Tutorial and workshop „Modelling with GroIMP and XL“ / Tutorial for beginners

University of Göttingen, 27 February, 2012

Winfried Kurth

Basic examples in XL (part 2)

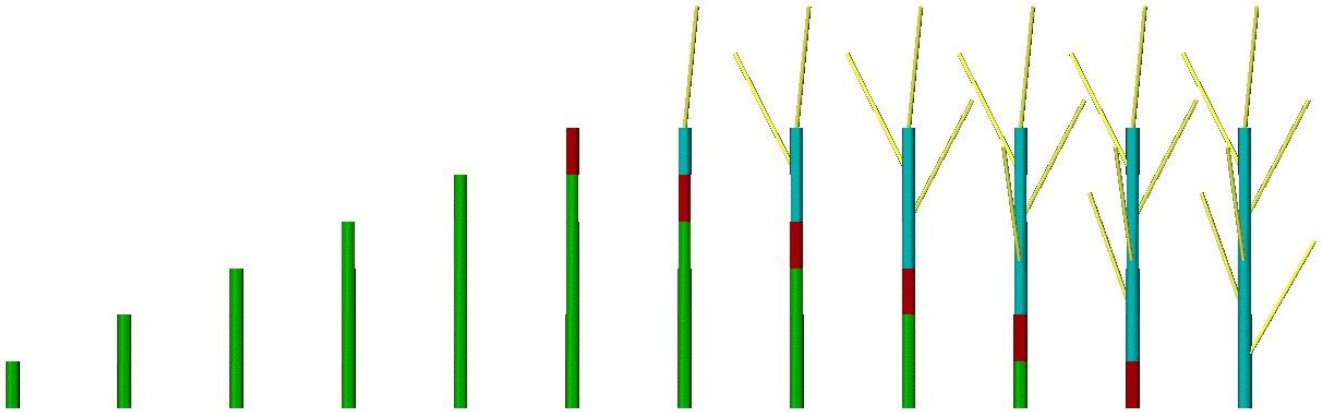
Context sensitivity

Query for a context which must be present to make a rule applicable

specification of the context in (* .... *)

example:

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



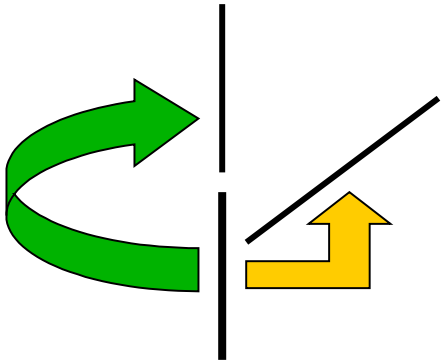
test the examples

`sm09_e14.rgg`	usage of a left context
`sm09_e15.rgg`	usage of a right context

The step towards graph grammars

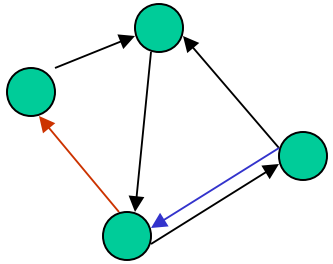
drawback of L-systems:

- 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**

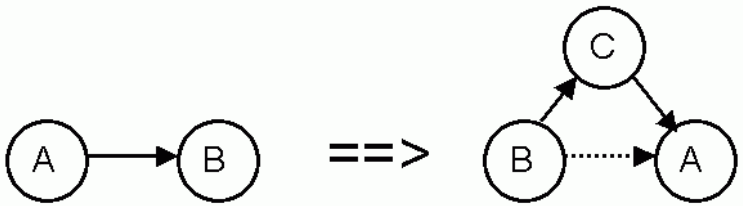
a string:
a very simple graph

- a string can be interpreted as a 1-dimensional graph with only one type of edges
- successor edges (successor relation)

to make graphs dynamic, i.e., to let them change over time:

graph grammars

example rule:



A relational growth grammar (RGG) (special type of Graph grammar) contains:

- an alphabet
  - the definition of all allowed
    - node types
    - edge types (types of relations)
- the Axiom
  - an initial graph, composed of elements of the alphabet
- a set of graph replacement rules.

