5. Foundations of programming

Paradigms of programming:

Different viewpoints and ways of thinking about how to conceive a computer and a programme

Imperative paradigm:
Computer = machine for the manipulation of variables
Programme = sequence of commands which change values of variables, together with specifications of the *control flow* (telling which command is executed next)
Languages: Fortran, Pascal, Basic, C ...

Example (works in C or Java or XL):

```
x = 0;
while (x < 100)
   x = x + 2;
```

The variable **x** is used to produce the even numbers from 0 to 100.
Attention: The *assignment command* **x = x + 2** is not a mathematical equality!

Object-oriented paradigm:

Computer = environment for virtual objects which are created and destroyed during runtime (and can interact)
Programme = collection of general descriptions of objects (so-called *classes*), together with their hierarchical dependencies (*class hierarchy*)
Objects can contain data and functionality (*methods*)
Languages: Smalltalk, C++, Java, ...

Example (in Java):

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

Typical: class (**`Car`**) with data (**`name`**, **`places`**) and methods (**`print_data`**). The class **`Car`** *inherits* further data and methods from a superclass, **`Vehicle`**.

Rule-based paradigm:

Computer = machine which transforms a given structure according to given rules

Programme = set of transformation rules (sometimes also called a *grammar*)

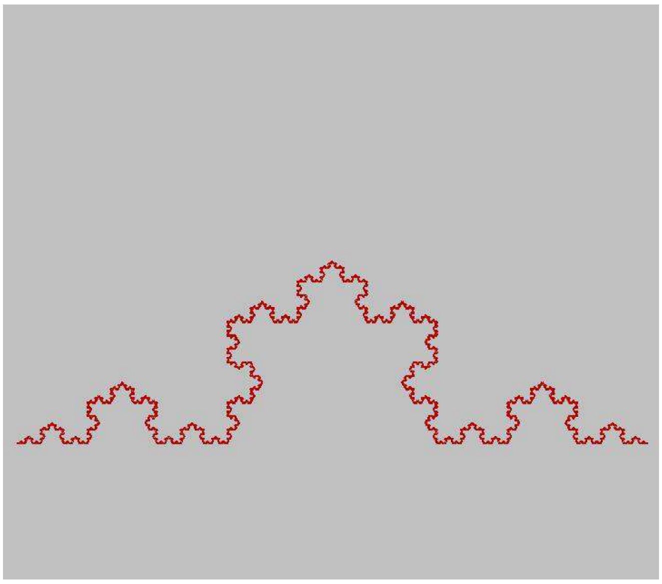
Each step of programme application consists of two substeps: Finding an applicable rule (*matching step*) and transformation of the current structure according to that rule (*rewriting step*).

Languages: Prolog, AI-languages, L-system languages, particularly XL

Example (in XL):

```
public void apply()
   [
   F(x)  ==> F(x/3) RU(-60) F(x/3) RU(120)
             F(x/3) RU(-60) F(x/3);
   ]
```

produces the so-called Koch curve:



Readability of programmes by humans

programmes: have to be executed by computers, but also *to be understood by humans*

Executability can be checked automatically, understandability not!

⇒ Recommendations:

- make frequent use of programme comments (**`/* ... */`** or **`// ...`** in Java, C++ or XL)
- use plenty of newlines and blanks
- put braces { ... } in lines of their own, put matching braces in same horizontal position:

```
{
  ....
}
```

- *indentation* makes containment and nesting of programme components visible
- avoid long lines, insert line breaks for readability
- avoid very long methods
- use "speaking" variable and function names (**`int iteration_counter`** is better than **`int x127`** !)
- do not use variable names twice for different purposes, even if the language allows it
- Initialise constants, default values etc. at the beginning of a source code file, not somewhere "deep in the code" where you don't find them later on
- *adhere to conventions used by competent programmers!*

Basic parts of Java and XL

Remark: The language XL is an extension of Java. The following examples can be compiled and run with GroIMP (see **`www.grogra.de`**), a modelling platform which contains a development toolkit for XL and possibilities for visualization.

A first demonstration programme:

```
/* A simple Java programme for execution
with the GroIMP software. */

protected void init()
   {
   println("Hello World!");
   }
```

(= example file **`prog_ex01.rgg`**)

Download of GroIMP:
`https://sourceforge.net/projects/groimp/`

Basic components

Comments, spaces, newline: For human readability, and for separating words (just like in normal written language).

Special symbols: To denote different kinds of groupings, to terminate commands, to construct paths etc.

Examples: Braces `{`, `}`; parentheses `(`, `)`; brackets `[`, `]`; dot; double-quotes `"`; semicolon

Literal values: character sequences representing a value directly, like a digit sequence for a number, or a character sequence in double quotes for a string.

Example: `"Hello World!"`

Sequences of letters or digits, starting with a letter: different categories: **1) Keywords**, **2) predefined identifiers**, **3) newly declared identifiers**.

1) Keywords: Are fixed in the language proper, can not be given a new meaning

Examples: `public`, `class`, `static`, `void` , `protected`

2) Predeclared identifiers: Meaning fixed by a declaration in the context, often can be "overwritten", i.e. given a new meaning. Examples:

`String`: data type for character sequences

`println`: predefined method – invoked with a string as its argument, it writes the string to the GroIMP console (a special output window) and adds a line feed.

3) newly declared identifiers: Their meaning is fixed by (explicit or implicit) declarations in the programme itself. Example: **`init`** is the name of the method which writes the text to the console. It expects no arguments (**`init()`**).

Use of simple data types and the "while" loop

```
/* A simple demonstration program,
   printing out the numbers from 0 to 10
   and their squares, each pair
   on an extra line. */

protected void init()
   {
   int i;
   i = 0;
   while (i <= 10)
      {
      println(i + ": " + (i*i));
      i = i+1;
      }
   println("Finished!");
   }
```

(example file **`prog_ex02.rgg`**)

While loop

`while` starts a **loop**: A sequence of commands which, under some condition, are executed repeatedly.

First, the condition given in parentheses is checked. Result must be boolean. **Our example**: Comparison of the current value of `i` (0) with `10`.

`0<10` is true: Thus, the body of the loop is executed: Pair of values 0 and 0*0 are printed, and `i` is incremented by one.

Then, execution continues with the check of the condition, and the loop is repeated until `i` has value `11`, such that `i <= 10` becomes false.

Then, the loop body is not repeated again, and the `main` method finishes.

Assignments

In our example:

`i = 0;`

the variable named **`i`** gets the new value 0

- fundamental operation in the imperative programming paradigm

effect: content of a place in the memory is changed

Attention:

`i = 0` in a Java programme does not have the same meaning as in a mathematical formula!

E.g., **`i = i+1`** would mathematically be a contradiction (it would imply 0 = 1)

– but makes sense in a programme (increment **`i`** by 1).
Mathematical meaning of this assignment:

$$i_{new} = i_{old} + 1.$$

In assignments, the *order is relevant*:
`x1 = x2;` has another effect as **`x2 = x1;`**

To underline the asymmetry, other languages (e.g., Pascal) use **`:=`** instead of **`=`** for assignments.

XL allows both notations
(but with a slightly different meaning: **`:=`** denotes a deferred assignment, i.e., it enables a quasi-parallel execution with other assignments.)

Comparison (checking for equality) is expressed in Java, C and XL by **`==`**

Java offers further assignment operators besides = :
`a += b` // add content of **`b`** to the content of **`a`**
`–=, *=, /=` etc. analogously.

Data types:

describe sets of values and the operations which can be performed on them.

Example: integers, with arithmetical operations (+, –, *, /, %) and comparisons (<, <=, >, >=, ...).

In the example programme: **`int`**, **`String`**.

`int`: type of 32-bit two's complement integers. The variable **`i`** used for running through the argument list has this type.

`i` starts with value 0 and is incremented in the loop until it has value 11.

