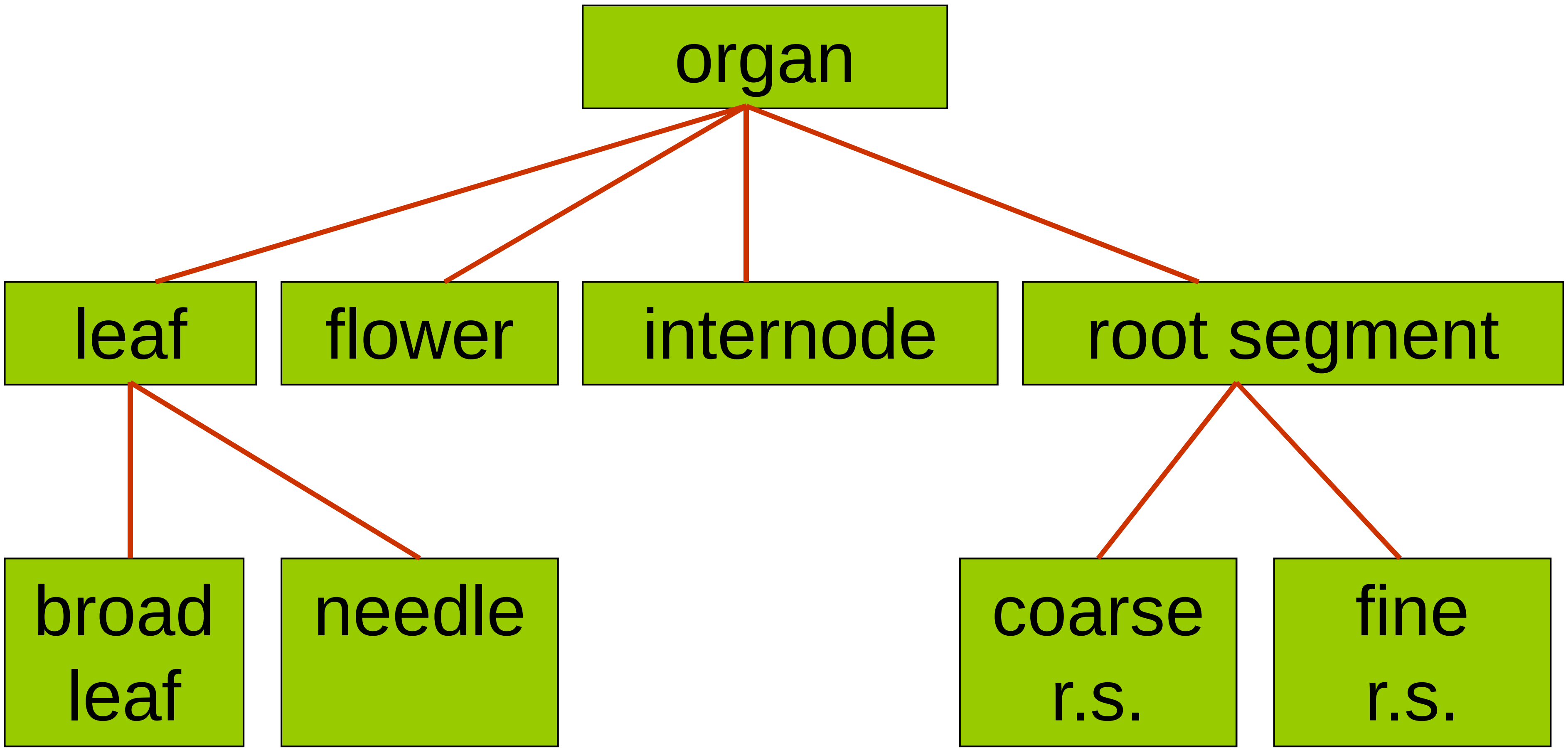
Inheritance of attributes and methods from superclasses to subclasses

```
public class Car extends Vehicle
   {
   public String name;
   public int places;
   public void show()
        {
        System.out.println("The car is a " + name);
        System.out.println("It has " + places + "places.");
        }
   }
```

typical:

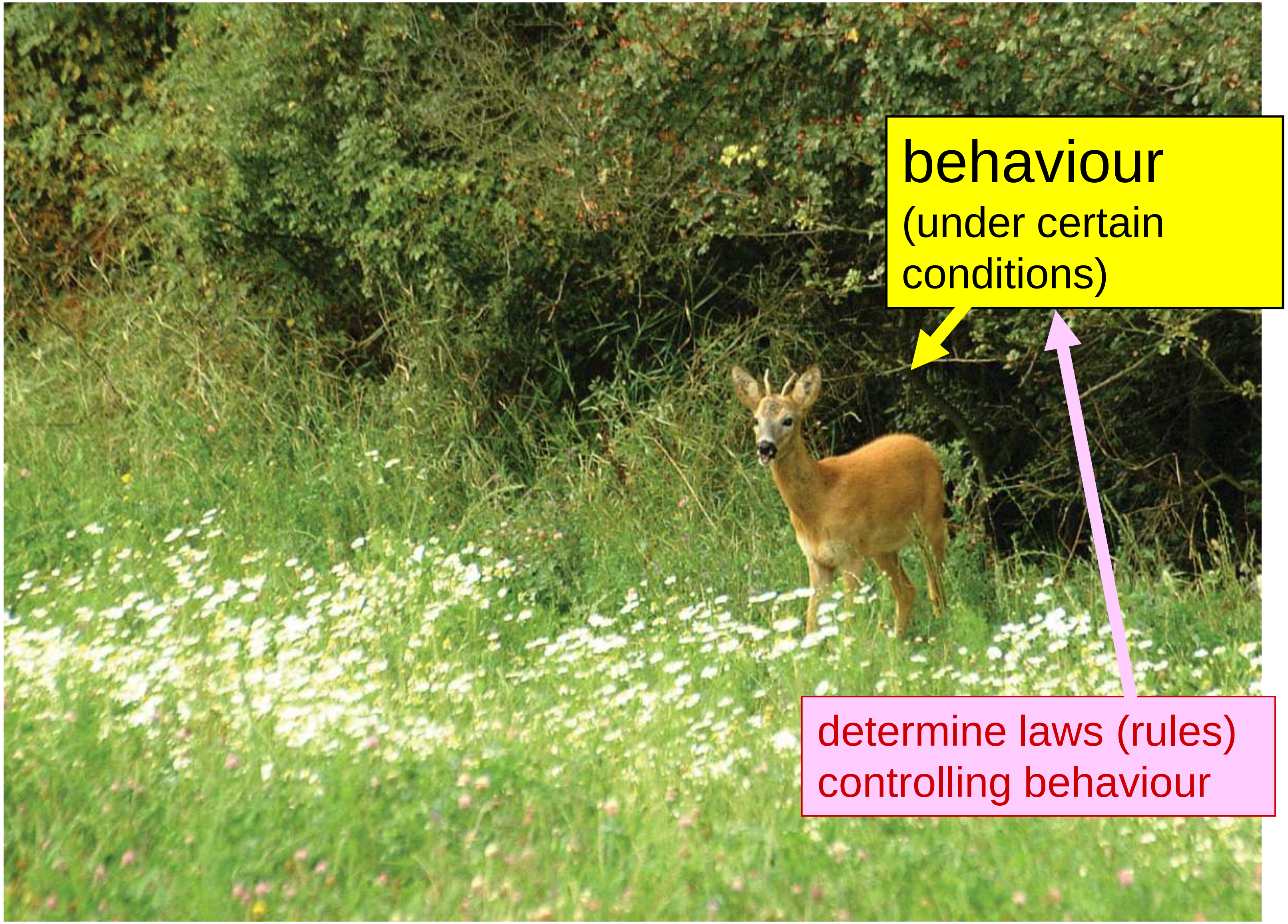
classes (**`Car`**) with data (**`name`**, **`places`**) and methods (**`show`**) usefulness of object hierarchies in biology

for example:



Simulating an ecosystem:

Simulating an ecosystem:



rule-based paradigm

computer = machine which transforms structures

There is a current structure (e.g., a graph) which is transformed as long as it is possible.

Work process: search and application.
matching: search for a suitable rule,
rewriting: application of the rule, thereby transformation of the structure.

rule-based paradigm

computer = machine which transforms structures

There is a current structure (e.g., a graph) which is transformed as long as it is possible.

Work process: search and application.
matching: search for a suitable rule,
rewriting: application of the rule, thereby transformation of the structure.

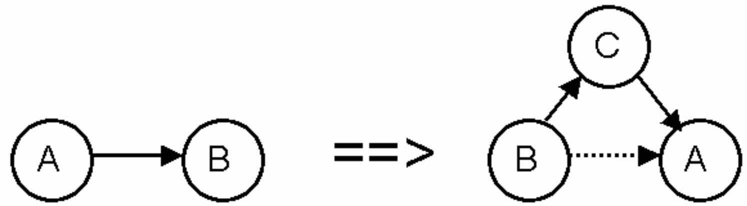
programme = set of transformation rules

to find a programme: specification of rules.

programming languages: L-system languages, AI languages, Prolog, ...

Example:

a graph grammar

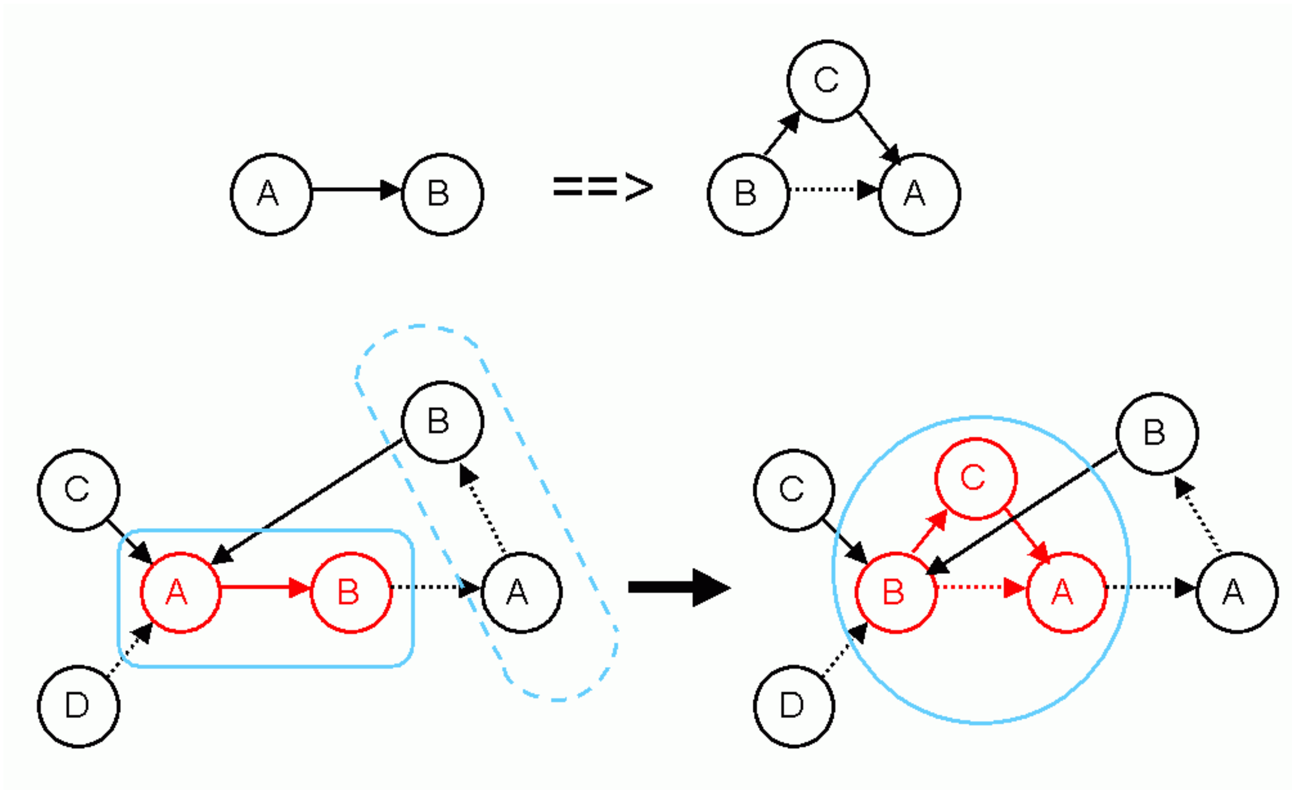
rule: 

Example:

a graph grammar

rule:

application:



Dynamical description of structures

L-systems (Lindenmayer systems)

rule systems for the replacement of character strings

in each derivation step *parallel* replacement of all characters for which there is one applicable rule

by A. Lindenmayer (botanist) introduced in 1968 to model growth of filamentous algae

Aristid Lindenmayer (1925-1989)

L-systems:

some results

L-systems mathematically:

a triple (Σ, α, *R*) with:

Σ a set of characters, the *alphabet*,

α a string with characters from Σ, the *start word* (also "Axiom"),

R a set of rules of the form

character $\rightarrow$ string of characters;

with the characters taken from Σ.

A *derivation step* (rewriting) of a string consists of the replacement of all of its characters which occur in left-hand sides of rules by the corresponding right-hand sides.

Convention: characters for which no rule is applicable stay as they are.

Result:

Derivation chain of strings, developed from the start word by iterated rewriting.

$$\alpha \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow ....$$

Example:

alphabet {A, B}, start word A

set of rules:

A $\longrightarrow$ B
B $\longrightarrow$ AB

A

Example:

alphabet {A, B}, start word A

set of rules:

A $\rightarrow$ B

B $\rightarrow$ AB

B

Example:

alphabet {A, B}, start word A

set of rules:

$A \rightarrow B$

$B \rightarrow AB$

AB

parallel replacement

Example:

alphabet {A, B}, start word A

set of rules:

A $\longrightarrow$ B

B $\longrightarrow$ AB

BAB

Example:

alphabet {A, B}, start word A

set of rules:

$A \longrightarrow B$

$B \longrightarrow AB$

BAB

Example:

alphabet {A, B}, start word A

set of rules:

$A \longrightarrow B$

$B \longrightarrow AB$

ABBAB

Example:

alphabet {A, B}, start word A

set of rules:

$A \longrightarrow B$

$B \longrightarrow AB$

derivation chain:

$A \longrightarrow B \longrightarrow AB \longrightarrow BAB \longrightarrow ABBAB \longrightarrow BABABBAB$

$\longrightarrow ABBABBABABBAB \longrightarrow BABABBABABBABBABABBAB$

$\longrightarrow \ldots$

Example:

alphabet {A, B}, start word A

set of rules:

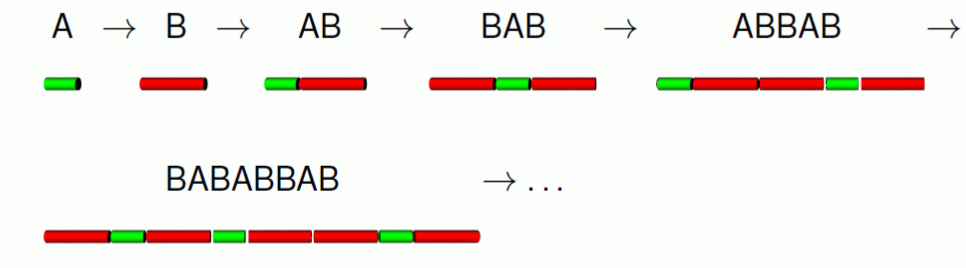
$A \longrightarrow B$

$B \longrightarrow AB$

derivation chain:

$A \longrightarrow B \longrightarrow AB \longrightarrow BAB \longrightarrow ABBAB \longrightarrow BABABBAB$
$\longrightarrow ABBABBABABBAB \longrightarrow BABABBABABBABBABABBAB$
$\longrightarrow \ldots$

geometrical visualization:



required for modelling biological structures in space:

a geometrical interpretation

Thus we add:

- a function which assigns to each string a subset of 3-D space

„interpreted“ L-system processing

$$
\begin{array}{ccccccccc}
\alpha & \rightarrow & \sigma_1 & \rightarrow & \sigma_2 & \rightarrow & \sigma_3 & \rightarrow & \ldots. \\
 & & \downarrow & & \downarrow & & \downarrow & & \\
 & & S_1 & & S_2 & & S_3 & & \ldots.
\end{array}
$$

S_1, S_2, S_3, ... can be seen as developmental steps of an object, a scene or an organism.

For the interpretation:

turtle geometry

the turtle command set becomes a subset of the character set of the L-system.

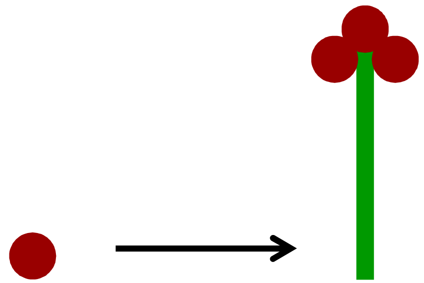
Symbols which are not turtle commands are ignored by the turtle.