How an RGG rule is applied

- 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.

a complete RGG rule can have 5 parts:

(* context *), left-hand side, (condition)

==>

right-hand side { imperative XL code }

in text form we write (user-defined) edges as

`-edgetype->`

edges of the special type "successor" are usually written as a blank (instead of **`-successor->`**)

also possible: `>`

2 types of rules for graph replacement in XL:

- L-system rule, symbol: ==>

provides an embedding of the right-hand side into the graph (i.e., incoming and outgoing edges are maintained)

- SPO rule, symbol: ==>>

incoming and outgoing edges are deleted (if their maintenance is not explicitely prescribed in the rule)

„SPO“ from „single pushout“ – a notion from universal algebra

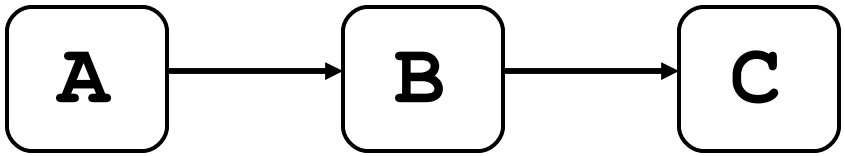
example:

```
a:A ==>> a C
B   ==>  D E
C   ==>  A
```

(SPO rule)

(L-system rules)

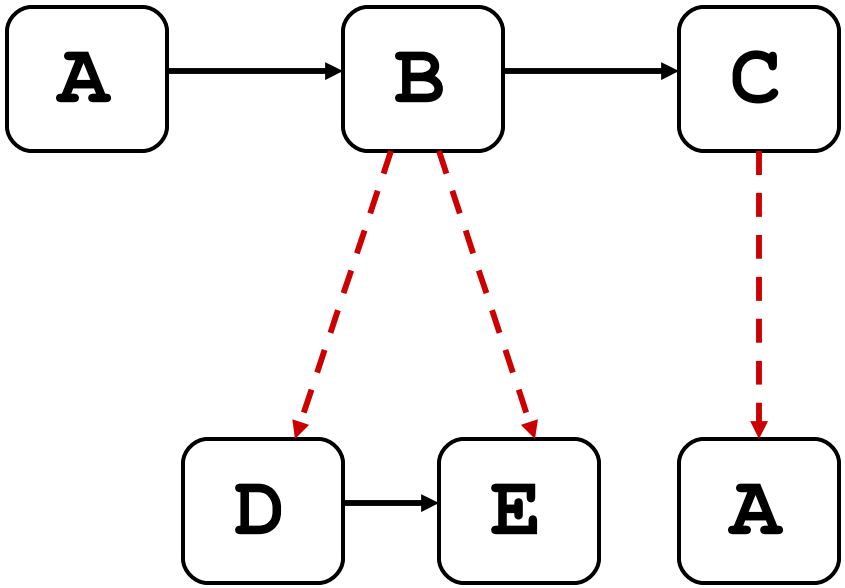
start graph:



```
a:A ==>> a C
B   ==>  D E
C   ==>  A
```

(SPO rule)

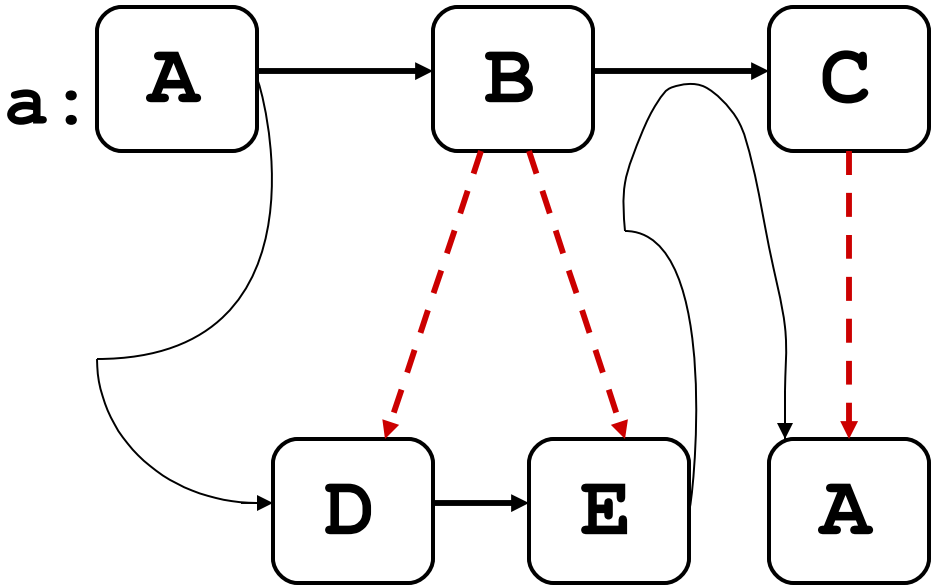
(L-system rules)



```
a:A ==>> a C
B   ==>  D E
C   ==>  A
```

(SPO rule)