`String`: type of character sequences. **`println`** expects a variable of this type as its argument.

Numbers are implicitly converted to strings here. Concatenation of strings by +.

("*Operator overloading*": different meanings of + for numbers and for strings.)

Literals

Literals denote values directly

String literals: Strings in quotes

Used character code for the string content: 16-bit Unicode

Special characters in strings: `\`: is used to introduce something "special". Examples:

\uXXXX (**XXXX**: up to four hexadecimal digits): The number of a Unicode character

`\n`: a line break; `\t`: a tabulator; `\xxx`, `xxx` a three-digit n octal number: The character with the given octal code.

Number literals: Signed digit sequence for integer types; for float types: decimal point and "E"-Notation. Examples: +3453; 3.141592653; 1.17E-6

Primitive Java data types:

primitive data type	defaults	size (bits)	min/max
`boolean`	`false`	1	n.a./n.a.
Unicode characters:			
`char`	`\u0000`	16	`\u0000`/`\uFFFF`
Two's complement integers:			
`byte`	0	8	-128/127
`short`	0	16	-32768/32767
`int`	0	32	-2147483648/2147483647
`long`	0	64	-9223372036854775808/ 9223372036854775807
IEEE 754 floating-point numbers: (min/max are those of absolute values)			
`float`	0.0	32	1.4023985E-45/3.40282347E+38
`double`	0.0	64	4.94065645841246544E-324/ 1.79769313486231570E+308

`void`: quasi-type for methods which return no value

Non-primitive Java data types: Arrays and objects

Arrays: collections of elements of the same type, accessed by **number** (from 0). Example declarations of integer arrays:

```
int[] p = {1,3,2,10};
int[] q = new int[5];
int[] r;
```

Values after these declarations:

`p` points to a memory block of four integers, with values 1, 3, 2 and 10.

`q` points to a memory block of five integers, all values 0.

`r` does not point anywhere (it has the special value `null`). This can be changed by the allocation of a block of memory via the Java operation `new`:

```
r = new int[1000];
```

Now, `r` points to a memory block of 1000 integers, all 0.

```
r = p;
```

Now, `r` points to the same memory block as p.

Array declarations and operations

Non-allocating declaration: `int[] a_empty;`

Allocated with room for 10 elements: `int[] a_ten = new int[10];`

Initialized array: `int[] lookup = {1,2,4,8,16,32,64,128};`

Multiple dimensions: `boolean[][] bw_screen = new boolean[1024][768];`

Non-rectangular: `int[][] pascal_triangle = {{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1},{1,5,10,10,5,1}};`

Array access: by integer-index in brackets. Start at 0. Array-access is checked (index may not be negative or too large)

Number of elements of array `a`: `a.length`

Objects: collections of elements of arbitrary types, plus associated operations, accessed by **name**.

Object types must be **declared** before they can be used; example:

```
class color {
  String name;
  float red;
  float green;
  float blue;
}
```

Use of object types

```
// Declare three color variables.
color r,w,b;

// Initialize the color variables to red, white and black.
r = new color;
r.name = "Red";   r.red = 1.0; r.green = 0.0; r.blue = 0.0;
w = new color;
w.name = "White"; w.red = 1.0; w.green = 1.0;  w.blue = 1.0;
b = new color;
b.name = "Black"; b.red = 0.0; b.green = 0.0; b.blue = 0.0;
```

Both non-primitive data types are handled **by reference**: The variable content is just the address of a memory block.

An assignment to such a variable only changes this address, **not the data of the memory block**.

`null` is the default value for reference types

Java operators

<table>
<tr><th>Prec</th><th>Operators</th><th>types</th><th>assoc.</th><th>meaning</th></tr>
<tr><td>1</td><td>++</td><td>arithmetic</td><td></td><td>pre- or post-increment</td></tr>
<tr><td></td><td>--</td><td>arithmetic</td><td></td><td>pre- or post-decrement</td></tr>
<tr><td></td><td>+,-</td><td>arithmetic</td><td></td><td>unary plus or minus</td></tr>
<tr><td></td><td>~</td><td>integral</td><td></td><td>bit complement</td></tr>
<tr><td></td><td>!</td><td>boolean</td><td></td><td>logical not</td></tr>
<tr><td></td><td>(type)</td><td>any</td><td></td><td>typecast</td></tr>
<tr><td>2</td><td>*,/,%</td><td>arithmetic</td><td>L</td><td>multiplication, division, remainder</td></tr>
<tr><td>3</td><td>+,-</td><td>arithmetic</td><td>L</td><td>addition, subtraction</td></tr>
<tr><td></td><td>+</td><td>String</td><td>L</td><td>concatenation</td></tr>
<tr><td>4</td><td><<</td><td>integral</td><td>L</td><td>shift bits left</td></tr>
<tr><td></td><td>>></td><td>integral</td><td>L</td><td>shift bits right, filling with sign</td></tr>
<tr><td></td><td>>>></td><td>integral</td><td>L</td><td>shift bits right, filling with zero</td></tr>
<tr><td>5</td><td><,<=,>,>=</td><td>arithmetic</td><td></td><td>comparisons</td></tr>
<tr><td></td><td>instanceof</td><td>object, type</td><td></td><td>type comparison</td></tr>
<tr><th>Prec</th><th>Operators</th><th>types</th><th>assoc.</th><th>meaning</th></tr>
<tr><td>6</td><td>==, !=</td><td>any</td><td>L</td><td>equality, inequality</td></tr>
<tr><td>7</td><td>&</td><td>integral</td><td>L</td><td>bitwise AND</td></tr>
<tr><td></td><td>&</td><td>boolean</td><td>L</td><td>boolean AND</td></tr>
<tr><td>8</td><td>^</td><td>integral</td><td>L</td><td>bitwise XOR</td></tr>
<tr><td></td><td>^</td><td>boolean</td><td>L</td><td>boolean XOR</td></tr>
<tr><td>9</td><td>|</td><td>integral</td><td>L</td><td>bitwise OR</td></tr>
<tr><td></td><td>|</td><td>boolean</td><td>L</td><td>boolean OR</td></tr>
<tr><td>10</td><td>&&</td><td>boolean</td><td>L</td><td>short-circuit AND</td></tr>
<tr><td>11</td><td>||</td><td>boolean</td><td>L</td><td>short-circuit OR</td></tr>
<tr><td>12</td><td>?:</td><td>boolean,any,any</td><td></td><td>conditional selection</td></tr>
<tr><td>13</td><td>=</td><td>variable, any</td><td>R</td><td>assignment</td></tr>
<tr><td></td><td>*=, /=, %=
+=, -=, <<=
>>=, >>>=, &=
^=, |=</td><td>variable, any</td><td>R</td><td>operation and assignment</td></tr>
</table>

("assoc" = order of association, i.e., evalutation from left (L) or right (R) when several operators of the same level occur in the same expression)

Functional abstraction, self-defined methods

Phenomenon to deal with: repetition of **identical or almost identical code fragments** – especially if these fragments are quite long.

Problems:

(1) Changes in the code **have to be repeated for each occurrence** of the code fragment.

(2) Code cannot occur in itself – **recursive algorithms cannot be coded directly**.

Solution: **methods** (in OO-languages) and **procedures and functions** (in non-OO languages).

Methods can be used like **extensions** of the language.

Example: compute maximum of two integers

```
int max(int p1, int p2)
{
  return (p1>p2 ? p1 : p2);
}
```

Use of the method:

```
int a, b;

int x;

x = max(a,b);
```

Example: compute the factorial of an integer

Reminder: "factorial" n! = n * (n–1) * ... * 3 * 2 * 1.

Recursion: Compute factorial

```
int fac(int i)
{
  if(i<=1)
  {
    return 1;
  }
  else
  {
    return i*fac(i-1);
  }
}
```

For this problem, **nobody would use recursion**! A simple `while`-loop would suffice. Recursion can be unnecessarily **inefficient**.

