Functional-structural modelling: basics of the Language XL

Replacement rules in L-systems

simple form

left-hand side ==> right-hand side; e.g. F0 ==> F0 F0; F0 will be replaced by two F0

complex form

(* context *) left-hand side, (condition) ==> right-hand side { imperative XL-code }

(* t:tree *), b:branch, (t[age] > 10) ==> b [RU(45) bud] [RU(-45) bud] {b[length] *= 1.1}

Elements of the syntax

Axiom	start object
F0	cylinder with variable length and diameter
F(10)	cylinder with length 10 and variable diameter
F(10, 1)	cylinder with length 10 and diameter 1
F(10, 1, 14)	cylinder with length 10, diameter 1 and color 14 (yellow) (meaning of color indices see below)
RU(10)	rotation by 10 degrees counterclockwise around up-axis
RL(10)	rotation by 10 degrees counterclockwise around left-axis
RH(10)	rotation by 10 degrees counterclockwise around head-axis
[	beginning of branching (start subgraph with branching relationship)
]	end of branching (end subgraph, return to point of origin)
M(3)	Movement in direction of head-axis without drawing an element. Negative values indicate a backwards movement.
Translate(0, 0, 3)	Movement without drawing an element in the global coordinate system, (x, y, z) = (0, 0, 3), in relation to the current position.
L(10) [MRel(0.7) …]	Symbol combination for a relative movement alongside the parent element. L(10) determines the length of following F0-commands. MRel(0.7) determines the relative position (0.7 of the current length, i.e.:7) alongside the parent element,

starting from its basis (at the bottom), where the next F0 is constructed.

Parameters

A(x) ==> A(x+1)	Calculating with parameter of the object of type A (which will be replaced)
a:A ==> a {a[carbonpool] += 2.5}	Calculating with attributes of the object a of type A, the object remains otherwise unchanged

Module definition

All used modules (object types) need to be defined. Basic modules predefined in GroIMP are F0, F(), M(), RU(), etc. Individual modules can be defined in the rgg-file itself. Using the module-definition a new object class is created. An object class can be seen as a blueprint for objects, where attributes and methods (which are special abilities and functions) are defined. Each created object is an instance of a class (a module type) and is set up following the specific blueprint. This process is called „construction“ or „instantiation“. The attributes are preset by the blueprint; the respective values are determined at construction but can be changed throughout the calculations. The class Humans has for example the attribute „height“, but the value is different for each instance of the class – i.e. for each human. Attributes, which need to be specified at construction, are called „call parameters“. Modules defined by yourself can be specified as followes:

module Shoot extends Null;

Simple, non-drawable object without call parameters.

module Shoot (int age) extends Null;

Simple, non-drawable object with one call parameter. Which means, that at initialization of an object of this type an integer has to be specified, e.g. Shoot(20).

module Shoot (int age) extends F(10);

Simple, drawable object based on the standard module F(). An object Shoot inherits all attributes and methods defined in the blueprint of the module F(). This is for example the attribute „length“. Call parameters for Shoot see above. The call parameter for F(10) is constantly set to be 10, i.e. all Shoot-objects will be created with an initial length of 10. The object attribute „length“ thus has the value 10. „age“ does not have anything to do with the length.

module Shoot (int age, float length) extends F(length);

The definition of the Shoot object is extended by another call parameter (float length). This parameter is passed on to F(length) at construction. Which means that the object will be created with a corresponding initial length. The object has the self-defined attribute „length“, as well as the inherited attribute „length“. Those are equal during the initialization (construction) of the object.

If one attribute is changed throughout the execution of the code, the other one remains unchanged. Only changing the inherited attribute „length“ results in a visible alteration of the object.

module Shoot (int age, super.length) extends F(length);

This definition of the Shoot object directly relates the second call parameter to the inherited attribute „length“ of the F-class. „super.“ is a reference to the blueprint of the class denoted by „extends“.

Addressing:

r.age or r[age]	addressing the attribute of the object r (with the second version each change automatically results in redrawing)
r.setTransform(0, 0, 1)	addressing the method setTransform(x,y,z) of the object r

Reminder:

r:Branch, (r[age] > 10) ==> … searching for all instances of the class Branch. r is an identifier (name) for the respective instance. It is only used for addressing (for example to frame the condition).

Rule types

There is more than one rule type.

==> **Rule in Lindenmayer-form**. Simple replacement rule, where one subgraph (often only one object) is replaced by another graph. The relationships are recovered. That means that if in a one-dimensional graph [A B C D] the node B is replaced by a node G, then G will be inserted into the graph with the original relationship of the node B: [A G C D]. The blank space always indicates a successor edge in XL.

Axiom ==> A B C D;
B ==> G;

==>> **Rule in SPO-form**. Complex replacement rule, where one subgraph is replaced by another graph. The relationships need to be recovered by the programmer. That means that if in a one-dimensional graph [A B C D] the node B is replaced by the node G, then the original relationships of B in the graph will not be maintained. If the relationship to C is not explicitly stated, then there will be no connection of the nodes C D to the graph. This entails that the nodes are no more visible, which means they are effectively deleted. Only the (unconnected) nodes A and G remain.

Axiom ==> A B C D;
B ==>> G;

In order to achieve the same result as in the example above (Lindenmayer-form), it is necessary to explicitly list the relationships to A and C on the right-hand side of the rule:

```
Axiom ==> A B C D;
a:A B c:C ==>> a G c;
```

This rule type is frequently used to delete subgraphs.

```
Axiom ==> A A [ B A A A ] A A [ C A A ] A

B ==>> ;
```

The resulting graph reads as follows „A A A A [C A A] A“.

::> **Update rule.** This rule type does not change the structure of the graph. It is used to change the attributes of the objects (nodes) of the graph.

```
c:C ::> {c[length] = c[length] * 20; }
```

Appendix:

Color table for the F-command:

0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	light grey
8	dark grey
9	light blue
10	light green
11	light cyan
12	light red
13	light magenta
14	yellow
15	white