$\longrightarrow$ connection with imperative paradigm

A “botanical“ example:

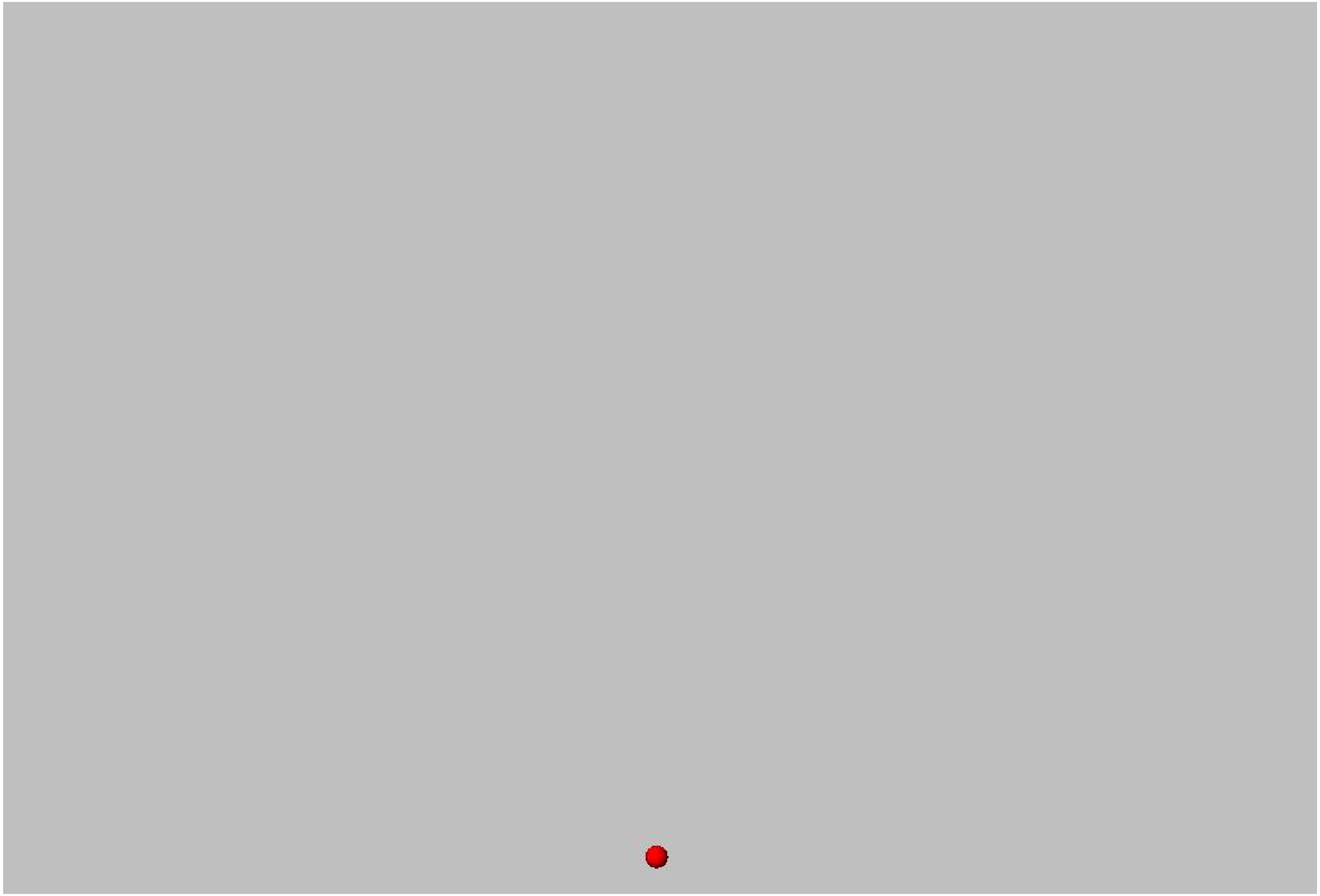
rule:

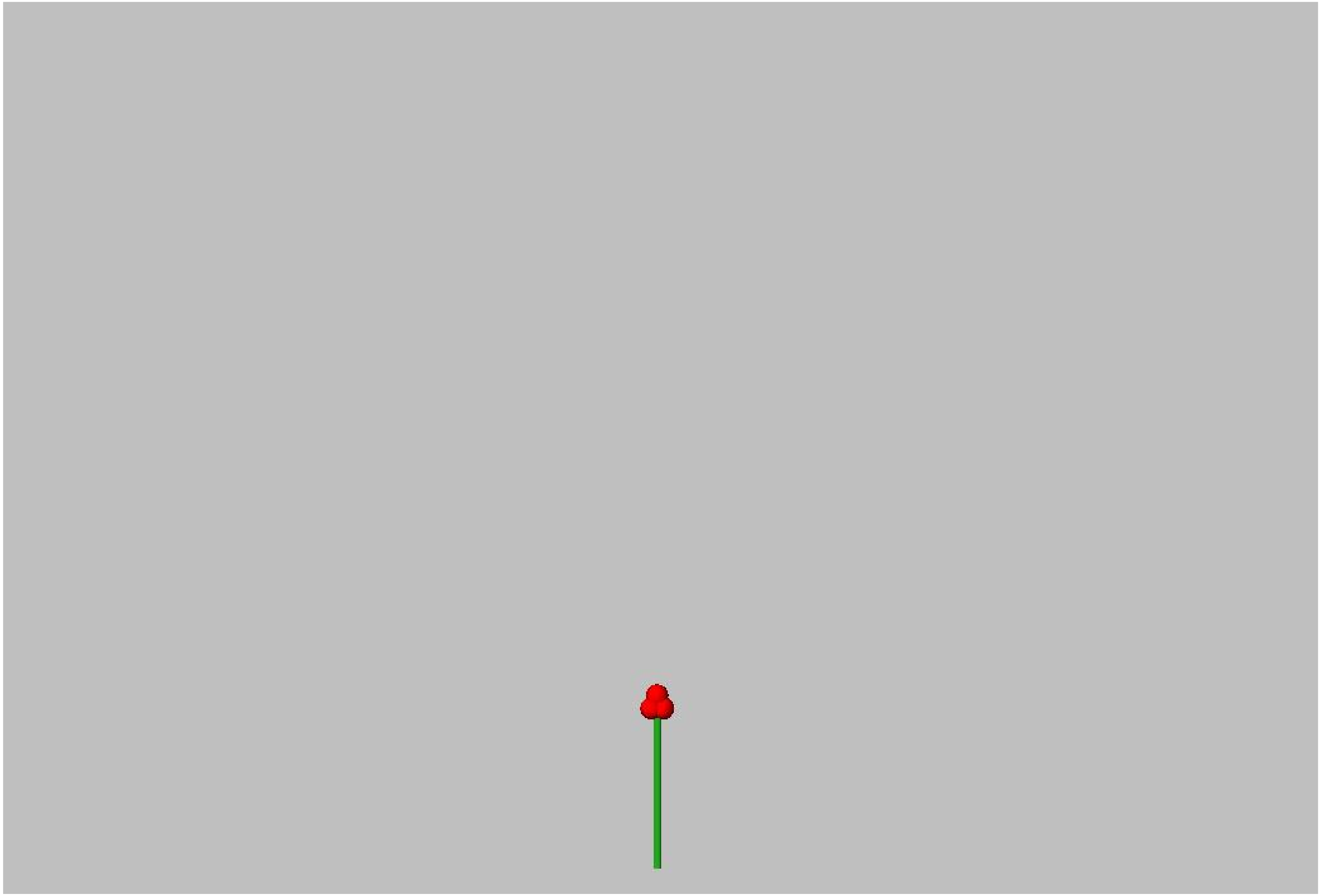
a bud (B) is replaced by a shoot (F0),
2 lateral buds and an apical bud

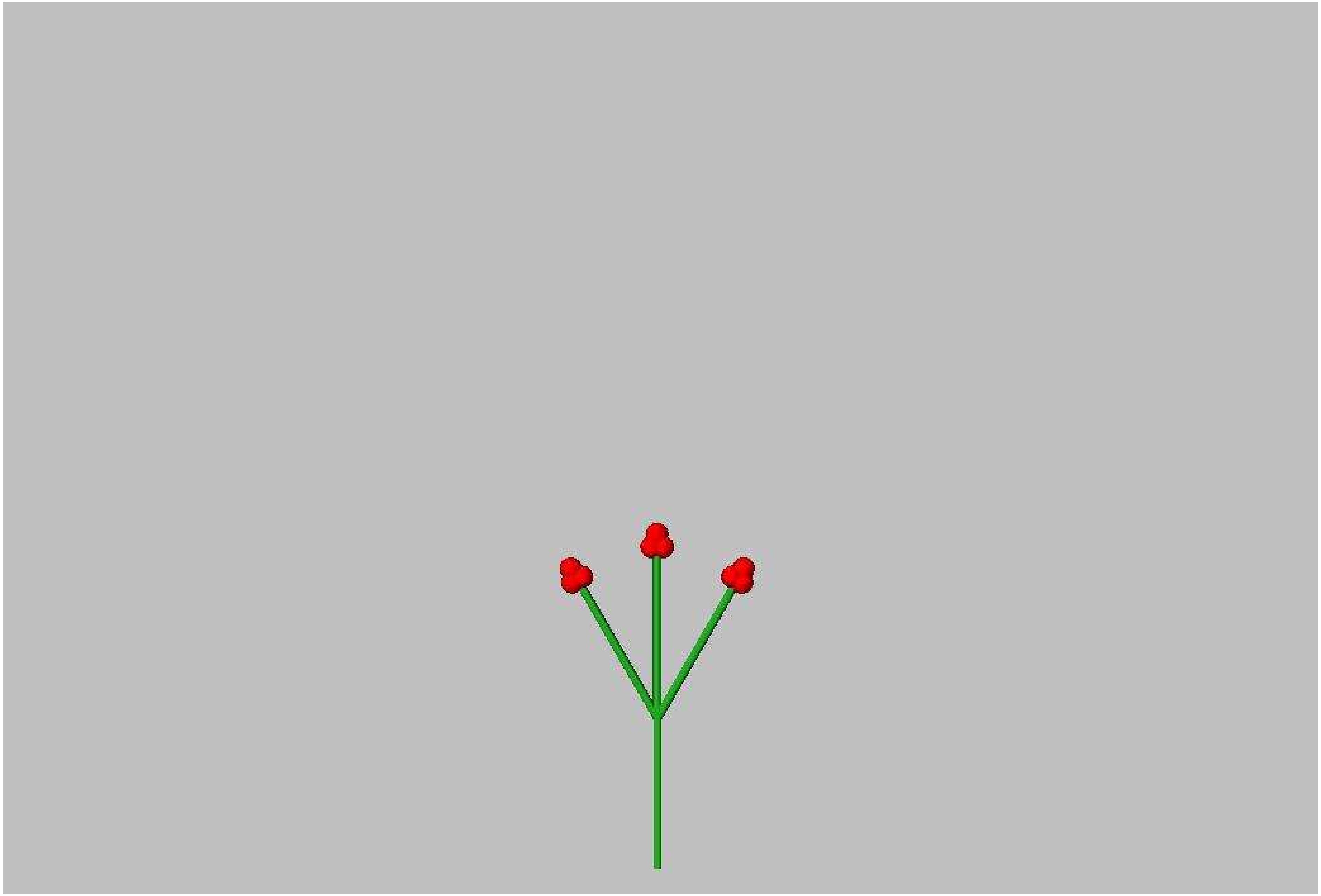


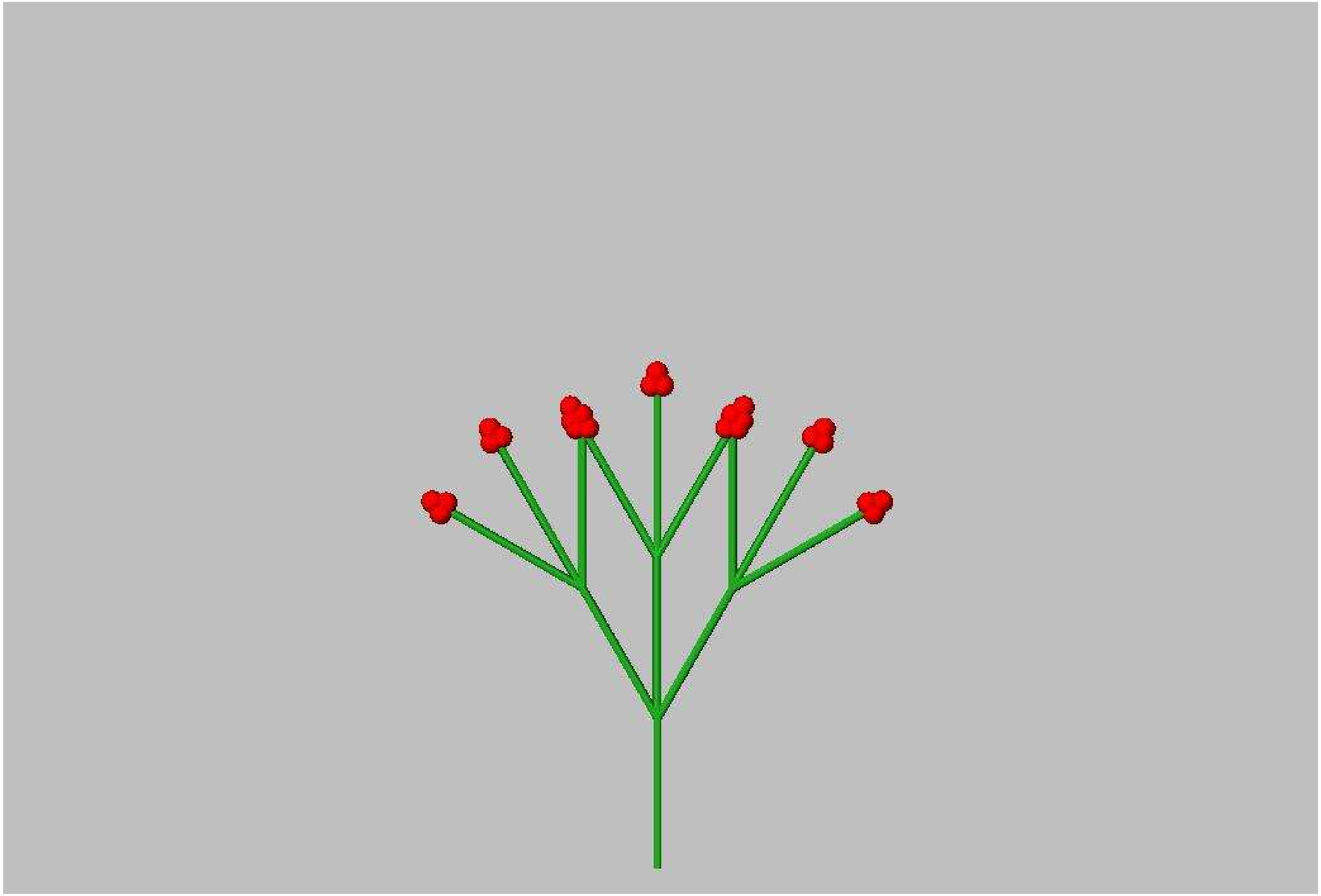
Description by L-system:

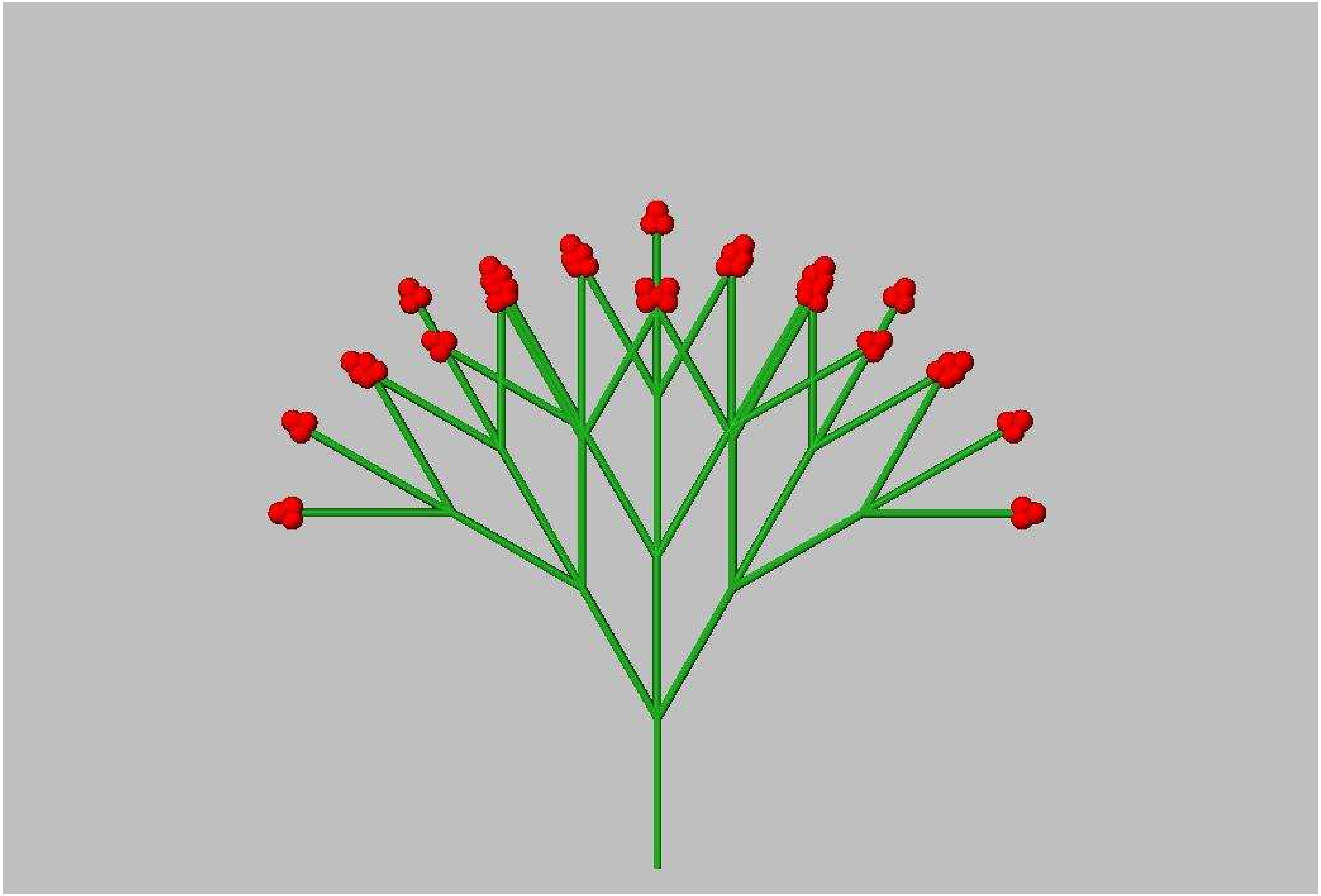
B $\rightarrow$ F0 [RU(-30) B] [RU(+30) B] B