Example (**prog_ex03.rgg**): Usage of compound data structures (*arrays*)

```
/* Computation of the sum of elements of
an integer array. */

protected void init()
   {
   int result = 0;
   int[] p = { 4, 3, 3, 5, 15 };
         /* initialization of an array */

   int i = 0;
   while (i < p.length)
      {
      result += p[i];
      i = i+1;
      }
   println("The sum is: " + result);
   }
```

The same as an extra method:

Example: compute the sum of the elements of an array:

```
int computeSum(int[] p)
{
  // This variable accumulates the result.
  int r = 0;

  // This variables points to the different positions in (p),
  // starting at 0 and running to the end.
  int i = 0;

  // Run with (i) through (p), accumulating the sum of elements in
  // (r).
  while(i < p.length)
  {
    r = r + p[i];
    i = i + 1;
  }

  // Return result.
  return r;
}
```

Questions regarding `computeSum`: Details are important!

Does it work for empty `(p)`?

Is `<` the right comparison in the condition of the `while` clause, or would `<=` be right?

Should `i` start with another value than 0?

How could a solution look like in which `i` runs through `p` in the opposite direction?

General structure of method declaration (incomplete version)

```
<type> <methodName> ( <parameterlist, empty for no parameters> )
{
  <method body, including ''return <expression>''>
}
```

Method interface: type of return value, name of method, and types and names of parameters.

Method body: code fragment performing the work.

`return` **statement**: Execution **leaves the method** and **returns the value of the expression** as result.

Problems solved:

(1) Similar code **does not have to be repeated** – where it is needed, it is just **invoked** or **called** with the proper parameters. Changes only have to be done **once**.

(2) Recursion can be **coded directly**.

Further consequences:

(3) Functionality of code fragments can be **documented by giving a symbolic name** to a code fragment.

(4) Code fragments **are usable without that all the details are known** – only knowledge about the **interface** and the **I/O-behavior** is necessary. Consequence: Implementation can be changed.

Method call:
e.g. **`x = max(a, b);`**

Effects:

- control flow jumps from the place where the method is called to the place where the method is defined
- the method is executed
- the control flow jumps back to the place where the method was called and the return value is assigned to **`x`**.

Control structures of Java

control structures:
language concepts designed to control the flow of operations
– typical for the imperative programming paradigm

particularly: *branching* of the programme; *loops*.

Variants of branching:

```
if(<condition>)
{
  <Code for fulfilled condition>
}
```

(if the condition is false, nothing happens)

```
if (<condition>)
   {
      <Code for fulfilled condition>
   }
else
   {
      <Code for unfulfilled condition>
   }
```

Nesting of **`if...else`** possible:

```
if(<cond1>)
{
  <Code for fulfilled <cond1>>
}
else if(<cond2>)
{
  <Code for non-fulfilled <cond1>, but fulfilled <cond2>>
}
else
{
  <Code to be executed if NO condition is fulfilled>
}
```

Example application: Finding the solutions of a quadratic equation ("pq-formula")

prog_ex04.rgg

```
/* Computation of the solutions of a quadratic
   equation, using a self-defined method */

public double[] solve_quadratic(double p,
                                double q)
   {
   double x = -p/2, y = x*x - q;
   double[] result;

   if (y < 0)
      {
      // term under the square root is
      // negative. No solution.
      result = new double[0];
      }
```

```
   else
      if (y < 1e-20)
         {
         // term under the square root is zero.
         // One solution.
         result = new double[1];
         result[0] = x;
         }
      else
         {
         // term under the square root is
         // positive. Two solutions.
         double z = Math.sqrt(y);
         result = new double[2];
         result[0] = x + z;
         result[1] = x - z;
         }
   return result;
   }

module A(double p, double q) extends Sphere(3);

protected void init()
{
   [
   Axiom ==> A(0, 0);
   ]
   println("Click on object for input (p,q)!");
}

public void calculate()
{
   double[] res;
   double p, q;

   [
   a:A ==> { p = a[p]; q = a[q]; };
   ]
```

```
   res = solve_quadratic(p, q);

   if (res.length == 0)
      println("There is no solution.");
   if (res.length == 1)
      println("Single solution: " + res[0]);
   if (res.length == 2)
      {
      println("First solution: " + res[1]);
      println("Second solution: " + res[0]);
      }
}
```

Loops:

We have already introduced the **`while`** loop.

The **`for`** loop:

```
for(<Initialization>;<Condition>;<Increment>)
{
  <Code to be repeated>
}
```

Similar to:

```
<Initialization>;
while(<Condition>)
{
  <Code to be repeated>
  <Increment>
}
```

Application example:

```
static public int computeSum(int[] p)
{
  int result = 0;

  for(int i=0; i<p.length; ++i)
  {
    result += p[i];
  }

  return result;
}
```

Exercises

1. Write Java expressions for the following mathematical expressions:

(a) $\dfrac{a}{b+\dfrac{1}{c}}+2.5\cdot 10^{6}$

(b) $e^{2k}\cdot\sqrt{x^{2}-2xy+1}$

(c) $z=\begin{cases}1 & \text{if } n \text{ is even}\\ 0 & \text{otherwise}\end{cases}$

(Remark: $\sqrt{x}$ is **`Math.sqrt(x)`**, e^{x} is **`Math.exp(x)`**,
`a % b` gives the rest when dividing **`a`** by **`b`**.)

2. The following Java method **`m`** gets an integer array **`x`** as its argument:

```
public int m(int x[])
   {
   int c, i;
   c = 0;
   for (i = 0; i < x.length; i++)
      if (x[i] % 2 == 1) c++;
   return c;
   }
```

What does this method calculate (or count) ?

3. (a) Which errors can possibly occur during runtime of the following Java program fragment?

```
int i;
float list[300];
float x, y;
...
/*  i, x and y are somehow calculated  */
...
list[i] = 1.5 / (x + y);
...
```

(b) Which conditions (to be specified in Java syntax) should be checked to capture these errors before they can cause trouble?

4. The following Java method **f** gets an integer array **x** and the length **n** of the array as arguments:

```
public int f(int x[], int n)
   {
   int i, k = 0;
   if (n <= 0) return –1;
   i = 1;
   while (i < n)
      {
      if (x[k] > x[i])
          k = i;
      i = i+1;
      }
   return k;
   }
```

(a) What does the method **f** calculate?

(b) What does it give as result if all fields of the array **x** contain the same number, let us say, 1 ?

5. Write an XL (or Java) program which prints all prime numbers between 1 and 1000 on the screen (and no other numbers).

Remark 1: An integer is a prime number if it is larger than 1 and if it is not divisible without rest by any other positive integer except 1 and itself.

Remark 2: **a % b** = rest of the division of integer **a** by integer **b** (0 ≤ (**a % b)** < **b**).

6. Introduction to rule-based simulation

Examples of processes which are studied by simulation on a computer:

- growth and crown development of a plant
- chemical reactions in a cell
- population dynamics of competing tree species
- foraging behaviour of ants
- water flow in the soil
- interception of photosynthetically-active radiation by a canopy
- dynamics of traffic on a road network
- economic decisions of traders on a market
- ...

Different formal systems, programming languages and software platforms are in use which support such simulations.
(See also: NetLogo, in "Ecosystem Modelling")

As an example, we demonstrate here the usage of graph-grammar rules in the language XL to simulate the 3-dimensional development of plants.

XL = eXtended L-system language

L-systems (Lindenmayer systems): rules working on character strings, named after the botanist Aristid Lindenmayer (1925-1989)

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

An L-system 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.

Example:

alphabet {A, B}, start word A

set of rules:

A → B

B → AB

derivation chain:

A → B → AB → BAB → ABBAB → BABABBAB
→ ABBABBABABBAB → BABABBABABBABBABABBAB
→ ...

still missing 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

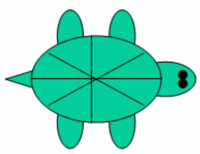
$$\begin{array}{ccccccccc} \alpha & \rightarrow & \sigma_1 & \rightarrow & \sigma_2 & \rightarrow & \sigma_3 & \rightarrow & \ldots \\ & & \downarrow & & \downarrow & & \downarrow & & \\ & & S_1 & & S_2 & & S_3 & & \ldots \end{array}$$

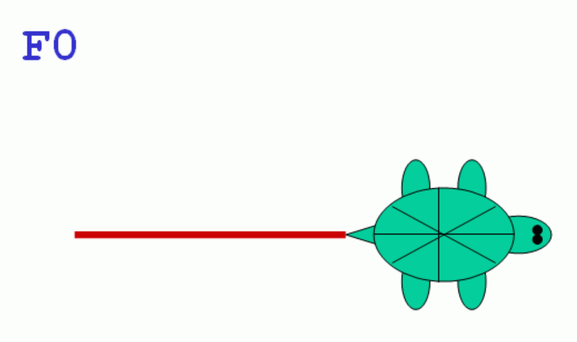
S_1, S_2, S_3, ... can be seen as developmental steps of an object, a scene or an organism.

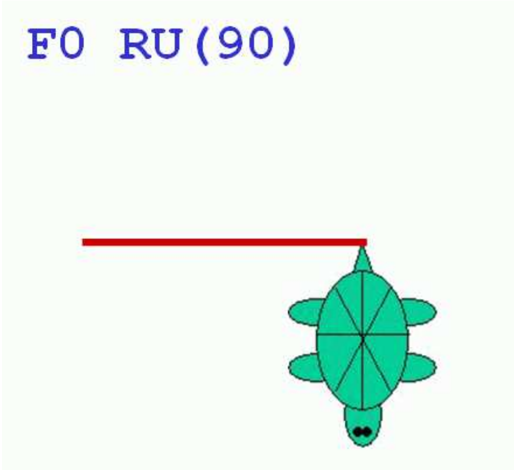
For the interpretation: *turtle geometry*

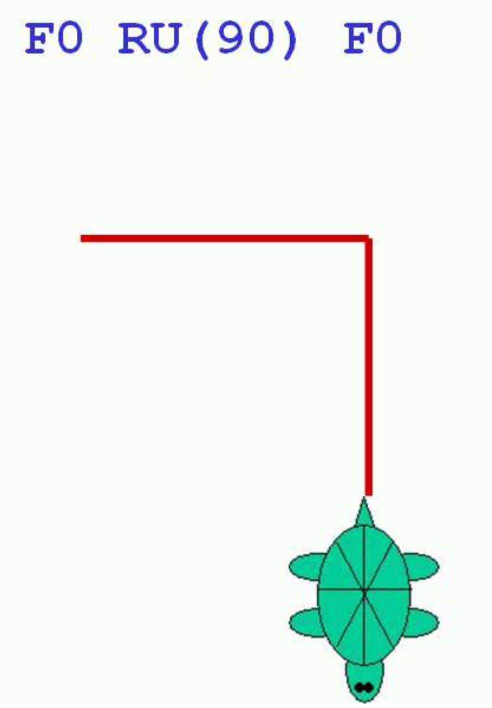
Turtle:

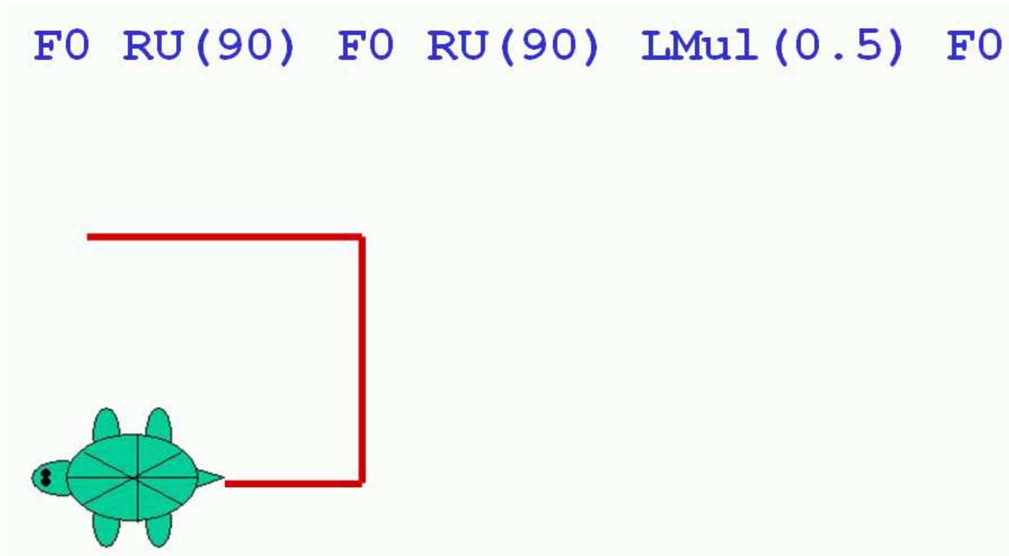
goes according to commands




F0


F0 RU(90)


F0 RU(90) F0


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

„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

Repetition of substrings possible with "`for`"

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

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

Exercise:

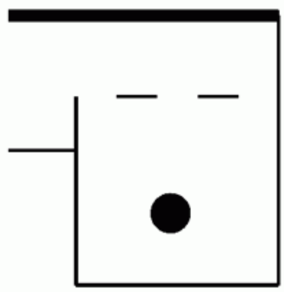
what is the result of the interpretation of

```
L(10) for ((1:6))
        ( F0 RU(90) LMul(0.8) )  ?
```

Example:

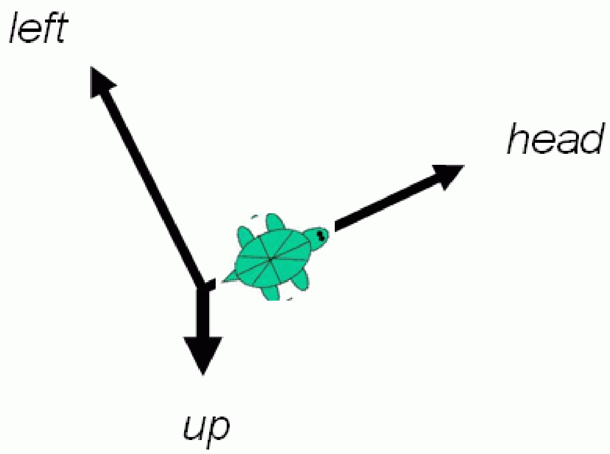
```
L(100) D(3) RU(-90) F(50) RU(90) M0 RU(90) D(10) F0 F0
       D(3) RU(90) F0 F0 RU(90) F(150) RU(90) F(140) RU(90)
       M(30) F(30) M(30) F(30) RU(120) M0 Sphere(15)
```

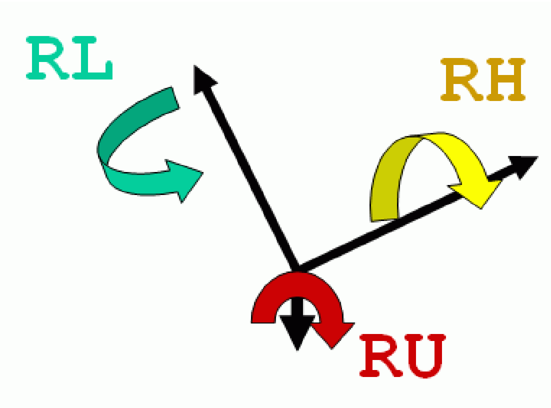
generates



Extension to 3-D graphics:

turtle rotations by 3 axes in space





3-D commands:

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

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

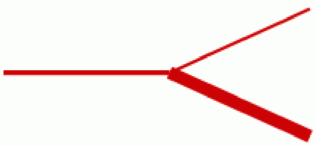
up-, *left*- and *head* axis form an orthogonal spatial coordinate system which is carried by the *turtle*

Branches:
realization with memory commands

[put current state on stack ("Ablage", Stack)

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

```
F0 [ RU(-20) F0 ] RU(20) DMul(2) F0
```

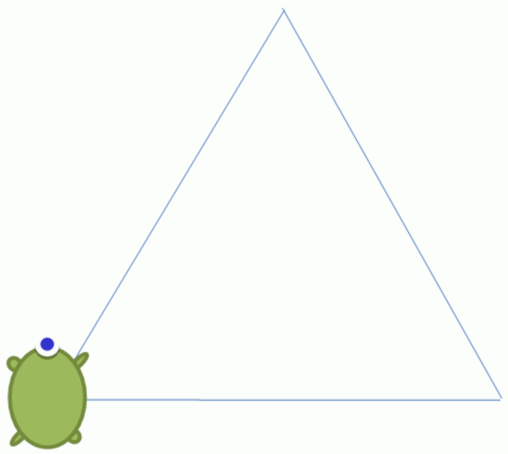


How to execute a turtle command sequence with GroIMP

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

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

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`

now we make the turtle-generated patterns dynamic

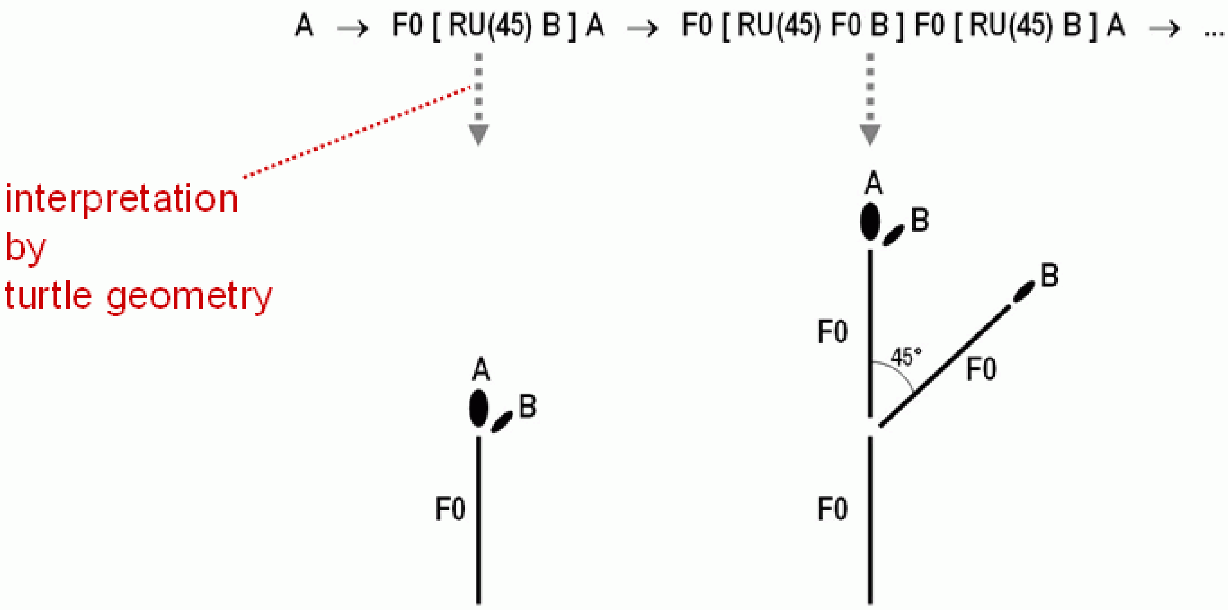
Interpreted L-system:
The alphabet of the L-system contains the turtle command language as a subset.

Example:

rules

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

start word **A**

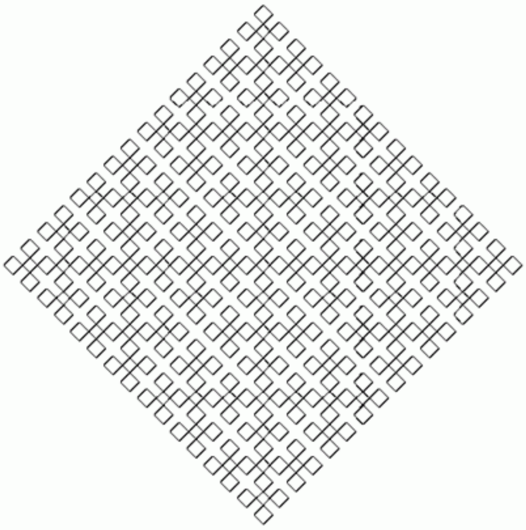


(**A** and **B** are normally not interpreted geometrically.)

also modelling of objects different from plants

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
```

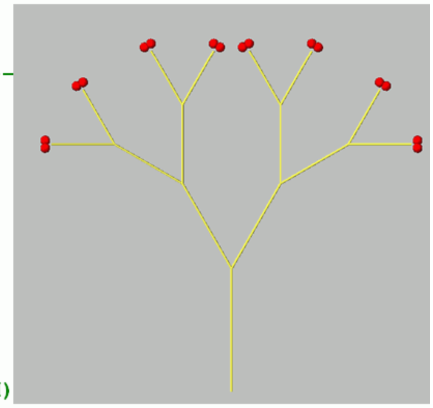


traditional Indian kolam
„Anklets of Krishna“

A simple plant with dichotomous branching:

sample file **sm09_e03.rgg** :

```
/* You learn at this example:
- how to construct a simple plant model (according to architectural model Schoute)
- how to specify branches with [ ]  */

// 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) ];
]
```



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) ==> ....`

Stochastic L-systems

usage of pseudo-random numbers

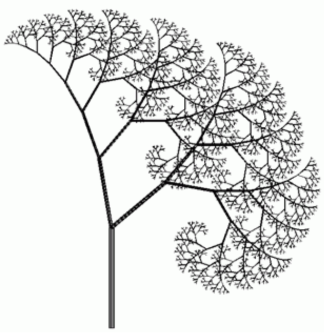
Example:

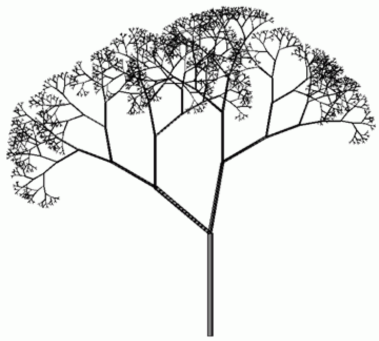
deterministic