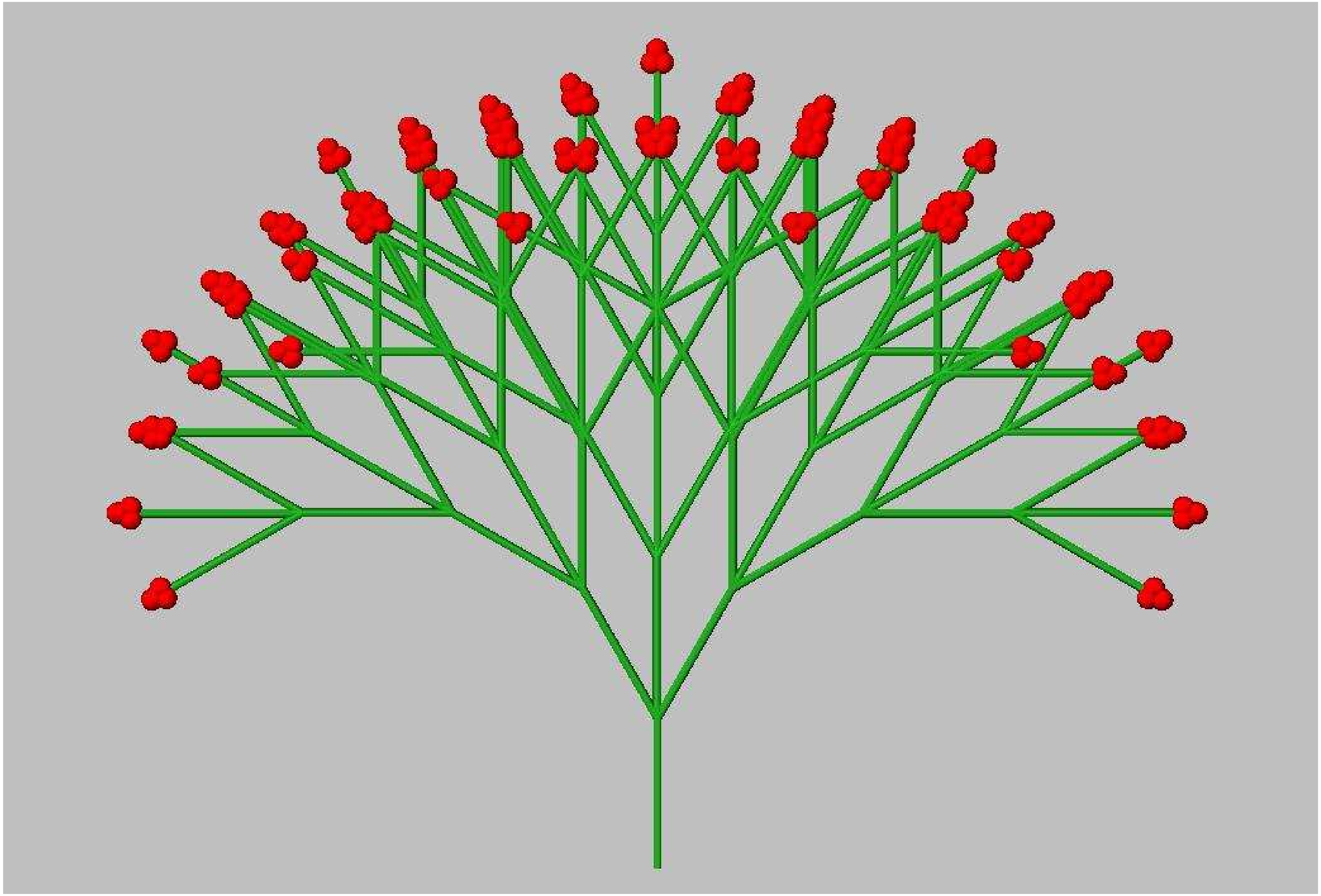








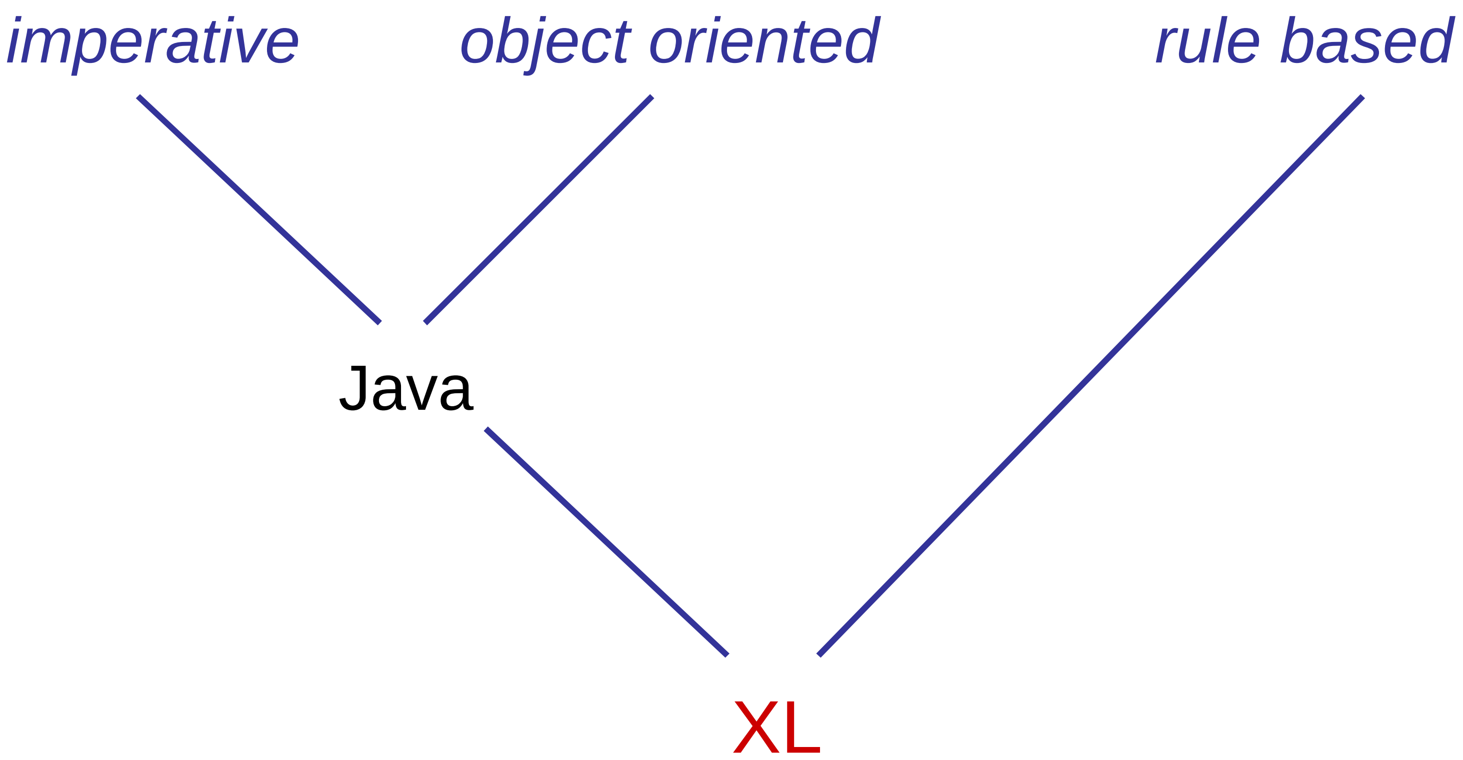




The programming language XL: a synthesis of three paradigms

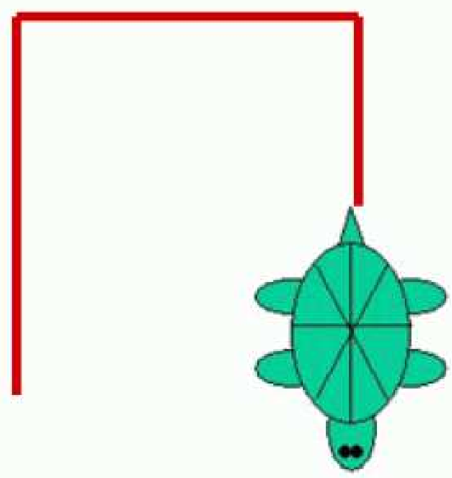
„eXtended L-system language“

programming language which makes parallel graph-grammars (extended L-systems) accessible in a simple way



let‘s come back to turtle geometry...

```
F0 RU(90) F0 RU(90) LMul(0.5) F0
```



Turtle geometry

„turtle": virtual device for drawing or construction in 2-D or 3-D space

- able to store information (graphical and non-graphical)

- equipped with a memory containing state information (important for branch construction)

- current turtle state contains e.g. current line thickness, step length, colour, further properties of the object which is constructed next

Turtle commands in XL (selection):

`F0` "Forward", with construction of an element (line segment, shoot, internode...), uses as length the current step size
(the zero stands for „no explicit specification of length")

`M0` forward without construction (*Move*)

`L(x)` change current step size (length) to *x*

`LAdd(x)` increment the current step size to *x*

`LMul(x)` multiply the current step size by *x*

`D(x)`, `DAdd(x)`, `DMul(x)` analogously for current thickness

`P(c)` change current colour to *c* (= 0 .. 15)

`F(x)`, `F(x,d)`, `F(x,d,c)` forward and construct cylinder with *x* = length, *d* = thickness, *c* = colour

`RU(a)` rotate right by *a* degrees

Repetition of substrings possible with "**`for`**"

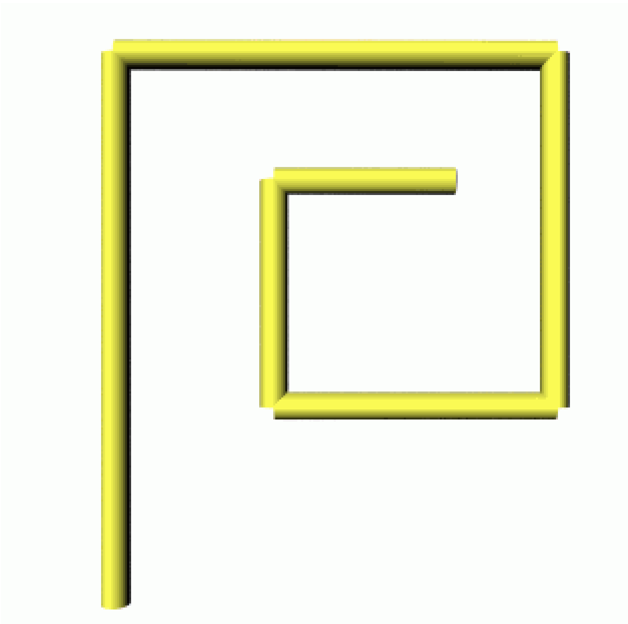
e.g., **`for ((1:3))  ( A B C )`**

yields **`A B C A B C A B C`**

what is the result of the interpretation of

```
L(2) for ((1:6))
         ( F0 RU(90) LMul(0.8) )
```
?

```
L(2) for ((1:6))
        ( F0 RU(90) LMul(0.8) )
```

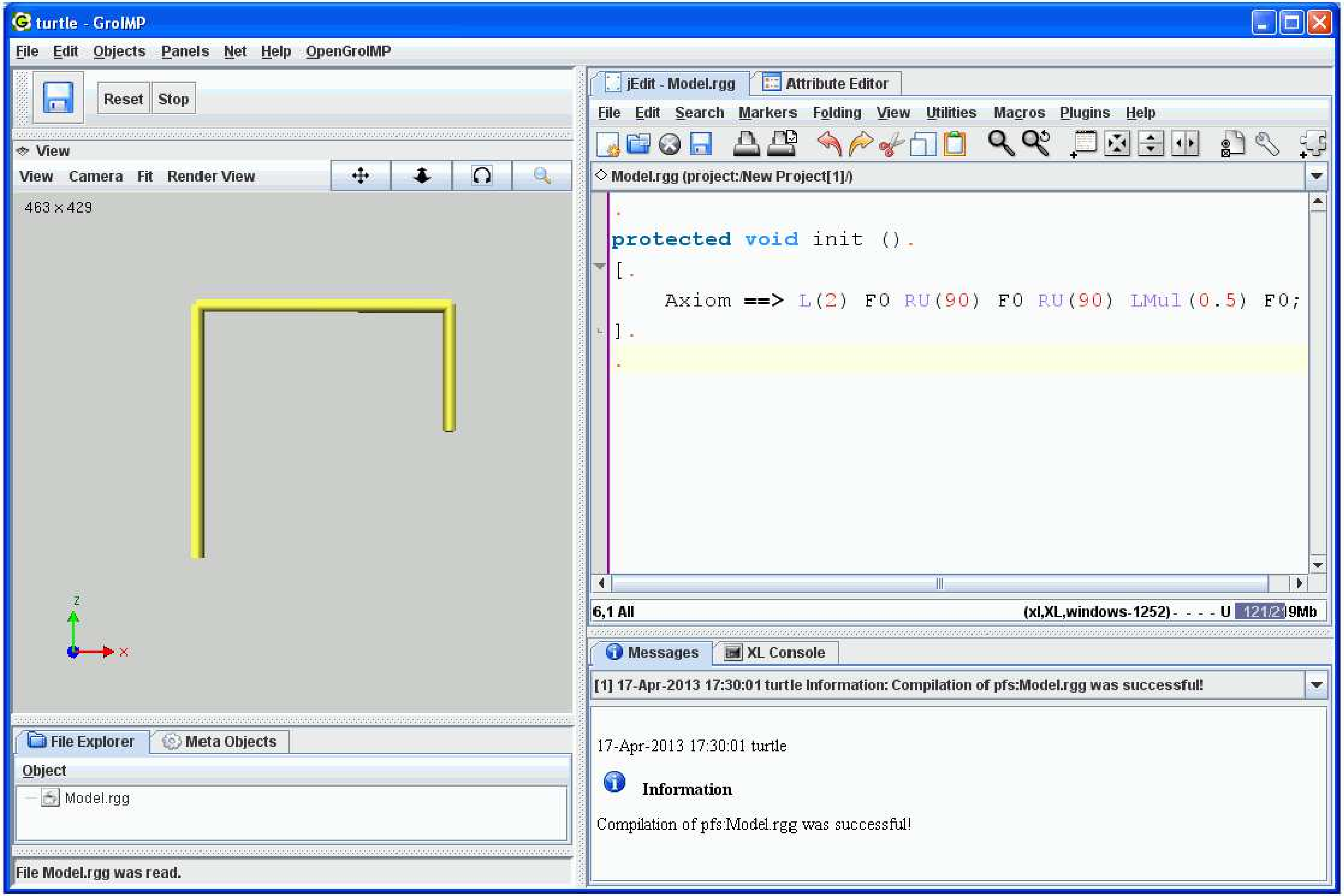


How to execute a turtle command sequence with GroIMP

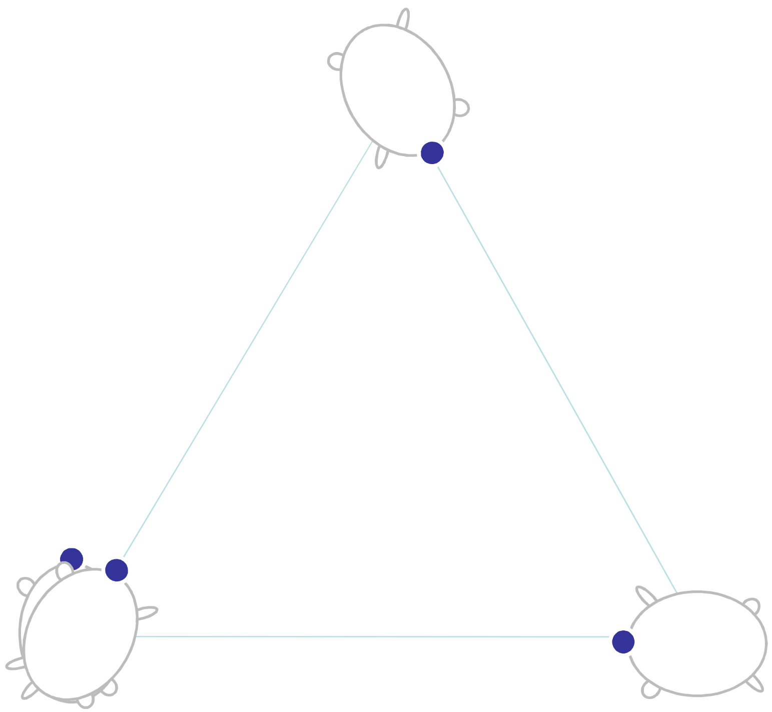
write into a GroIMP project file (or into a file with filename extension **`.rgg`**):

```
protected void init()
    [
    Axiom ==> turtle command sequence ;
    ]
```

Turtle geometry with GroIMP



Example: Drawing a triangle

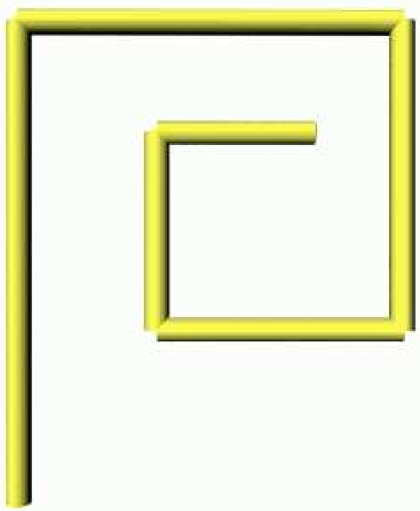


```
protected void init()
   [ Axiom ==> RU(30) F(10) RU(120) F(10) RU(120) F(10) ]
```

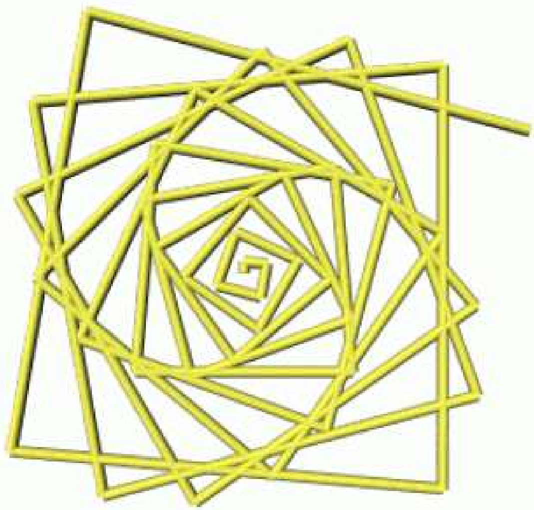
see file `sm09_e01.rgg`

Variations of the spiral pattern

```
Axiom ==>
    L(2)
    for ((1:6)) (
        F0 RU(90) LMul(0.8)
    )
;
```