```
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 ] );
```



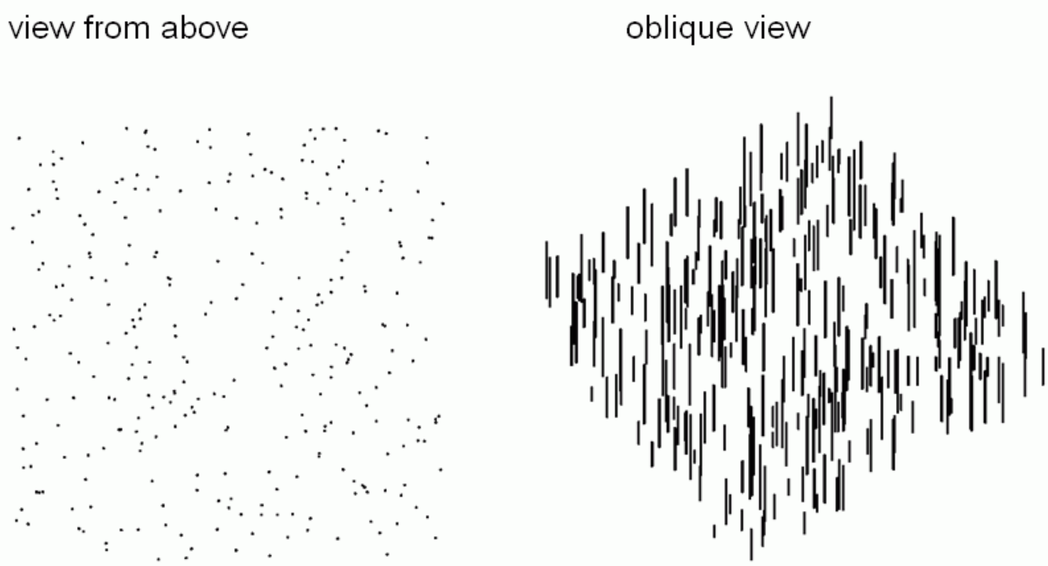


XL functions for pseudo-random numbers:

`Math.random()`	generates floating-point random number between 0 and 1
`random(a, b)`	generates floating point random number between **`a`** and **`b`**
`probability(x)`	gives 1 with probability x, 0 with probability 1–x

How to create a random distribution in the plane:

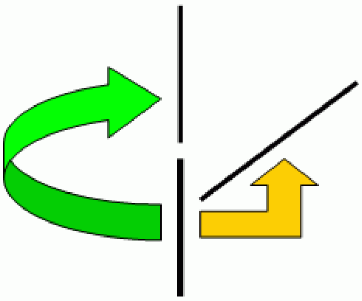
```
Axiom ==> D(0.5) for ((1:300))
          ( [ Translate(random(0, 100), random(0, 100), 0)
              F(random(5, 30)) ] );
```



The step towards graph grammars

drawback of L-systems:

- in L-systems with branches (by turtle commands) only 2 possible relations between objects: "direct successor" and "branch"



extensions:

- to permit additional types of relations
- to permit cycles

→ **graph grammar**

Example of a graph grammar rule:

- each left-hand side of a rule describes a subgraph (a pattern of nodes and edges, which is looked for in the whole graph), which is replaced when the rule is applied.

- each right-hand side of a rule defines a new subgraph which is inserted as substitute for the removed subgraph.

special variant of graph grammars:
Relational growth grammars (RGG)

- parallel application, same as for L-systems
- attributed vertices and edges
- vertex types with object hierarchy (a vertex type can inherit properties from another vertex type)

The language XL

specification: Kniemeyer (2008)

- extension of Java
- allows also specification of L-systems and RGGs (graph grammars) in an intuitive rule notation

imperative blocks, like in Java: { ... }

rule-oriented blocks (RGG blocks): [...]

During execution of an XL program, there is one graph (represented in the computer memory) which is transformed by the rules

- the nodes (vertices) of this graph are basically Java objects
  (they can also be geometrical objects)

Example:
rules for the fractal curve shown previously

```
public void derivation()
  [
  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);
  ]
```

nodes of the graph

edges (type „successor“)

Queries in the graph

a query is enclosed by `(*     *)`

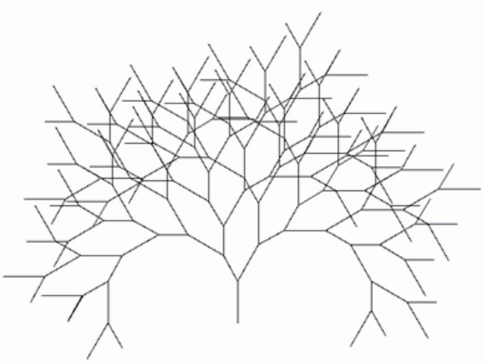
The elements are given in their expected order, e.g.:
`(* A A B *)` searches for a subgraph which consists of a sequence of nodes of the types `A A B`, connected by successor edges.

example for a graph query:

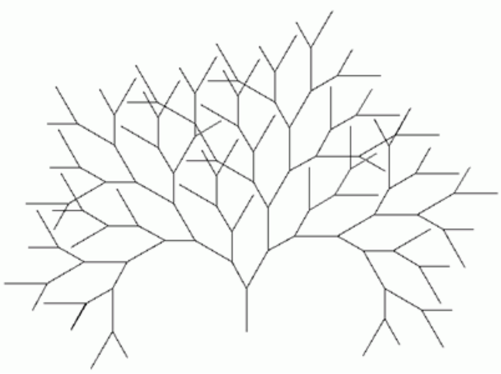
binary tree, growth shall start only if there is enough distance to other `F` objects