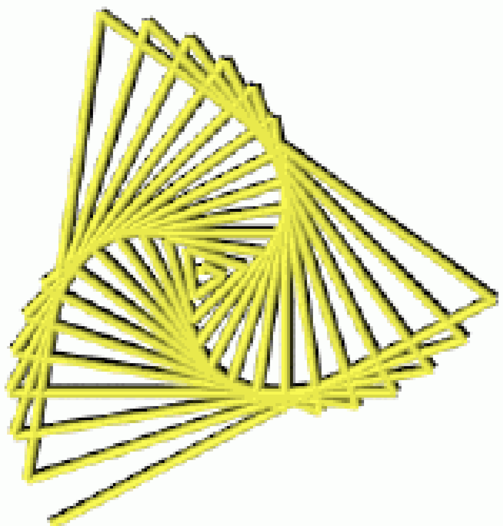


```
Axiom ==>
    L(0.1)
    for ((1:40)) (
        F0 RU(95) LAdd(0.1)
    )
;
```



Variations of the spiral pattern

```
Axiom ==>
     L(0.1)
     for ((1:40)) (
          F0 RU(117) LAdd(0.1)
     )
;
```

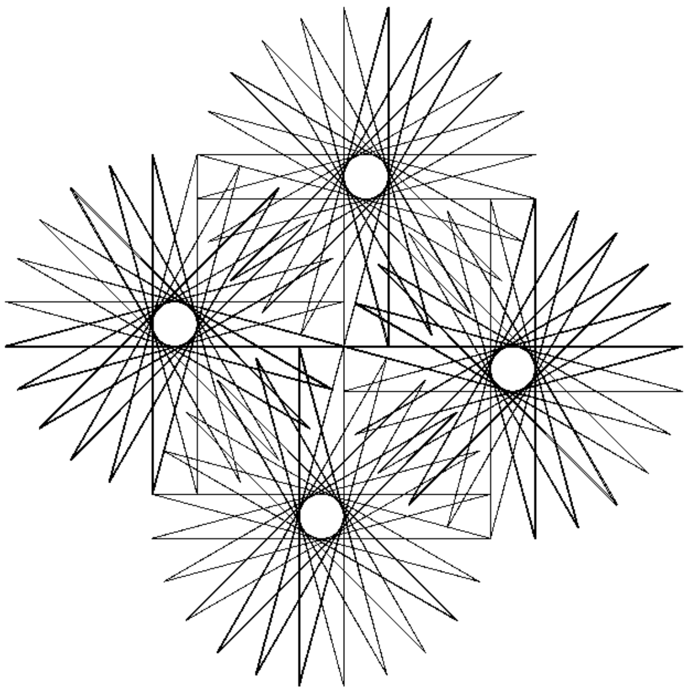


...and more...:

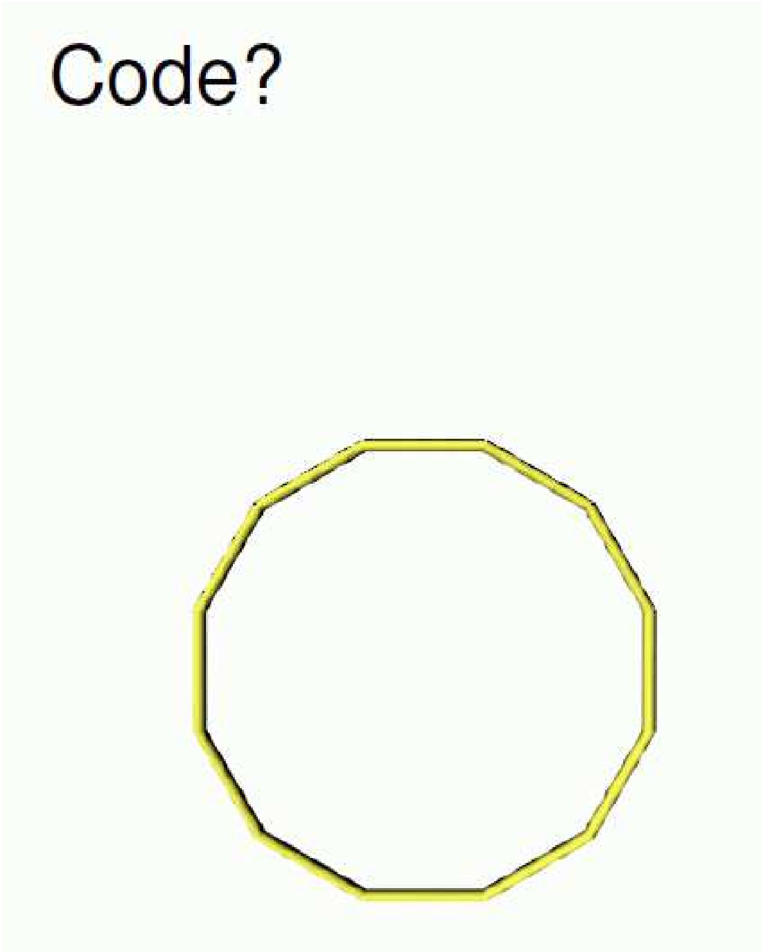
```
for ((1:20)) ( for ((1:36))
                 ( F0 RU(165) F0 RU(165) ) RU(270) )
```

...and more...:

```
for ((1:20)) ( for ((1:36))
                    ( F0 RU(165) F0 RU(165) ) RU(270) )
```

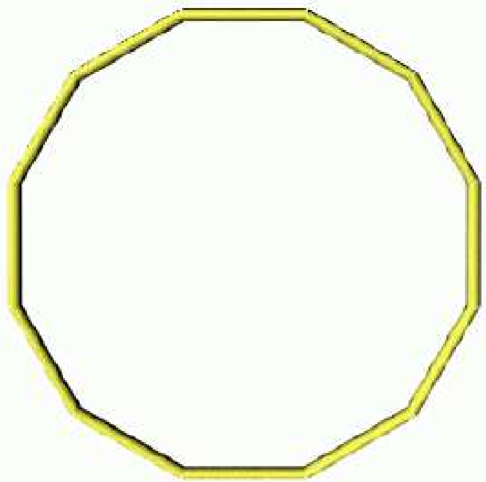


How to draw a “circle“



How to draw a “circle“

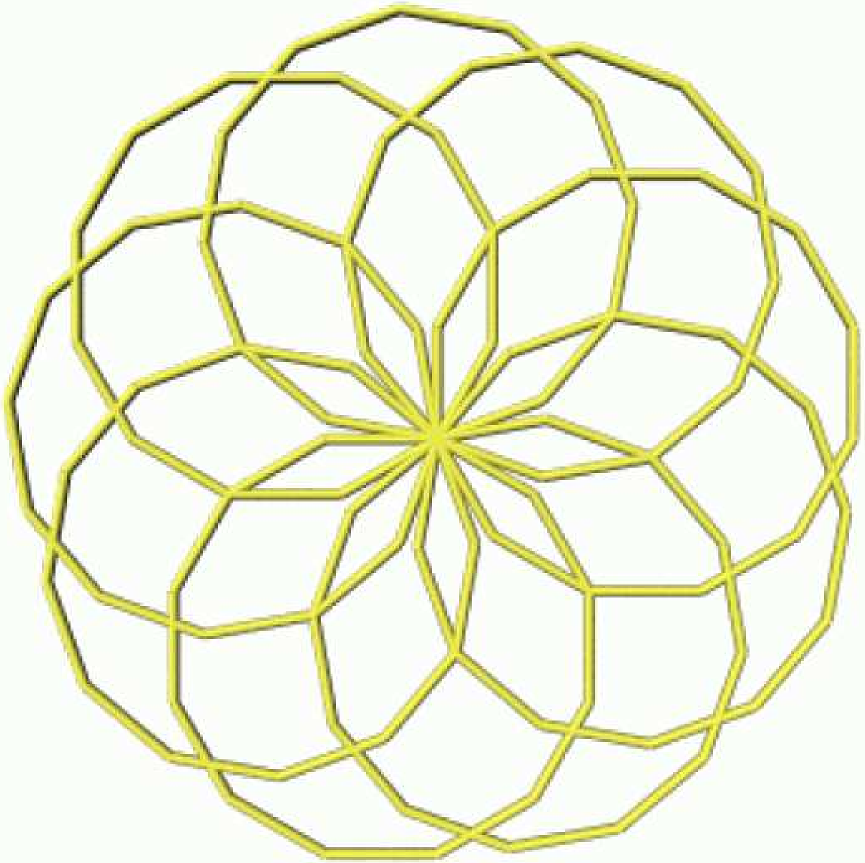
```
Axiom ==>
     L(1)
     for ((1:12)) (
          F0 RU(360/12)
     )
;
```



```
Axiom ==>
     L(1)
     for ((1:9)) (
          for ((1:12)) (
               F0 RU(360/12)
          )
          RU(40)
     )
;
```

Result?

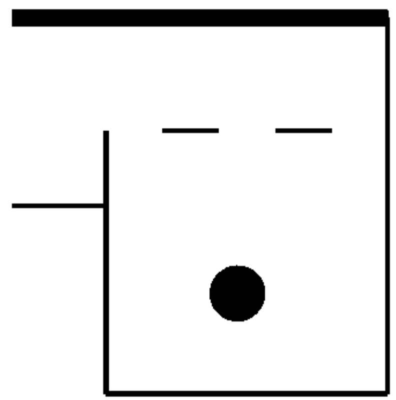
```
Axiom ==>
    L(1)
    for ((1:9)) (
        for ((1:12)) (
            F0 RU(360/12)
        )
        RU(40)
    )
;
```



One more example:

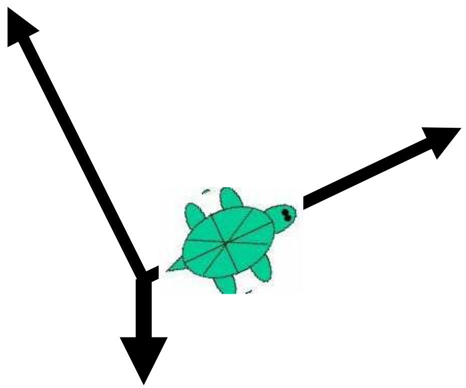
```
L(100) D(3) RU(-90) F(50) RU(90) M0 RU(90) D(10) F0 F0
      D(3) RU(90) F0 F0 RU(90) F(150) RU(90) F(140) RU(90)
      M(30) F(30) M(30) F(30) RU(120) M0 Sphere(15)
```

generates



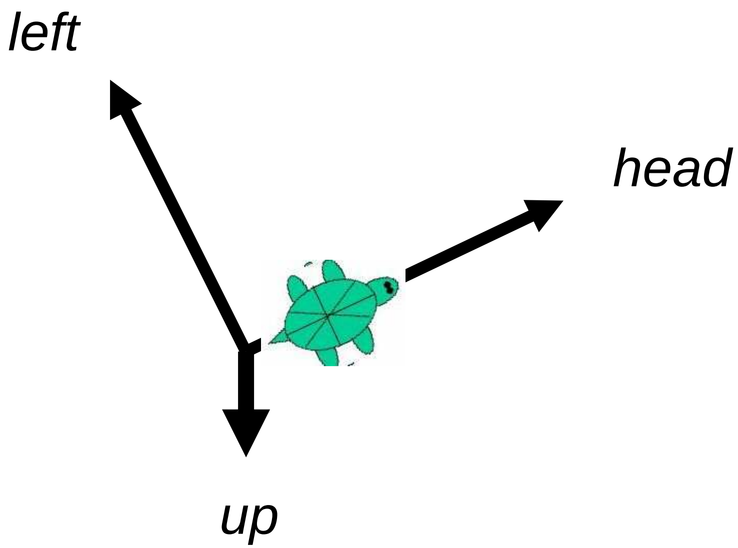
Extension to 3-D graphics:

turtle rotations by 3 axes in space



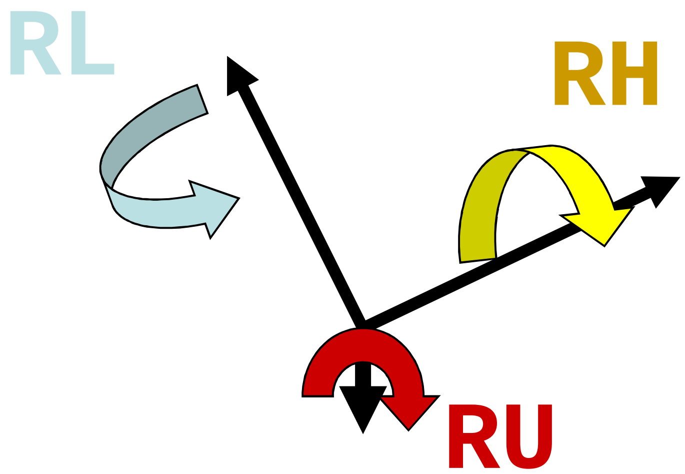
Extension to 3-D graphics:

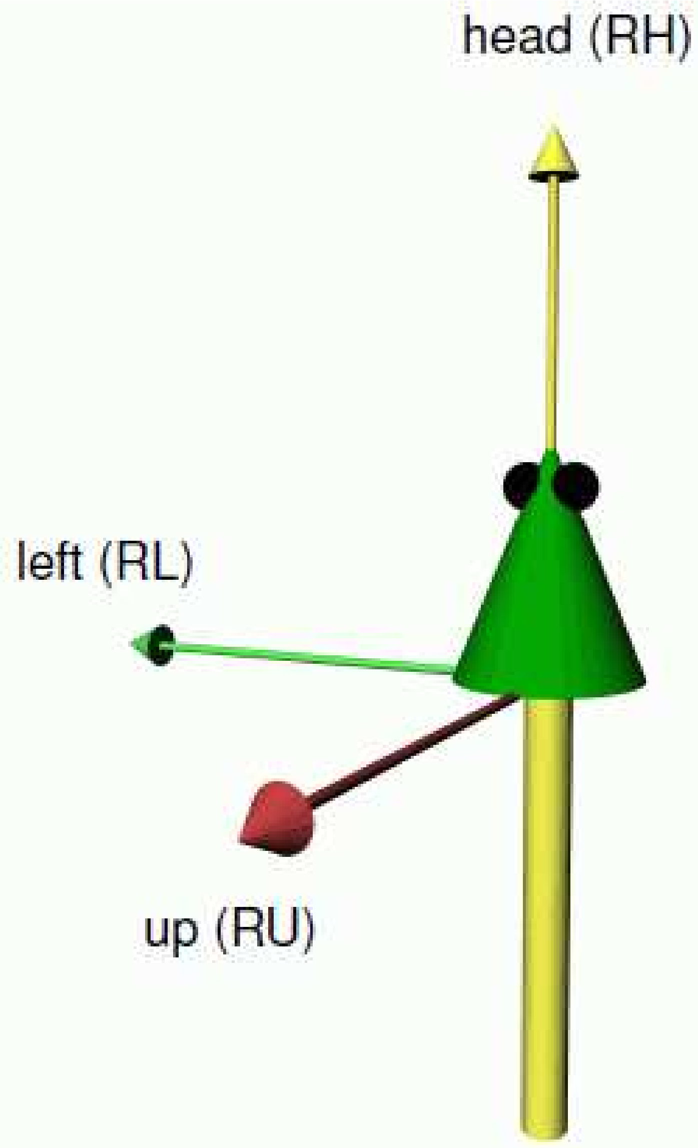
turtle rotations by 3 axes in space



Extension to 3-D graphics:

turtle rotations by 3 axes in space




head (RH)
left (RL)
up (RU)

3-D commands:

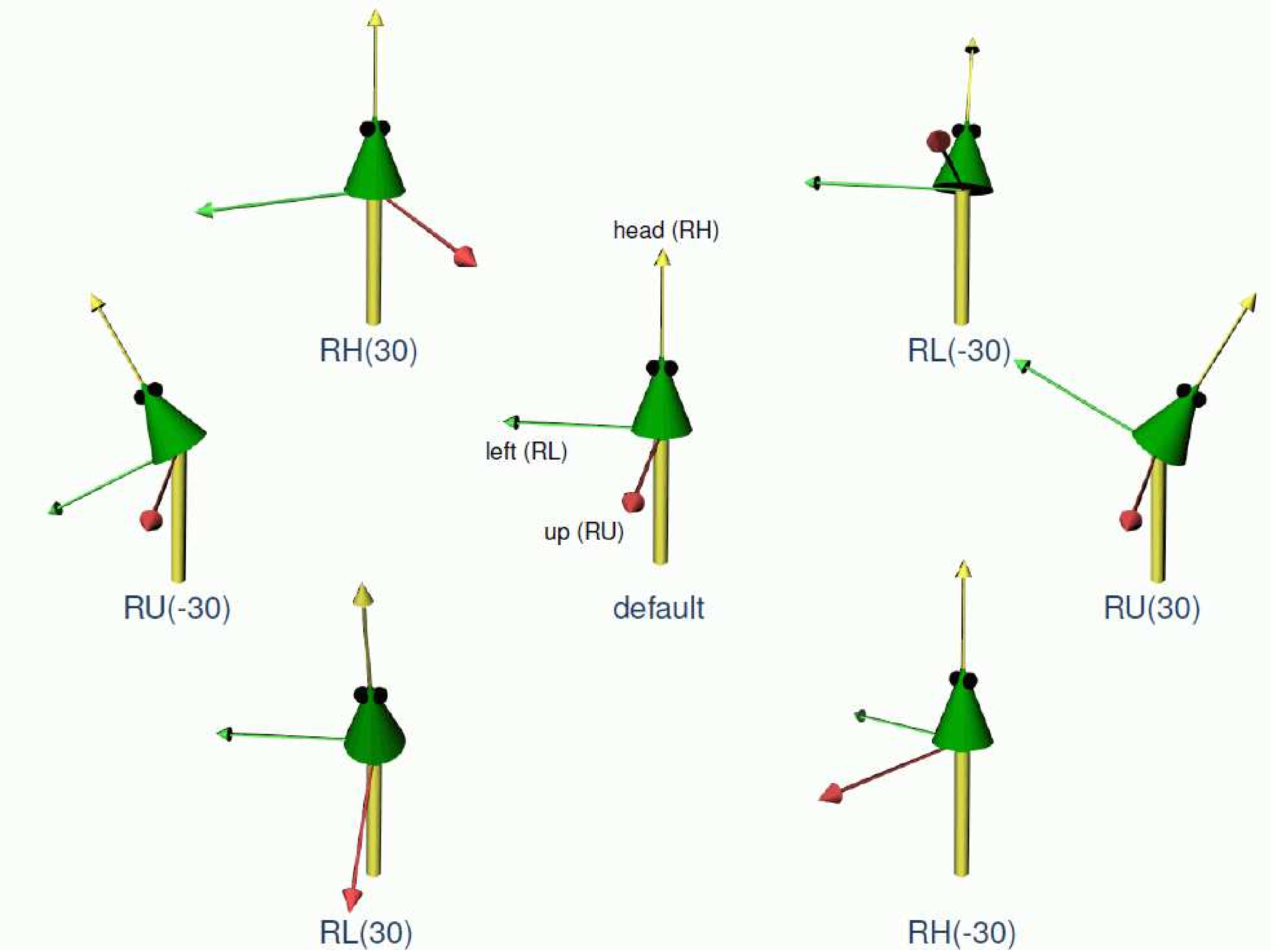
`RU(45)` rotation of the *turtle* around the "up" axis by 45°

`RL(...)`, `RH(...)` analogously by "left" and "head" axis

up-, *left*- and *head* axis form an orthogonal spatial coordinate system which is carried by the *turtle*

`RV(x)` rotation "to the ground" with strength given by **x**

`RG` rotation absolutely to the ground (direction (0, 0, -1))

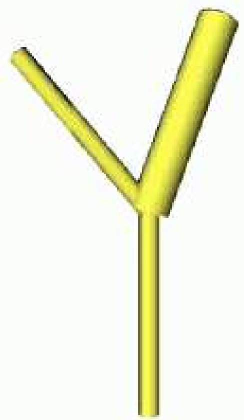

head (RH)
left (RL)
up (RU)
default
RH(30)
RL(-30)
RU(-30)
RU(30)
RL(30)
RH(-30)

Branches:
realization with memory commands

[put current state on stack
(last-in-first-out storage)

] take current state from stack
and let it become the current state
(thus: end of branch!)

```
Axiom ==>
    L(1) F0 [RU(-40) F0] RU(20) DMul(2) F0
;
```



Remember:

L-systems (Lindenmayer systems)

rule systems for the replacement of character strings

in each derivation step *parallel* replacement of all characters for which there is one applicable rule

interpreted L-systems:

turtle command language is subset of the alphabet of the L-system

Aristid Lindenmayer (1925-1989)

Example:

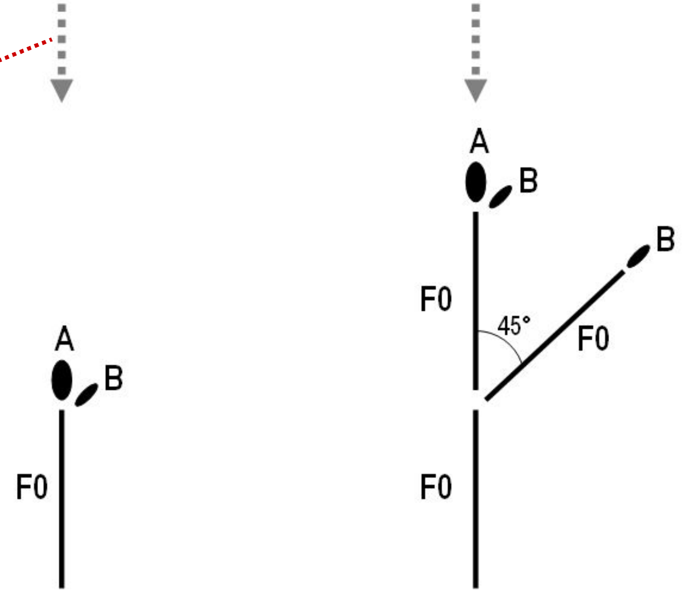
rules

```
A ==> F0 [ RU(45) B ] A ;
B ==> F0 B ;
```

start word **A**

A → F0 [RU(45) B] A → F0 [RU(45) F0 B] F0 [RU(45) B] A → ...

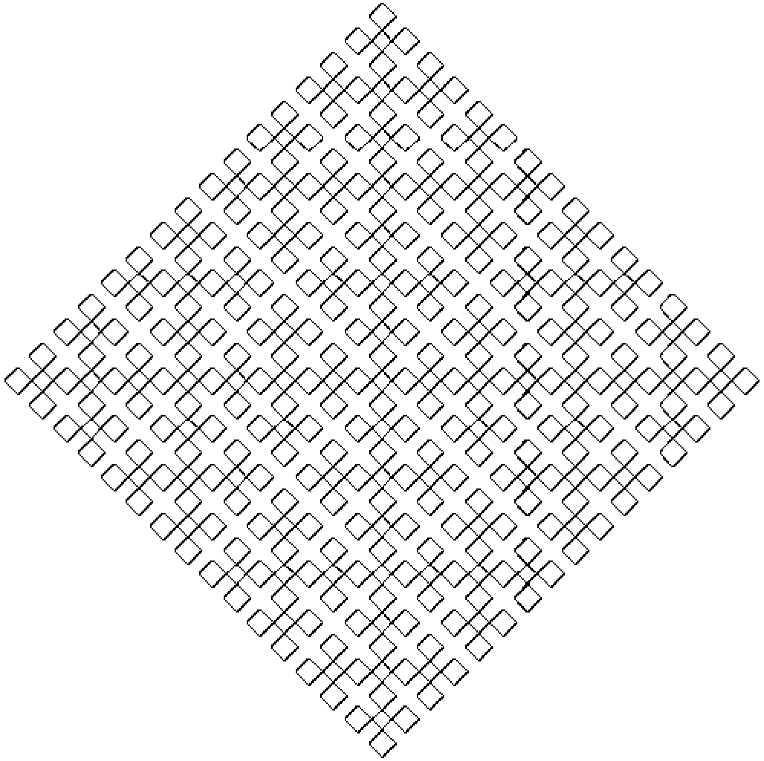
interpretation
by
turtle geometry



(**A** and **B** are normally not interpreted geometrically.)

example space filling curve:

```
Axiom ==> L(10) RU(-45) X RU(-45) F(1) RU(-45) X;
X ==> X F0 X RU(-45) F(1) RU(-45) X F0 X
```

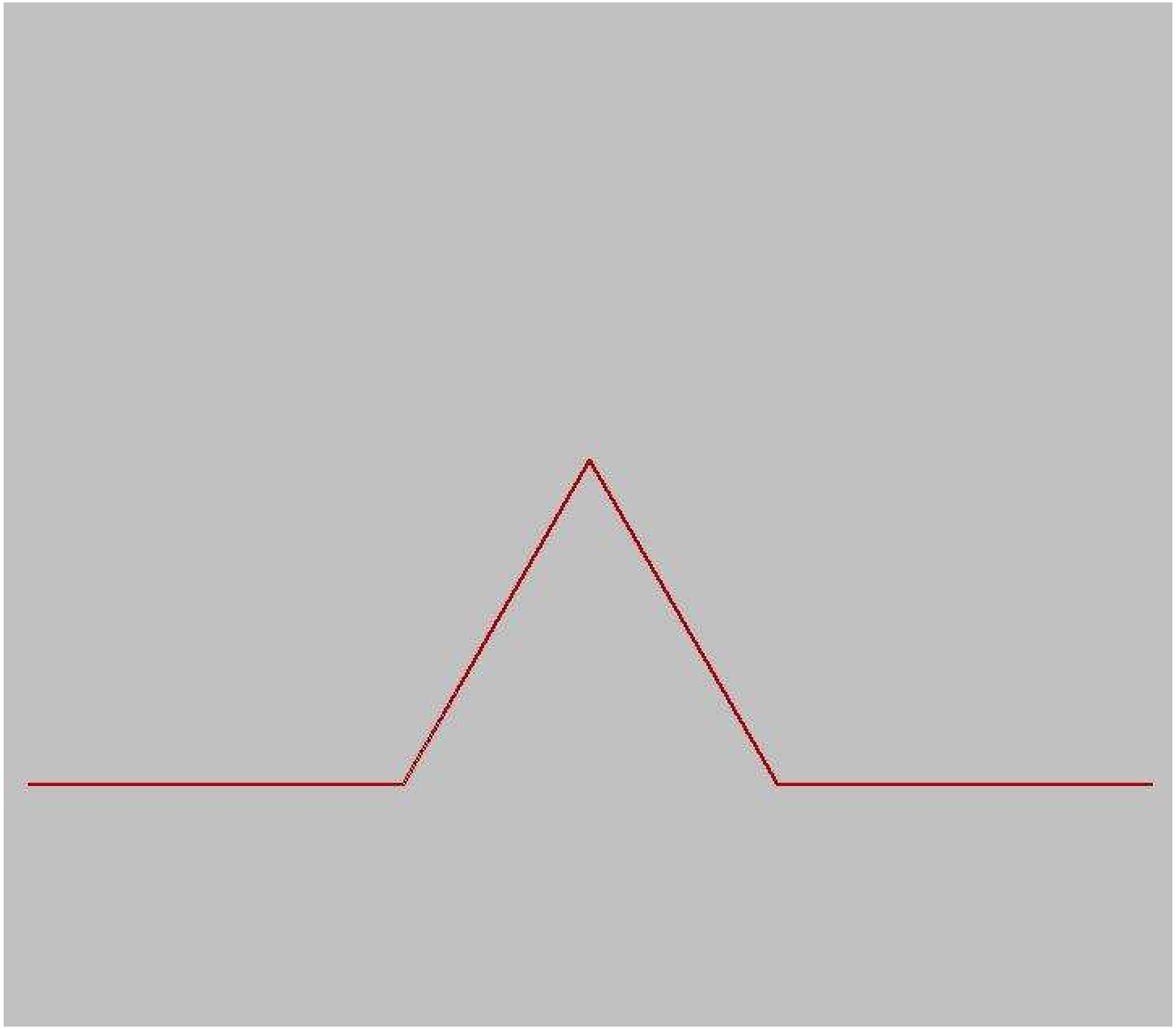


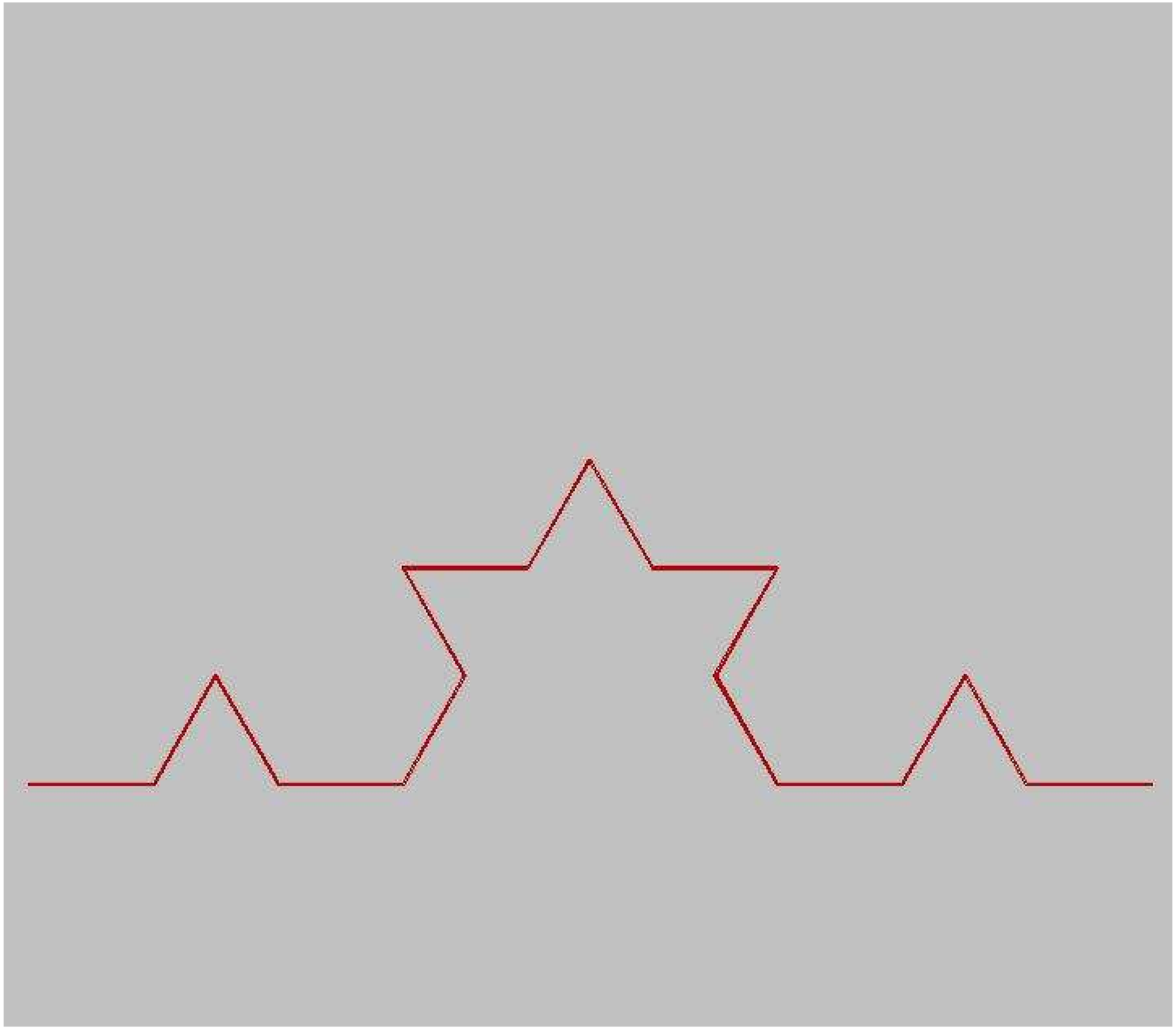
traditional Indian kolam
„Anklets of Krishna“

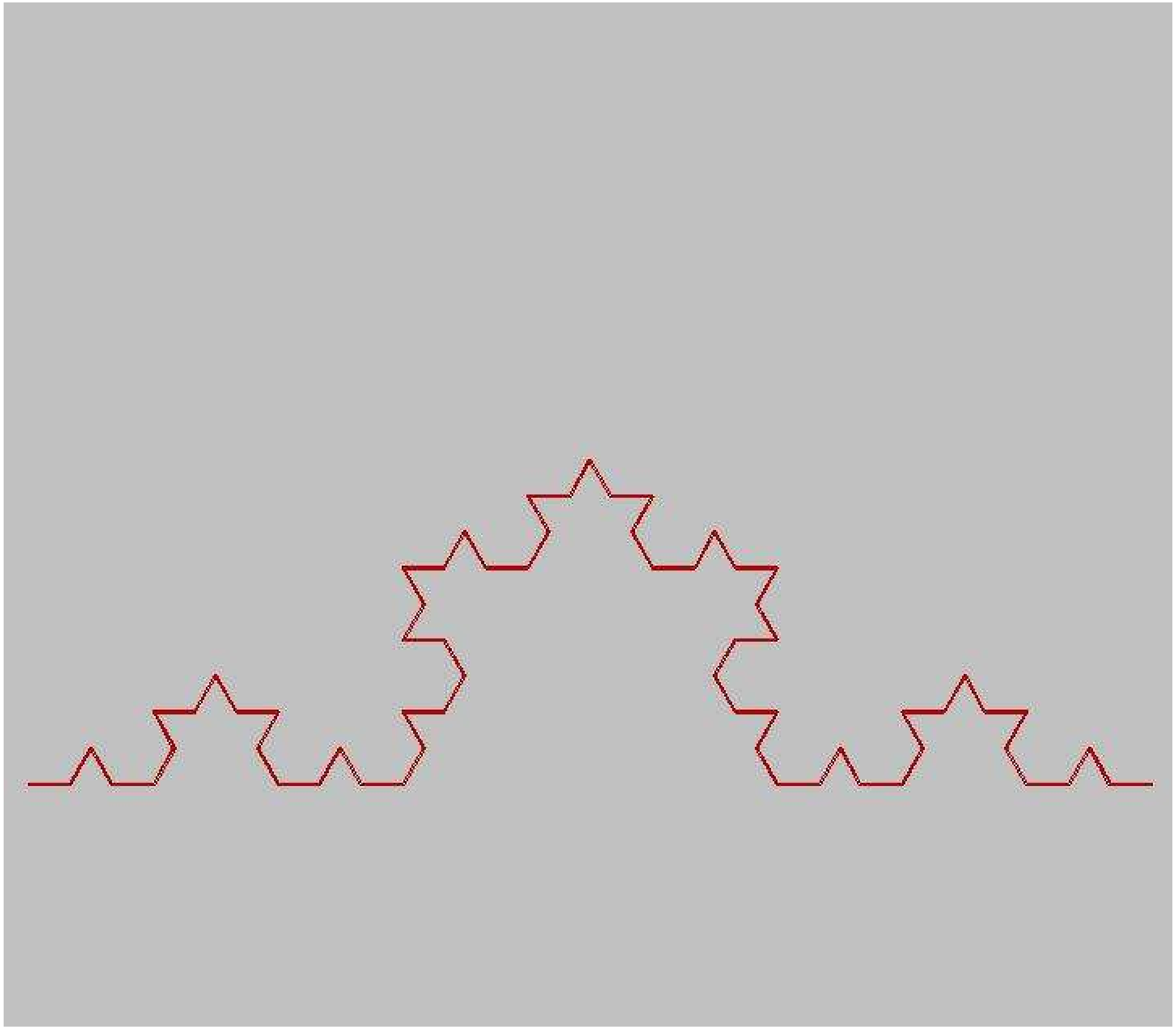
example for a fractal:

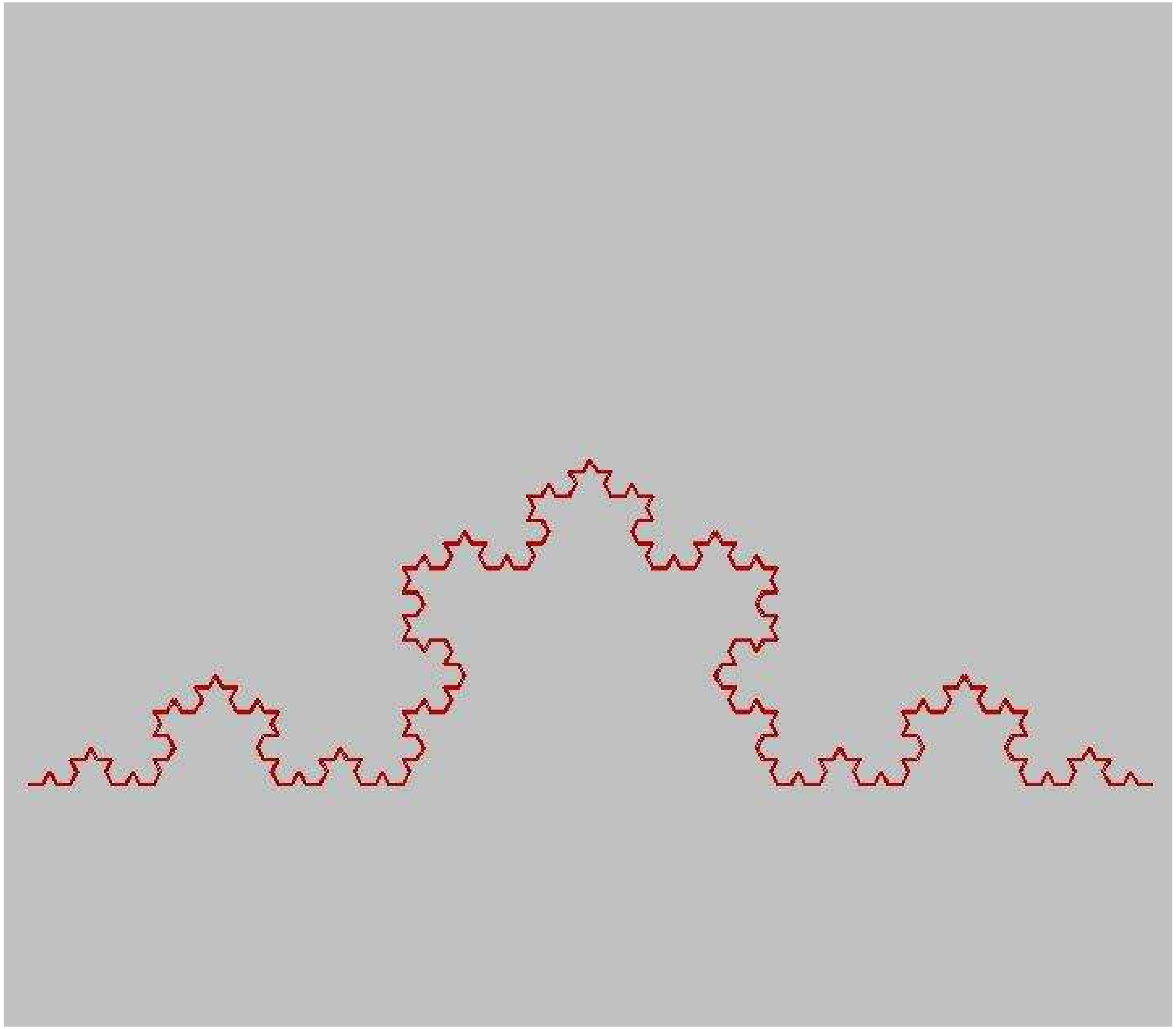
Koch curve

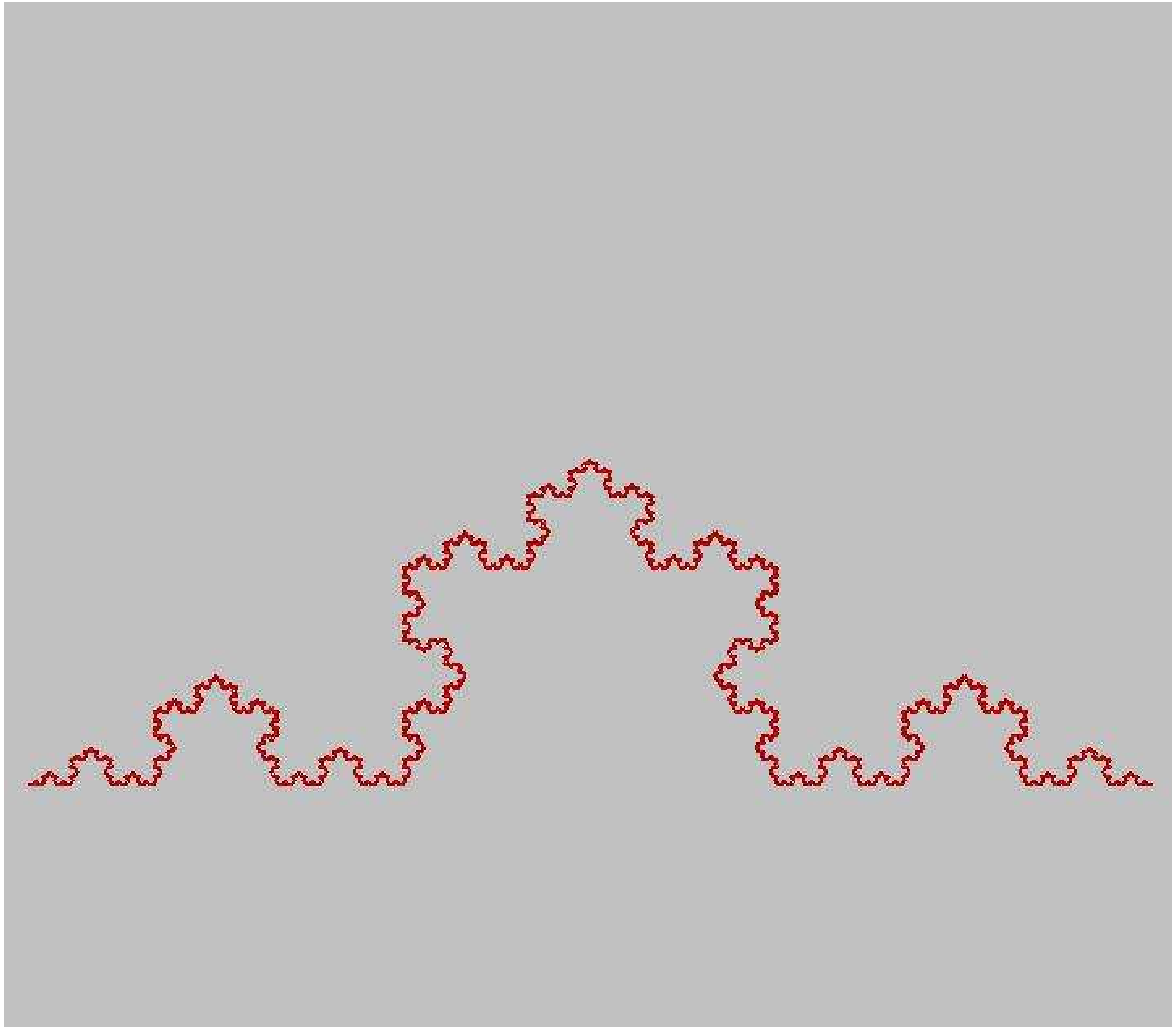
```
Axiom ==> RU(90) F(10);
F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3)
```











cf. sample file **sm09_e02.rgg** :

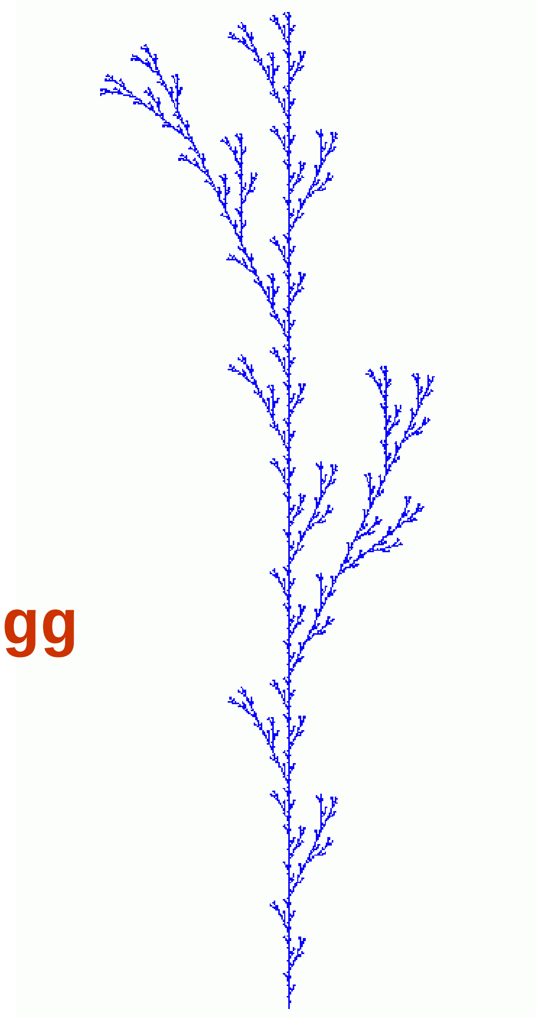
closed Koch curve, developed from triangle

```
protected void init()
[ Axiom ==> RU(50) F(10) RU(120) F(10) RU(120) F(10); ]

// public method for interactive usage in GroIMP
// (via button):
public void application()
// rules must be set in [] and finished with ;
[
       // each F() is replaced by 4 smaller F()
       // the length of the F on the left-hand side is taken over
       // by x to the right-hand side

       F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
]
```

example with branching:

F0 ==> F0 [RU(25.7) F0] F0 [RU(-25.7) F0] F0;

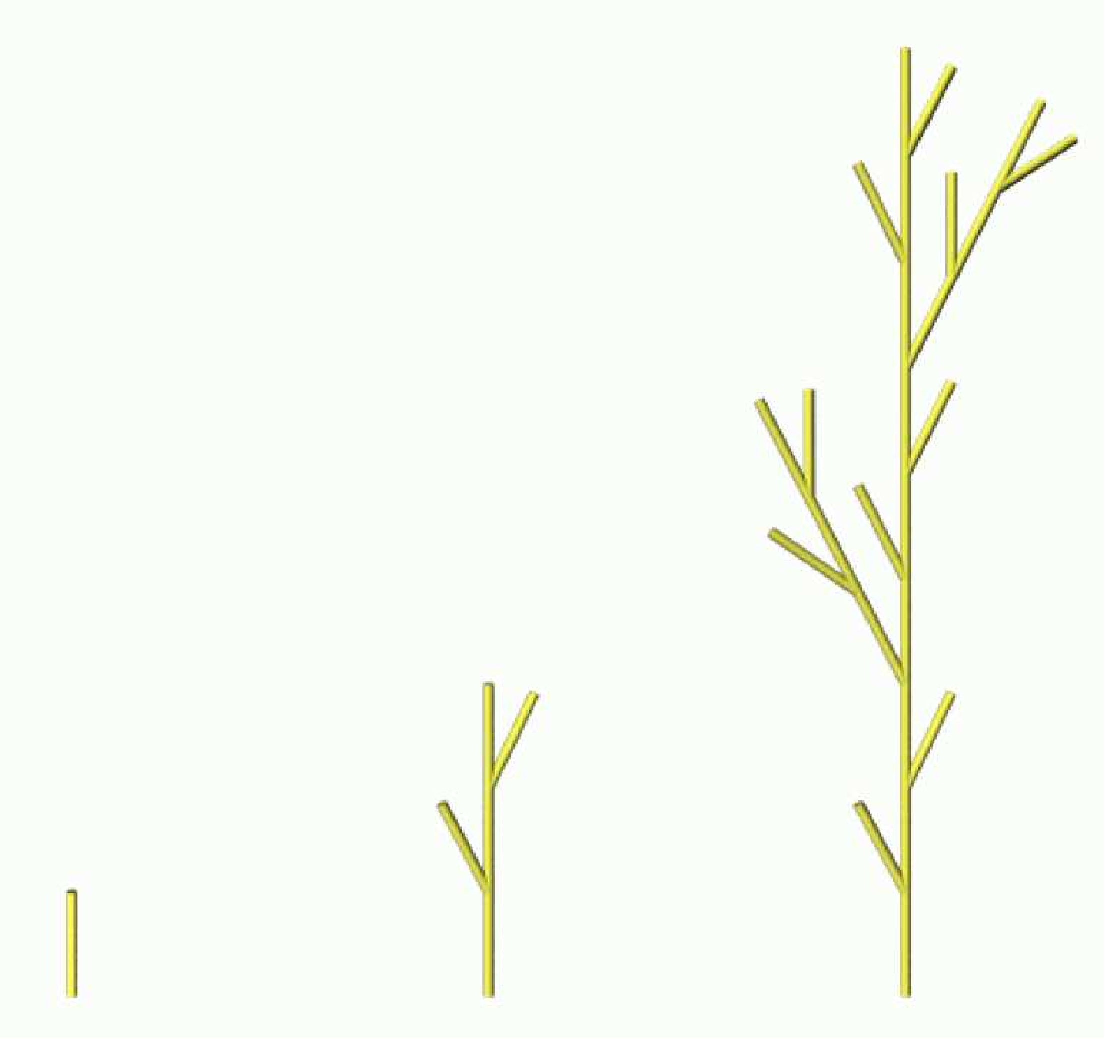
Result after 7 steps:



(start word **L(10) F0**)

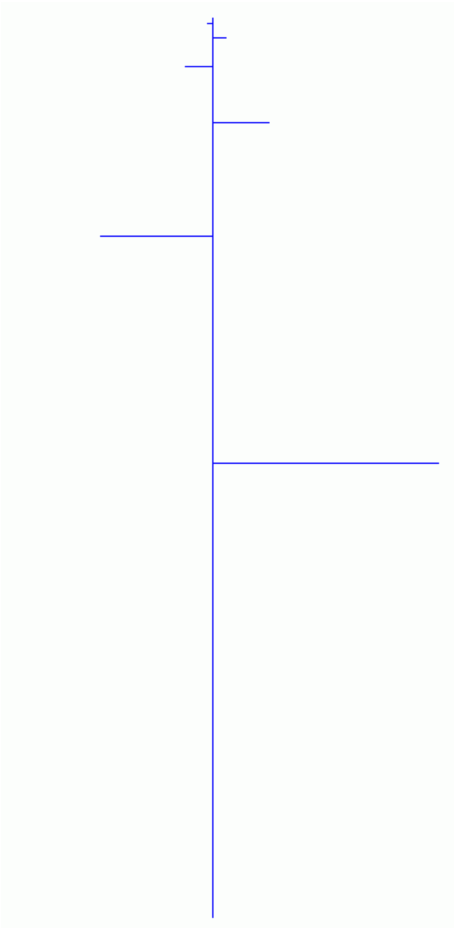
sample file **sm09_e02b.rgg**

```
F0 ==> F0 [ RU(25.7) F0 ] F0 [ RU(-25.7) F0 ] F0;
```



branching, alternating branch position and shortening:

```
Axiom ==> L(10) F0 A;

A ==> LMul(0.5) [ RU(90) F0 ] F0 RH(180) A;
```



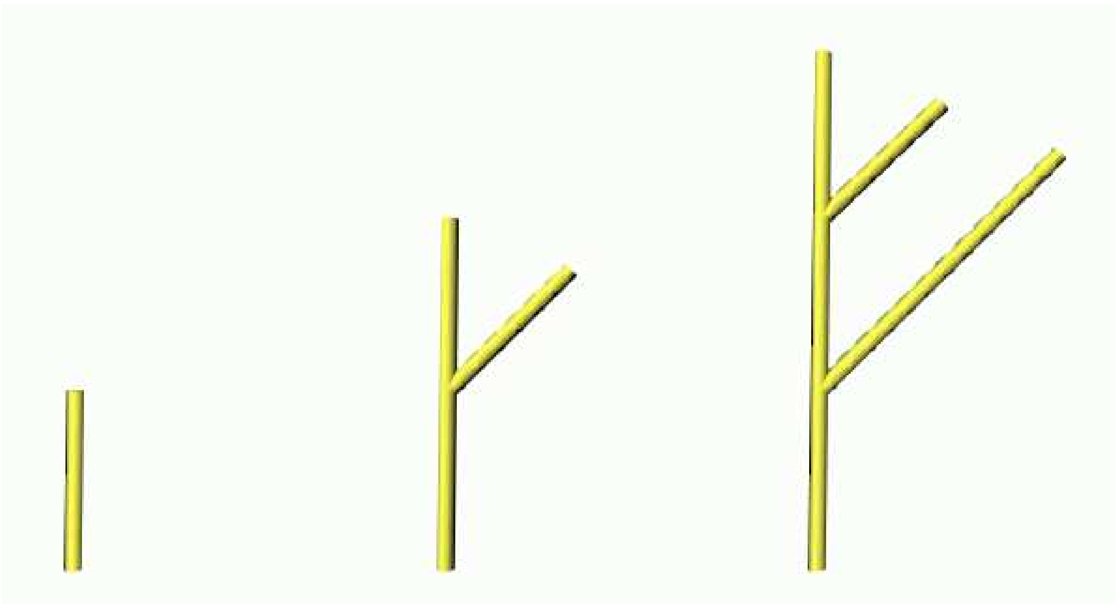
in XL, **A** must be declared as module before:

```
module A;
```

sample file **sm09_e02c.rgg**

Declaration of new modules (which are not visible):