```
Axiom ==> F(100) [ RU(-30) A(70) ] RU(30) A(100);
a:A(s) ==> if ( forall(distance(a, (* F *)) > 60) )
           ( RH(180) F(s) [ RU(-30) A(70) ] RU(30) A(100) )
```

without the „if“ condition



with the „if“ condition



A simple functional-structural plant model (FSPM) in XL:
see example file **`sfspm09.gsz`**

includes:
- light emitted from a lamp
- interception of light by the leaves of the plant
- a submodel for photosynthesis
- transport of assimilates along the plant axes
- formation of new internodes and leaves
- growth of the organs
- flowering

executable by GroIMP

The software GroIMP

GroIMP = "growth-grammar related interactive modelling platform"

see **`http://www.grogra.de`**,

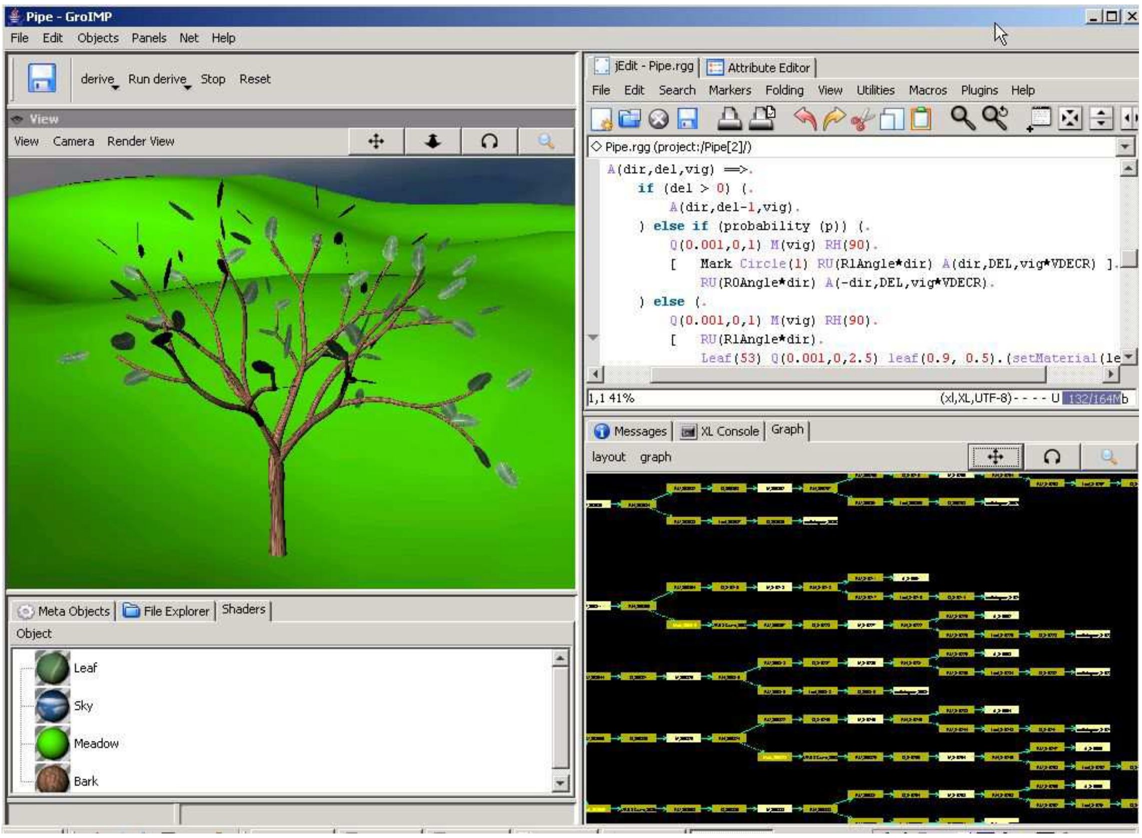
there you find also the link to the download site
`http://sourceforge.net/projects/groimp/`
and a gallery of examples.

See also the learning units about GroIMP (author: K. Petersen, M.Sc. Forest Science), available in StudIP.

GroIMP is an open source project. It combines:

- XL compiler and interpreter
- a development environment for XL
- an interactive 3-d modeller
- several 3-d renderers
- a 2-d graph visualization tool
- an editor for 3-d objects and attributes
- tools for texture generation
- an interface for measured tree architecture data
- a simulation tool for radiation in scenes
- support for solving differential equations in a numerically stable way (for submodels)
- interfaces for data formats like dxf, obj, mtg, pdb
- ...

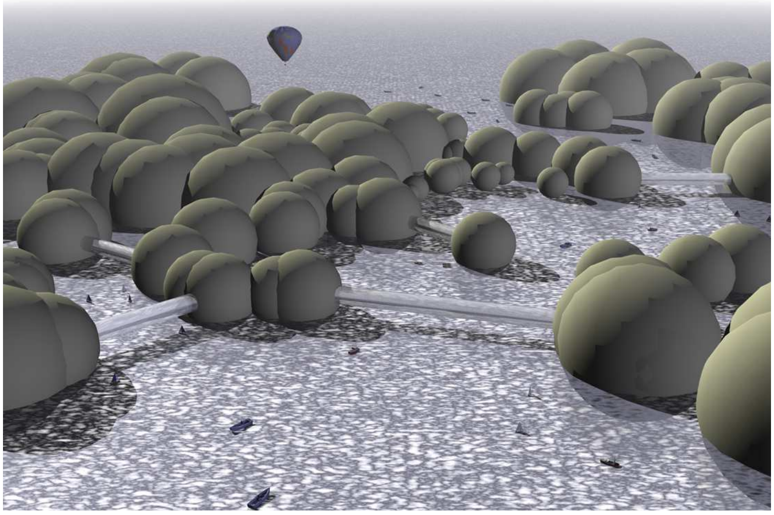
screenshot:



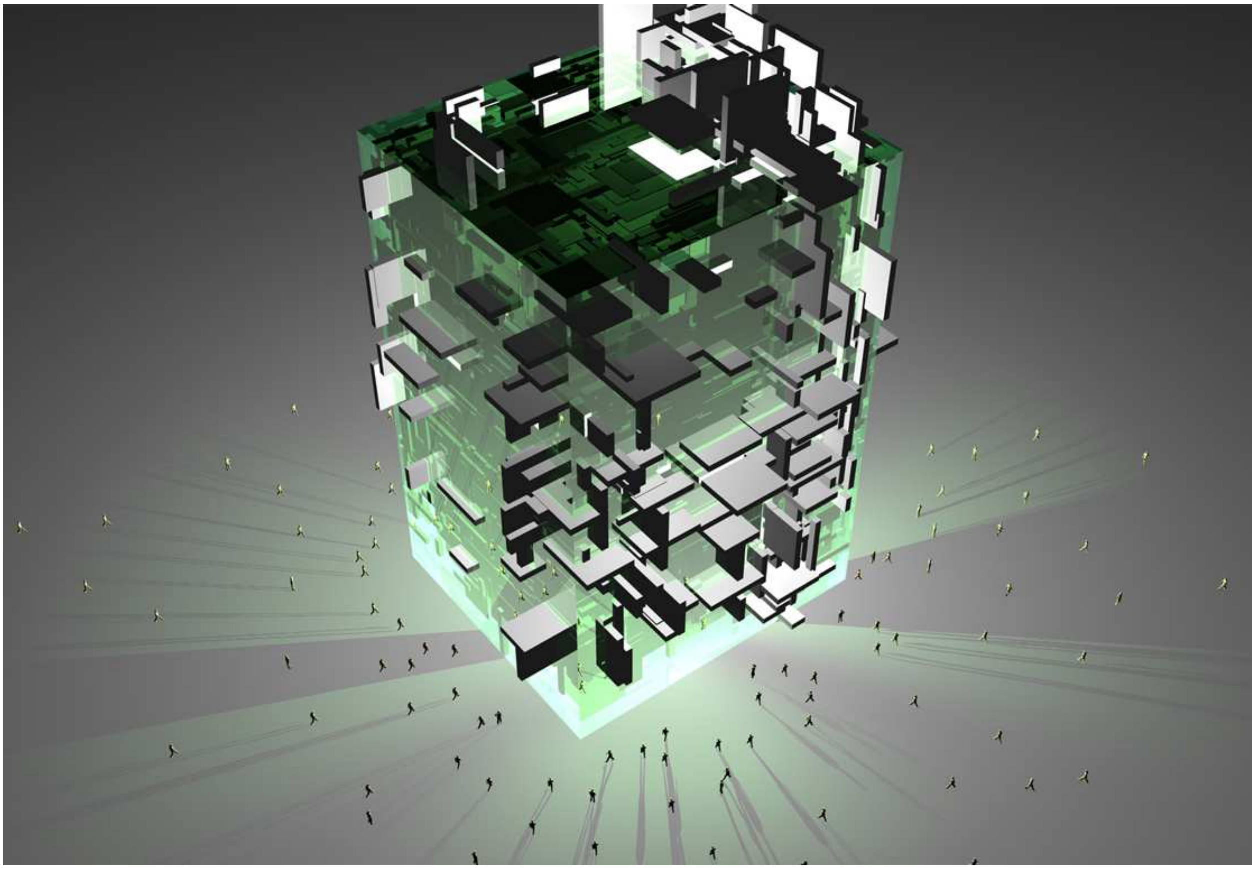
example applications:

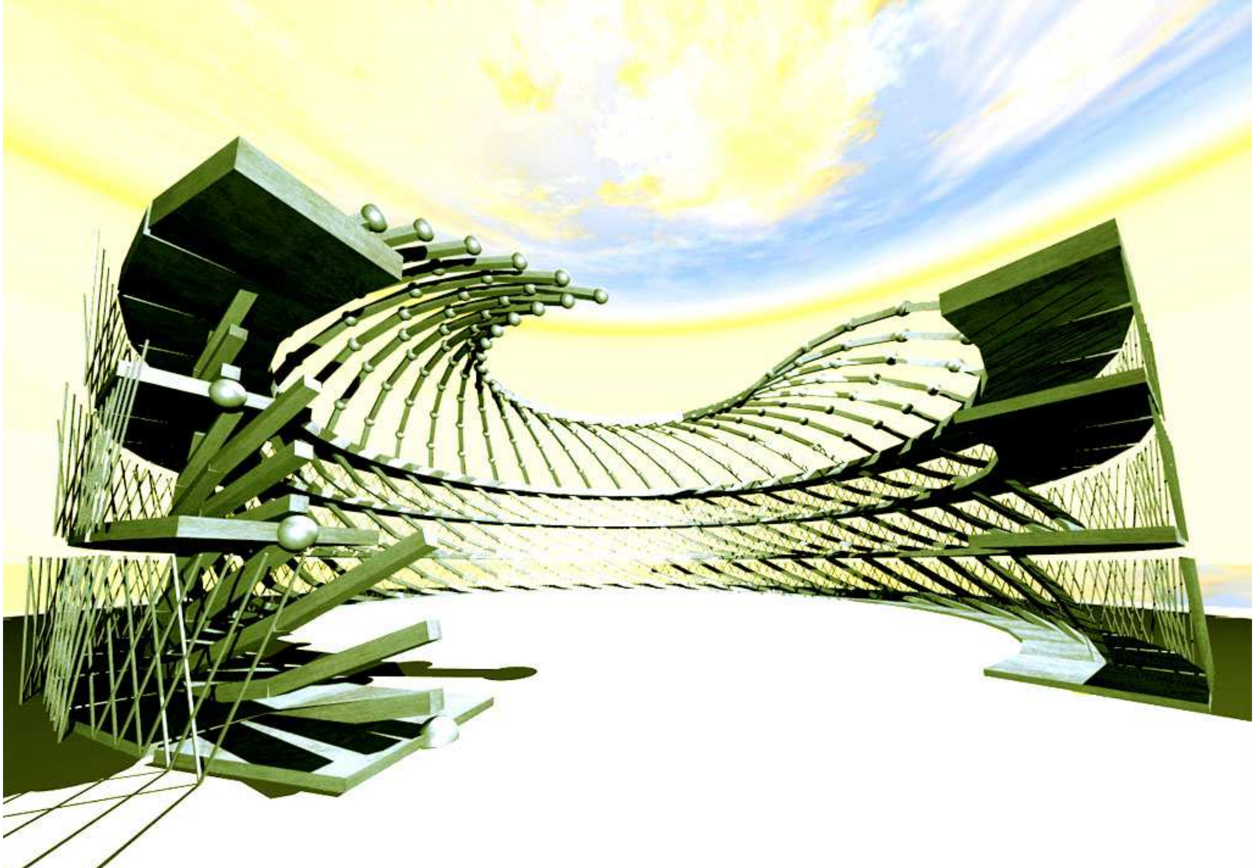
virtual barley
(Buck-Sorlin 2006)

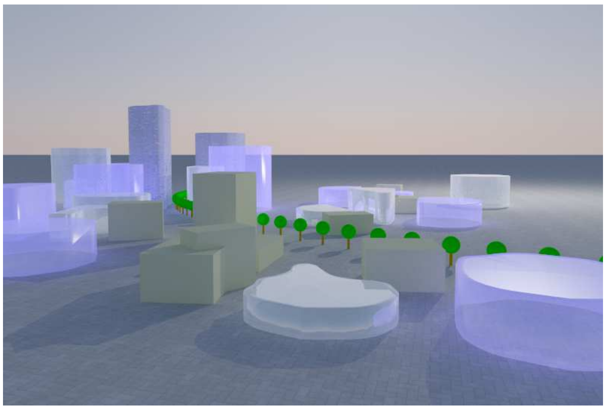
virtual Black Alder tree, generated with GroIMP,
in a VRML scene (for Branitz Park Foundation, Cottbus;
Rogge & Moschner 2007)



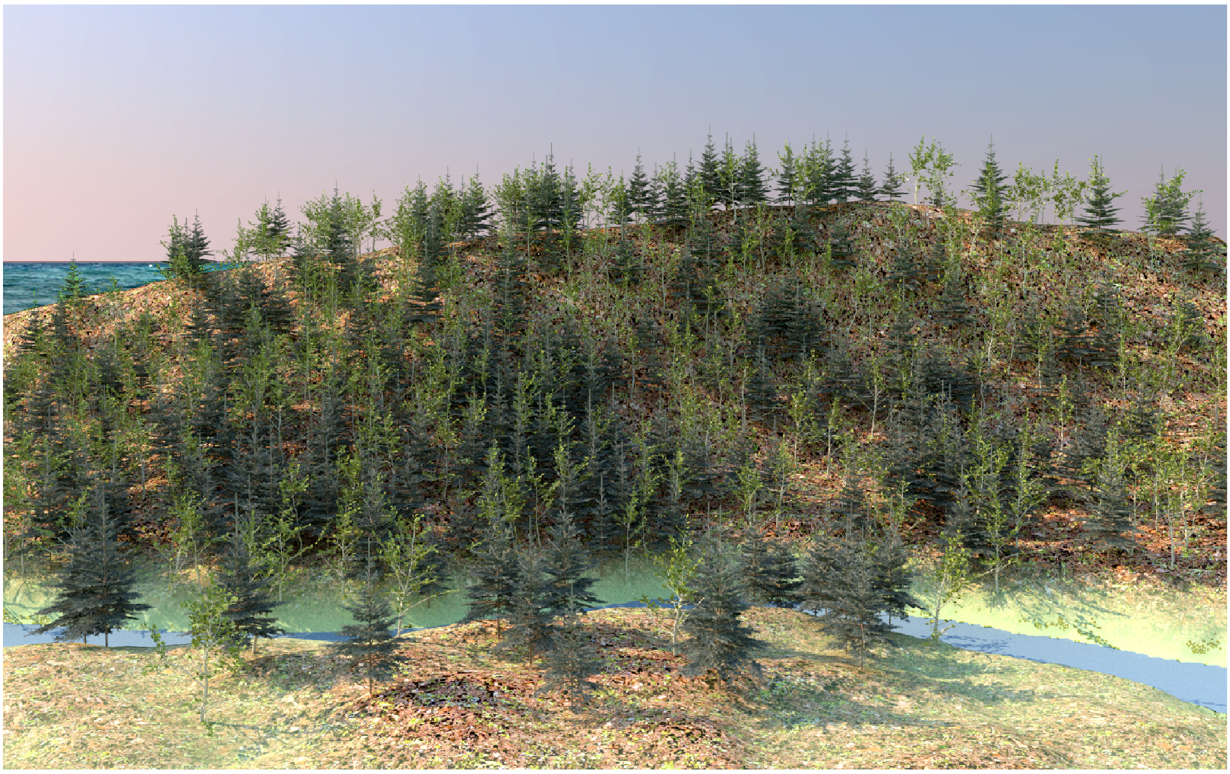
This and next images: students' results from
architecture seminar, BTU Cottbus 2007



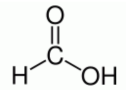




virtual landscape with beech-spruce mixed stand (Hemmerling et al. 2008)



Exercises

1. Write a turtle command sequence which generates a balls-and-sticks molecule model of formic acid (structural formula: ).
The atoms shall be represented by spheres with different sizes and colours (depending on the chemical element), and the bonds by cylinders (`F(...)` command of the turtle). The double bond shall be represented by a thicker cylinder.
Hint: The colour of an atom `X` can be specified as in the following example:

```
module X extends Sphere(1.0)
   {{ setShader(BLUE);}};
```

Test your solution with GroIMP.

2. (a) Write an L-system which simulates the primary growth of a plant in annual steps. The *annual shoots* of the vertical main axis (stem) shall all have the same length. The uppermost annual shoot shall bear an *apical bud* (= a red sphere) and a *lateral bud* (= a green sphere). The apical bud is supposed to produce a new annual shoot of the main axis next year, and from the lateral bud shall grow a shorter *lateral shoot* with a branching angle of 45°, which will terminate its growth next year (i.e., there are no buds at the lateral shoots). The positions of the lateral branches are alternating (left-right-left-right-...) along the stem. The simulation shall start with an apical bud.

(b) Modify the model by introducing a trend: Assume that the annual shoots get 10 per cent shorter each year.

(c) Assume additionally that the apical bud produces a flower (= a large blue cone) after 7 years, and that the plant then stops to grow.

Test your solutions with GroIMP.

Remarks: By **M(-*s*)** you can cause the turtle to move back along the main axis by stepsize *s*. **Cone(*h*, *r*)** stands for a cone with height *h* and radius *r*.

3. Open the example "Molecules" in GroIMP's built-in example portfolio ("File" / "Show Examples").
Make several model runs by clicking on the buttons "Run run" and "Reset", and observe what happens.
Now modify the model in the following ways:
(a) Increase the number of atoms from 10 to 20.
(b) Switch off the output of text to the console.
(c) Double the distance threshold for formation of a bond between two atoms.