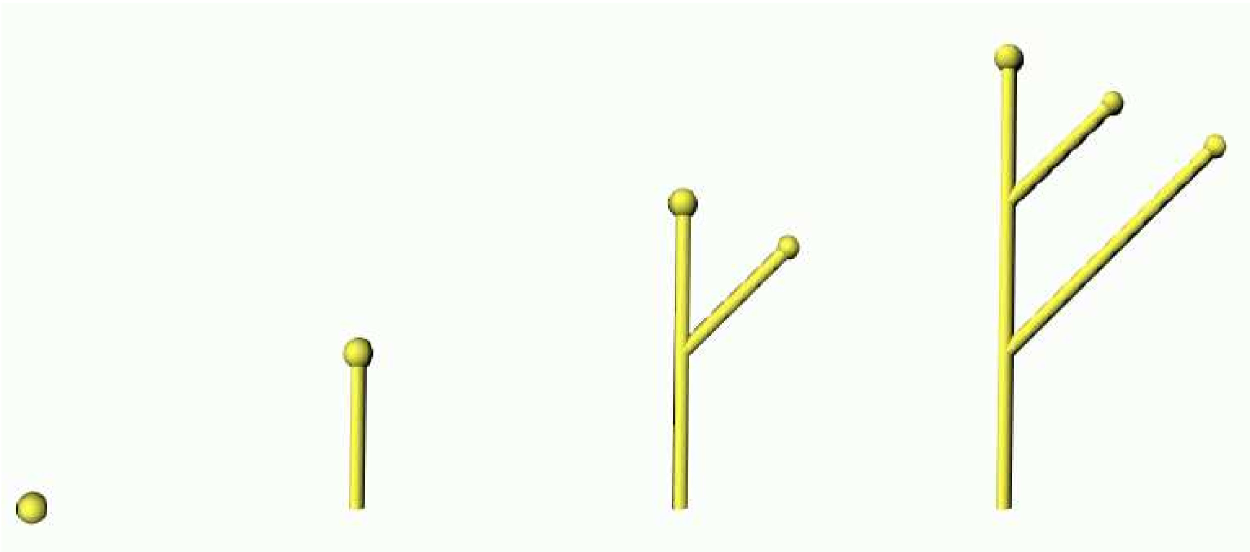
```
module A;
module B;
protected void init()
   [ Axiom ==> L(1) A; ]
public void run()
   [ A ==> F0 [ RU(45) B ] A;
     B ==> F0 B; ]
```



sample file **sm09_e02d.rgg**

How to declare new modules with geometrical interpretation:

```
module A extends Sphere(0.1);
module B extends Sphere(0.08);
protected void init()
   [ Axiom ==> L(1) P(14) A; ]
public void run()
   [ A ==> F0 [ RU(45) B ] A;
     B ==> F0 B; ]
```



sample file **sm09_e02e.rgg**

extension of the concept of symbol:

allow real-valued parameters not only for turtle commands like "**`RU(45)`**" and "**`F(3)`**", but for all characters

→ *parametric L-systems*

- arbitrarily long, finite lists of parameters
- parameters get values when the rule matches

Example:

rule **`A(x, y) ==> F(7*x+10) B(y/2)`**

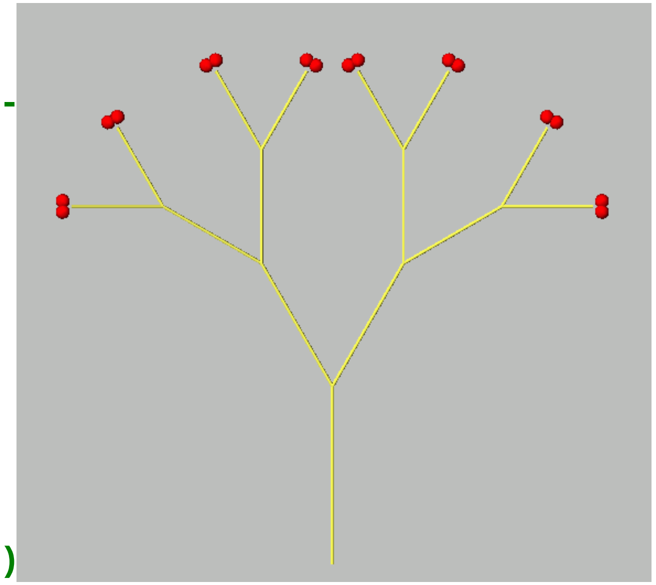
current symbol is e.g.: **`A(2, 6)`**
after rule application: **`F(24) B(3)`**

parameters can be checked in conditions
(logical conditions with Java syntax):

```
A(x, y) (x >= 17 && y != 0) ==> ....
```

sample file **sm09_e03.rgg** :

```
// Example of a simple tree architecture (Schoute architecture)

//----------- Extensions to the standard alphabet ----------
//Shoot() is an extension of the turtle-command F() and stands for an annual shoot
module Shoot(float len) extends F(len);

// Bud is an extension of a sphere object and stands for a terminal bud
// Its strength controls the length of the produced shoot in the next timestep
module Bud(float strength) extends Sphere(0.2)
{{ setShader(RED); setTransform(0, 0, 0.3); }};
//-------------------------------------------------------

protected void init ()
[  // start structure (a bud)
   Axiom ==> Bud(5);
]

public void run ()
[
   // a square bracket [] will indicate a branch
   // (daughter relation)
   // Rotation around upward axis (RU) and head axis (RH)
   // Decrease of strength of the Bud (each step by 20%)

   Bud(x) ==> Shoot(x) [ RU(30) Bud(0.8*x) ] [ RU(-30) Bud(0.8*x) ];
]
```



Test the examples

`sm09_e04.rgg`	two blocks of rules, conditions
`sm09_e05.rgg`	alternating positions of branches, object (bud) with 2 parameters
`sm09_e07.rgg`	colour specifications for single objects, how to give names to objects

How to define colours with XL

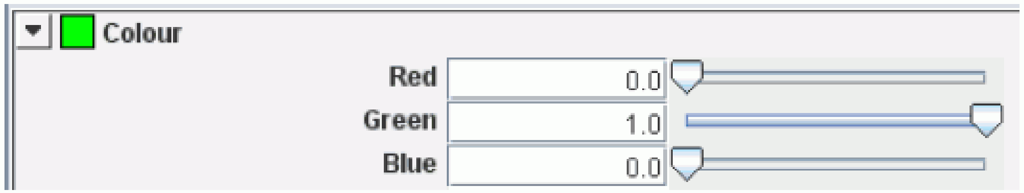
- Turtle command **P**

```
//P(EGA_10) A
P(10) A
//P(GREEN) A
//P(LIGHT_GRAY) A
//P(0x00FF00) A
```

- method **setShader**

```
module A extends Sphere(0.1)
{
    //{ setShader(EGA_10); }
    { setShader(GREEN); }
}
//module A extends Sphere(0.1).(setShader(GREEN));
```

- method **setColor**




```
module A extends Sphere(0.1)
{
    //{ setColor(0x00FF00); }
    { setColor(0.0, 1.0, 0.0); }
}
```

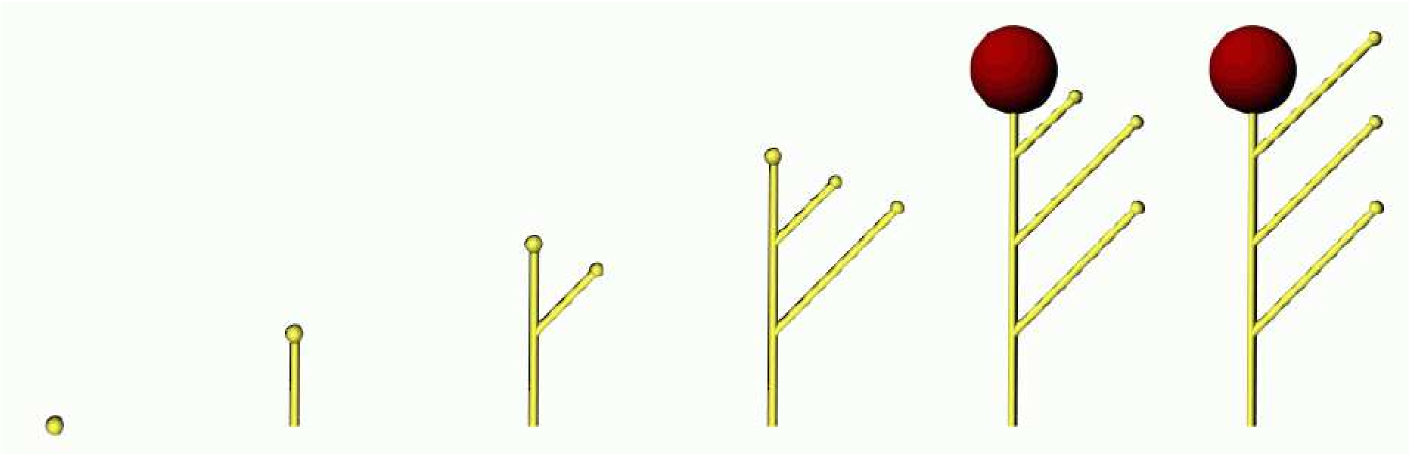
Test the examples

sm09_e08.rgg using your own modules as abbreviations

sm09_e21.rgg different ways to position lateral branches

A further parametric L-system with conditions:

```
module A(int age) extends Sphere(0.1);
module B(int age) extends Sphere(0.08);

protected void init()
[
    Axiom ==> P(14) A(0);
]

public void run()
[
    A(age), (age < 3) ==> L(1) F0 [RU(45) B] A(age+1);
    A(age), (age == 3) ==> F0 P(4) Sphere(0.5);
    B(age), (age < 2) ==> L(1) F0 B(age+1);
]
```



Context-sensitive L-systems

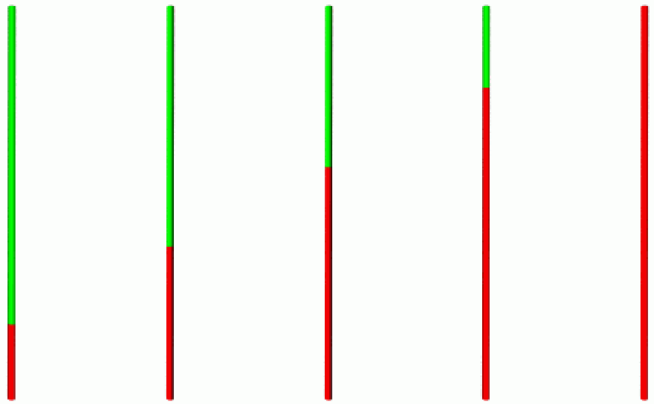
Often, the development of some object is influenced by a neighbour object

Context sensitivity:

- checking for a context (in the representing string code) which must be present in order for the rule to be applicable
- in XL, the context is given in **(*** ... ***)**

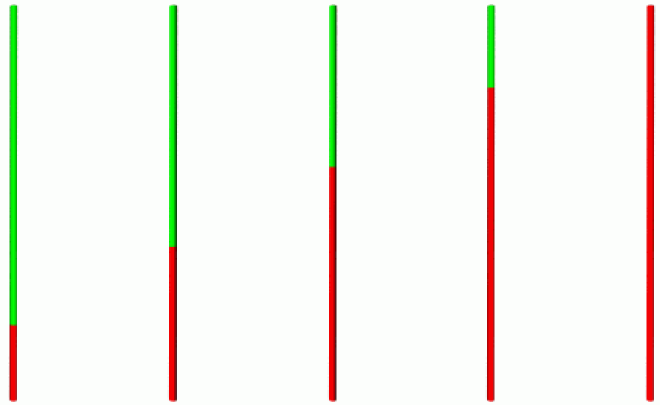
Examples:

```
module GreenF extends F(1, 0.1, 2);
module RedF extends F(1, 0.1, 4);

// left context
protected void init()
[
    Axiom ==> RedF GreenF GreenF GreenF GreenF;
]

public void run()
[
    (* RedF *) GreenF ==> RedF;
]
```

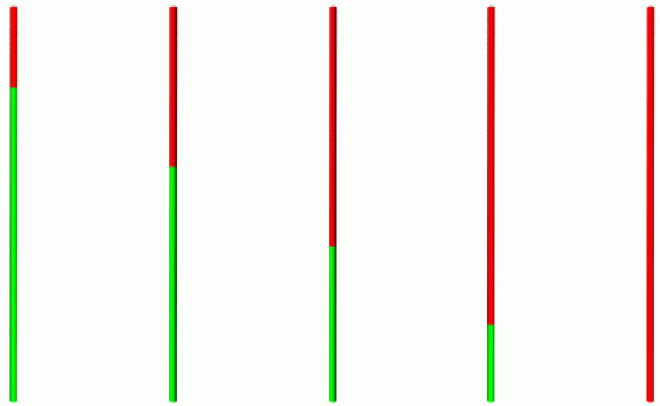


Examples:

```
module GreenF extends F(1, 0.1, 2);
module RedF extends F(1, 0.1, 4);

// left context

protected void init()
[
    Axiom ==> RedF GreenF GreenF GreenF GreenF;
]

public void run()
[
    (* RedF *) GreenF ==> RedF;
]
```



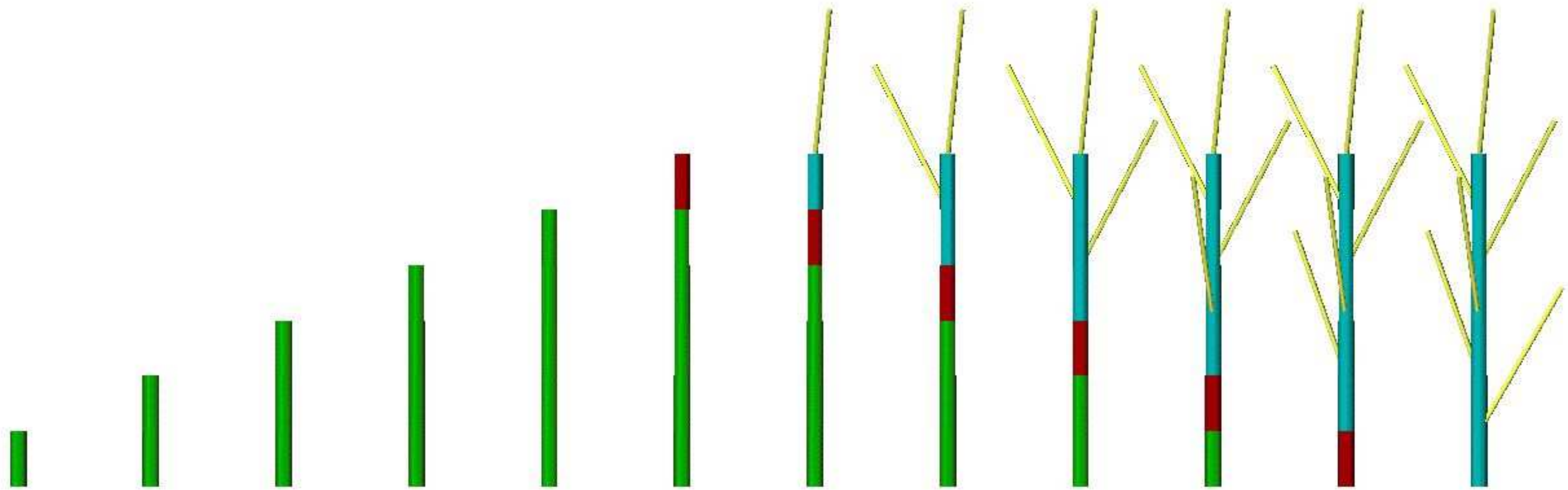
```
// right context

protected void init()
[
    Axiom ==> GreenF GreenF GreenF GreenF RedF;
]

public void run()
[
    GreenF (* RedF *) ==> RedF;
]
```



see also **sm09_e14.rgg**, **sm09_e15.rgg**

Further example for context sensitivity

```
const int green = 2, blue = 3, red = 4;
module A(int age);
module B(super.length, super.color) extends F(length, 3, color);
Axiom ==> A(0);
A(t), (t < 5) ==> B(10, green) A(t+1);
A(t), (t == 5) ==> B(10, red);
B(s, green) (* B(r, red) *) ==> B(s, red);
B(s, red) ==> B(s, blue) [ RH(random(0, 360)) RU(30) F(30, 1, 14) ];
```



Usage of imperative code in XL programmes

Commands like the assignment of values to variables, additions, function calls, output (print commands) etc. are specified in the same way like in Java and enclosed in braces { ... } .

Examples:

```
int i;                    // declaration of an integer variable with name i
float a = 0.0;            // declaration and initialization of a floating-point var.
int[] x = new int[20];    // declaration of an array
                          // of length 20; access: x[0], ..., x[19]
float[] y = { 0.1, 0.2, 0.7, -1.4 };
                          // declaration and initialization of an array
i = 25;     // assignment
i++;        // i is incremented by 1
i--;        // i is decremented by 1
```

Nesting of rule-oriented blocks [...] and imperative blocks { ...}

```
module A(float len) extends Sphere(0.1);

int time;

protected void init()
[
    { time = 0; }

    Axiom ==> A(1);
]

public void run()
[
    {
        time++;
        println("Time: " + time);
    }

    A(x) ==> F(x) [RU(30) RH(90) A(x*0.8)] [RU(-30) RH(90) A(x*0.8)];
]
```

Nesting of rule-oriented blocks [...] and imperative blocks { ...}

```
module A(float len) extends Sphere(0.1);

int time;

protected void init()
[
    { time = 0; }

    Axiom ==> A(1);
]

public void run()
[
    {
        time++;
        println("Time: " + time);
    }

    A(x) ==> F(x) [RU(30) RH(90) A(x*0.8)] [RU(-30) RH(90) A(x*0.8)];
]
```

```
// alternatively:
protected void init()
{
    time = 0;

    [
        Axiom ==> A(1);
    ]
}
```

usage of imperative code (continued)

```
i += 5;                // i is incremented by 5
i -= 5;                // i is decremented by 5
i *= 2;                // i is doubled
i /= 3;                // i gets the value i/3
n = m % a;             // n gets assigned the rest of m from integer division by a
x = Math.sqrt(2);      // x gets assigned the square root of 2
if (x != 0) { y = 1/x; }  // conditional assignment of 1/x to y
while (i <= 10) { i++; }  // loop: as long as i ≤ 10,
                          // i is incremented by 1
for (i = 0; i < 100; i++) { x[i] = 2*i; }  // imperative
                                           // for-loop
if (i == 0) { ... }    // test for equality ( „=“ would be assignment!)
```

The most important primitive data types:

`int`	integers
`float`	floating-point numbers
`double`	floating-point numbers, double precision
`char`	characters
`void`	void type (for functions which return no value)

More detailed overview:

type	*range of values*
`boolean`	`true` oder `false`
`char`	16-Bit-Unicode-Zeichen (0x0000 … 0xFFFF)
`byte`	-2^7 bis 2^7 - 1 (-128 … 127)
`short`	-2^15 bis 2^15 - 1 (-32.768 … 32.767)
`int`	-2^31 bis 2^31 - 1 (-2.147.483.648 … 2.147.483.647)
`long`	-2^63 bis 2^63 - 1 (-9.223.372.036.854.775.808 … 9.223.372.036.854.775.807)
`float`	1,40239846E-45f … 3,40282347E+38f
`double`	4,94065645841246544E-324 … 1,79769131486231570E+308

http://openbook.galileocomputing.de/javainsel/javainsel_02_003.html *mathematical constants:*

`Math.PI`	π
`Math.E`	e

logical operators:

`&&`	and
`||`	or
`!`	not

mathematical functions:

`Math.abs`	absolute value	**`Math.sqrt`**	square root
`Math.acos`	arcus cosine	**`Math.tan`**	tangens
`Math.asin`	arcus sine	**`Math.toDegrees`**	
`Math.atan`	arcus tangens	**`Math.toRadians`**	
`Math.cos`	cosine		conversion degrees $\leftrightarrow$ radians
`Math.exp`	exponential function e^x		
`Math.log`	natural logarithm		
`Math.max`	maximum of two numbers		
`Math.min`	minimum of two numbers		
`Math.round`	functin for rounding		
`Math.sin`	sine		

`sm_progbsp01.rgg`: writes the numbers from 1 to 10 to the GroIMP console

```
protected void init()
{
  int i;
  for (i=1; i<= 10; i++)
     println(i);
  println("end.");
}
```

sm_progbsp02.rgg: writes odd square numbers

```
protected void init()
{
   int a, b;
   for (a = 1; a <= 10; a++)
     {
     b = a*a;
     if (b % 2 != 0) println(b);
     }
   println("end.");
}
```

sm_progbsp03.rgg: writes the Fibonacci numbers

```
protected void init()
{
   int i;
   int[] fibo = new int[20]; /* array declaration */
   fibo[0] = fibo[1] = 1;
   for (i=2; i <= 19; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
   for (i=0; i <= 19; i++)
      println(fibo[i]);
   println("end.");
}
```

sm_progbsp04.rgg: Usage of a function

```
/*  a simple imperative programme:
A function written by the user calculates x^2 + 1;
this is evaluated for x from 0 to 1 in steps by 0.1.
Be aware of rounding errors and of the correct upper limit for x.  */

public float function(float x)
   {
   return x*x + 1;
   }
protected void init()
   {
   float a = 0.0;              /*  floating point number  */
   while (a <= 1.00001)
      {
      println(function(a));  /*  apply function and print  */
      a += 0.1;              /*  increment a */
      }
   println("end.");
   }
```

test the examples

`sm09_e20.rgg` usage of arrays

`sm09_e22.rgg` for-loop for lateral branches

Remember:

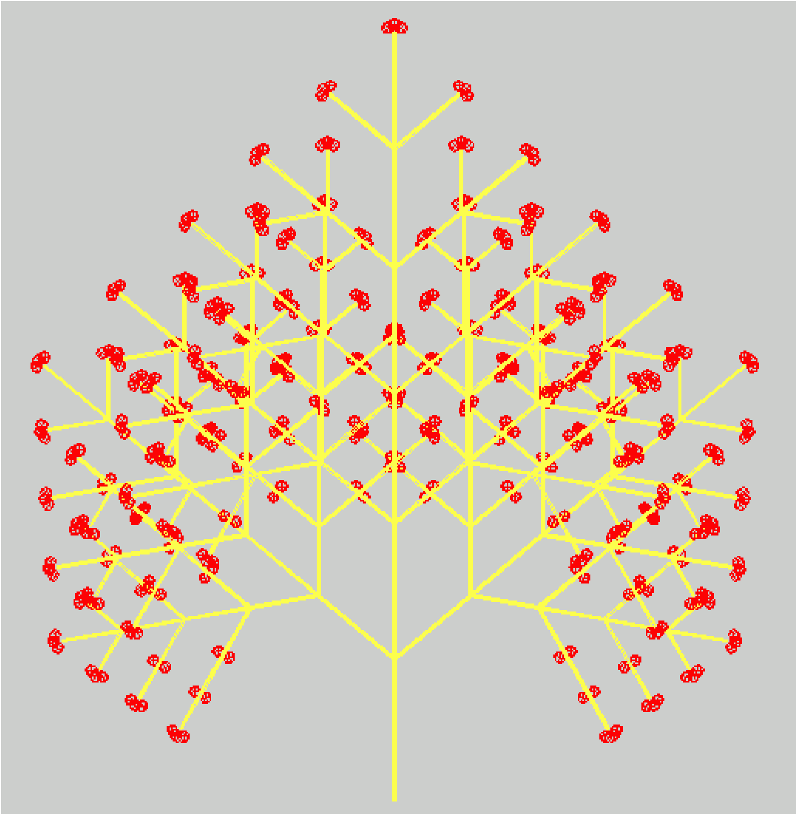
parameters can be checked in conditions
(logical conditions with Java syntax):

```
A(x, y) (x >= 17 && y != 0) ==>  ....
```

test the examples

`sm09_e11.rgg` conditions for rule applications

`sm09_e13.rgg` connection of two conditions with logical “and“ (**&&**)



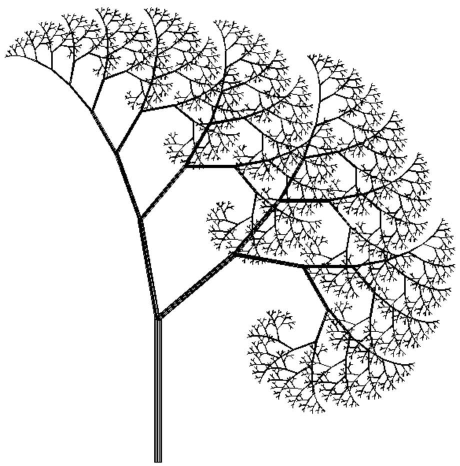
Stochastic L-systems

usage of pseudo-random numbers

Example:

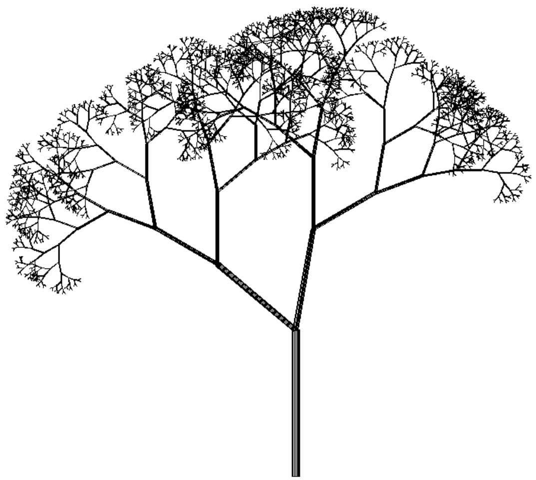
deterministic

```
Axiom ==> L(100) D(5) A;

A ==> F0 LMul(0.7) DMul(0.7)
  [ RU(50) A ] [ RU(-10) A ];
```



stochastic

```
Axiom ==> L(100) D(5) A;

A ==> F0 LMul(0.7) DMul(0.7)
  if (probability(0.5))
   ( [ RU(50) A ] [ RU(-10) A ] )
  else
   ( [ RU(-50) A ] [ RU(10) A ] );
```



XL functions for pseudo-random numbers:

`Math.random()`	generates floating-point random number between 0 and 1 (uniformly distributed)
`random(a, b)`	generates floating point random number between **`a`** and **`b`** (uniformly distributed)
`irandom(j, k)`	generates integer random number between **`j`** and **`k`** (uniformly distributed)
`normal(m, s)`	generates normally-distributed random numbers with mean **`m`** and standard deviation **`s`**
`probability(x)`	gives 1 with probability x, 0 with probability 1–x

XL functions for pseudo-random numbers:

`Math.random()`	generates floating-point random number between 0 and 1 (uniformly distributed)
`random(a, b)`	generates floating point random number between **`a`** and **`b`** (uniformly distributed)
`irandom(j, k)`	generates integer random number between **`j`** and **`k`** (uniformly distributed)
`normal(m, s)`	generates normally-distributed random numbers with mean **`m`** and standard deviation **`s`**
`probability(x)`	gives 1 with probability x, 0 with probability 1–x
`setSeed(n)`	determines a start value for the pseudo-random number generator

XL functions for pseudo-random numbers:

`Math.random()`	generates floating-point random number between 0 and 1 (uniformly distributed)
`random(a, b)`	generates floating point random number between **`a`** and **`b`** (uniformly distributed)
`irandom(j, k)`	generates integer random number between **`j`** and **`k`** (uniformly distributed)
`normal(m, s)`	generates normally-distributed random numbers with mean **`m`** and standard deviation **`s`**
`probability(x)`	gives 1 with probability x, 0 with probability 1–x

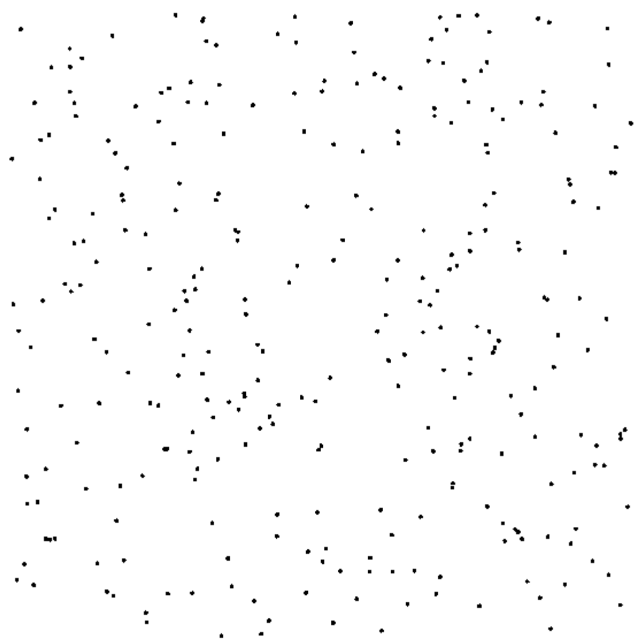
test the example

`sm09_b19.rgg` stochastic L-system

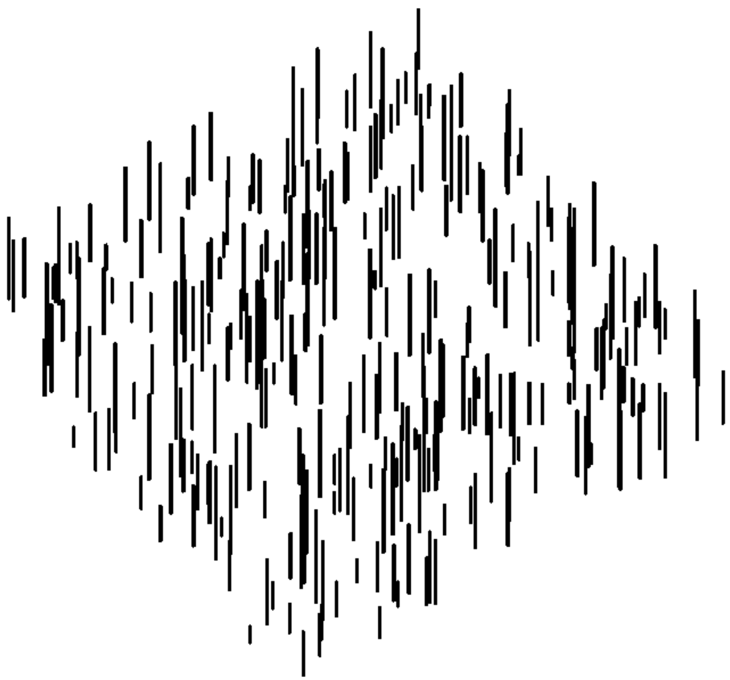
How to create a random distribution in the plane:

```
Axiom ==> D(0.5) for ((1:300))
            ( [ Translate(random(0, 100), random(0, 100), 0)
                F(random(5, 30)) ] );
```

view from above



oblique view



How to create a GroIMP project with textures from graphics files (e.g., photos of leaves or bark)

1. File → New → RGG Project

2. insert name of the RGG file (text file)

3. delete the default programme from the GroIMP editor, write new programme or insert it from another source file

4. store file in the editor (automatic compilation must be successful)
   - textured objects are still shown in simplified form (without textures)

5. Panels → Explorers → 3D → Shaders → Object → New → Lambert

6. click twice on the name „Lambert“ (with delay between the clicks) (or F2), overwrite it with the name which is foreseen in the programme (argument of the function „shader(...)“), finish with <return> (don‘ forget this!!)

7. doubleclick on sphere icon → Attribute Editor opens

8. click there on: Diffuse colour → Surface Maps → Image

9. click there on: Image [?] → From File

how to create a project
(continued)

10. choose image file, „open“
11. „Add the file“: OK
12. store editor file again / compile

    - textured objects are now shown with texture

13. to store the complete project:

    File $\rightarrow$ Save, write name of the project (must not necessarily coincide with the name of the RGG source code file).

A simple GroIMP project with textures:

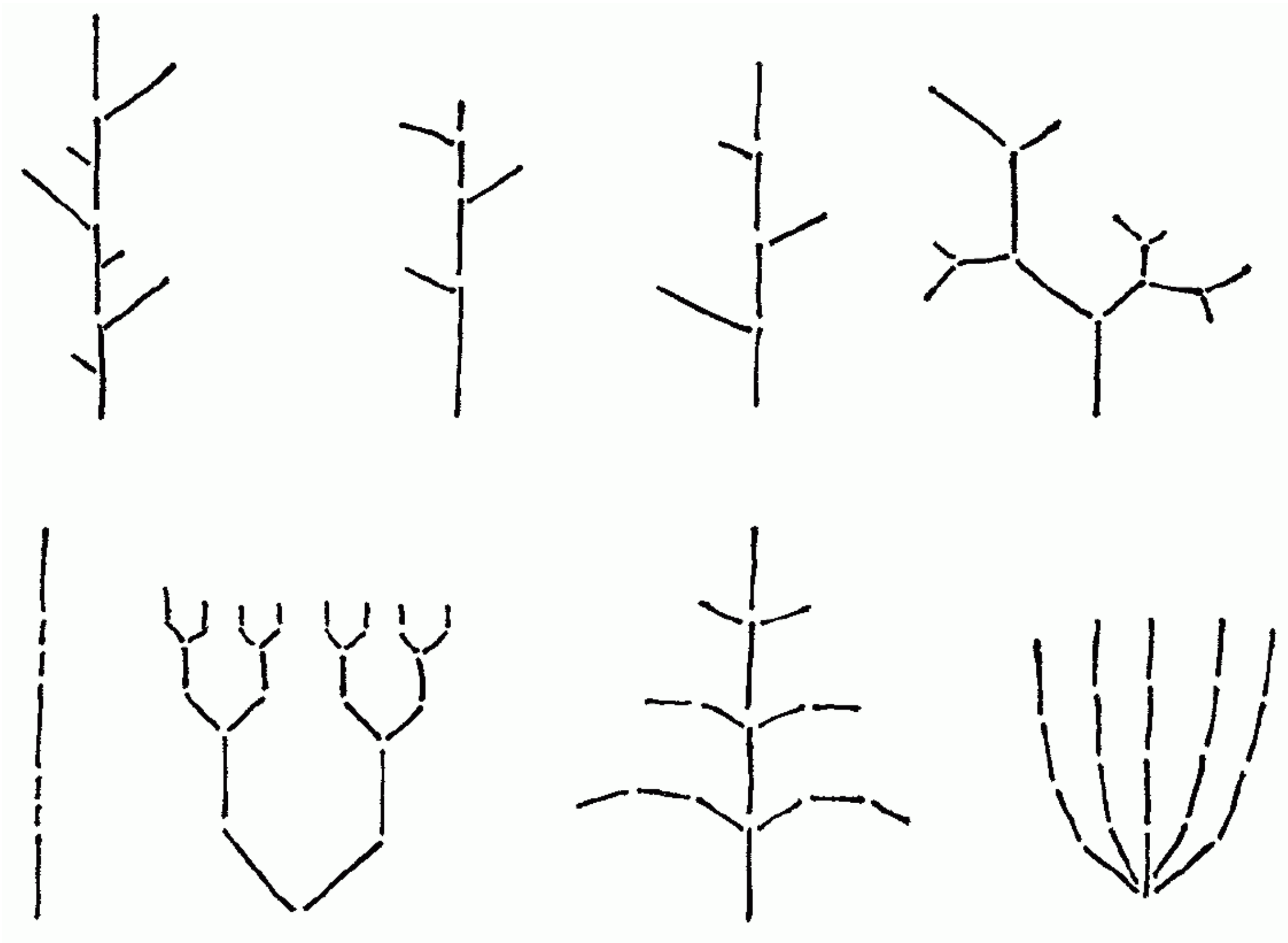
```
module Tree extends Parallelogram(3, 2)
{
    { setShader(treeShader); }
}

const ShaderRef treeShader = shader("tree");

protected void init()
[
    Axiom ==> Tree;
]
```

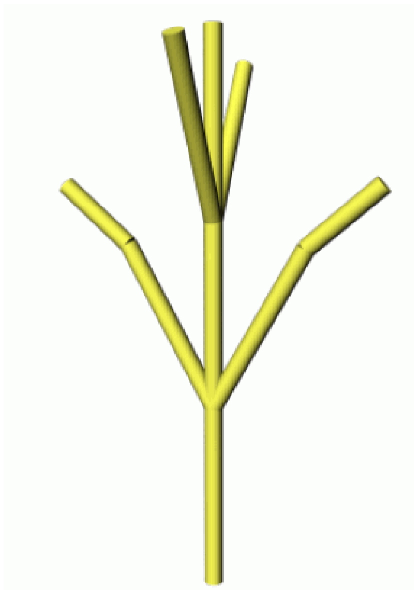
test the example

`sm09_e10.gsz` usage of a surface texture (leaf texture)

Suggestions for team session:

Create the following simple branching patterns with XL and add colours to their elements





Suggestions for team session (2):

Create the following patterns as textured structures with XL