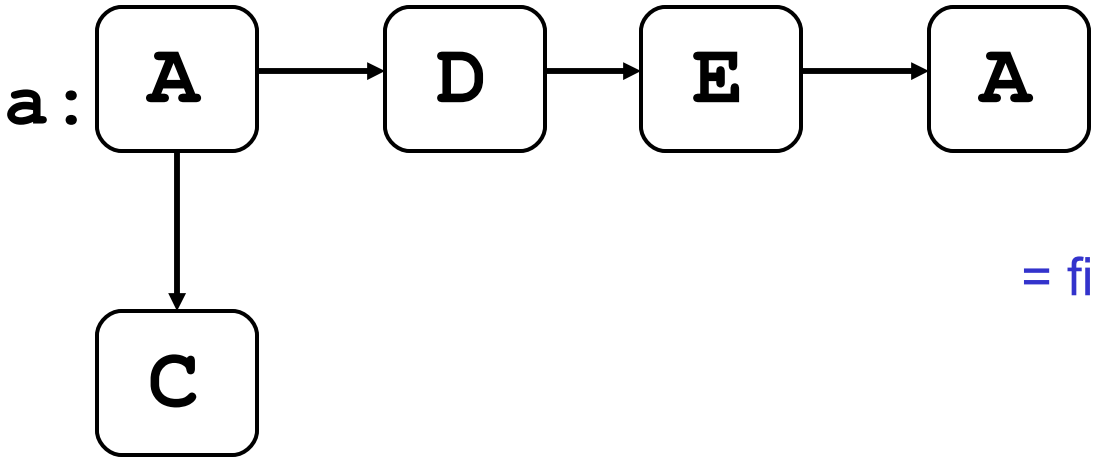
(L-system rules)



```
a:A ==>> a C
B   ==>  D E
C   ==>  A
```

(SPO rule)

(L-system rules)



= final result

test the example **sm09_e27.rgg** :

```
module A extends Sphere(3);

protected void init()
[  Axiom ==> F(20, 4) A; ]

public void runL()
[
   A ==> RU(20) F(20, 4) A;
]

public void runSPO()
[
   A ==>> ^ RU(20) F(20, 4, 5) A;
]
```

(^ denotes the root node in the current graph)

Mathematical background:

Chapter 4 of Ole Kniemeyer‘s doctoral dissertation (`http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:kobv:co1-opus-5937`).

The language XL

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

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

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

Features of the language XL:

- nodes of the graph are Java objects (including geometry objects)

example: XL programme for the Koch curve (see part 1)

```
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“)

special nodes:

geometry objects

Box, Sphere, Cylinder, Cone, Frustum, Parallelogram...

access to attributes by parameter list:

`Box(x, y, z)` (length, width, height)

or with special functions:

`Box(...).(setColor(0x007700))` (colour)

special nodes:

geometry objects

Box, Sphere, Cylinder, Cone, Frustum, Parallelogram...

transformation nodes

`Translate(x, y, z)`, `Scale(cx, cy, cz)`, `Scale(c)`,

`Rotate(a, b, c)`, `RU(a)`, `RL(a)`, `RH(a)`, `RV(c)`, `RG`, ...

light sources

PointLight, DirectionalLight, SpotLight, AmbientLight

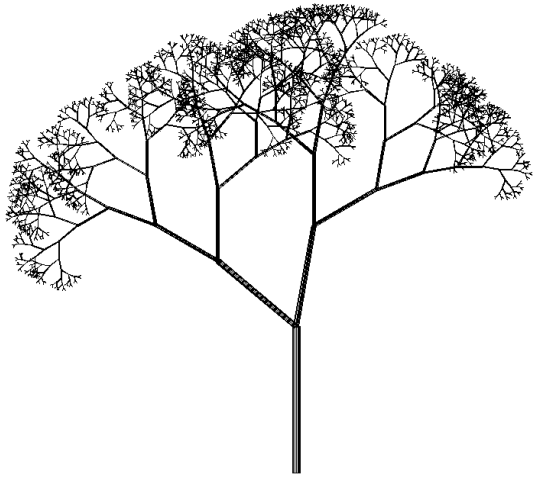
Features of the language XL:

- rules organized in blocks [...], control of application by control structures

example: rules for the stochastic tree

```
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 ] );
```



Features of the language XL:

- parallel application of the rules

  (can be modified: sequential mode can be switched on, see below)

Features of the language XL:

- parallel execution of assignments possible

special assignment operator := besides the normal =

quasi-parallel assignment to the variables x and y:

```
x := f(x, y);
y := g(x, y);
```

Features of the language XL:

- operator overloading (e.g., „+“ for vectors as for numbers)

  (see later presentation by Reinhard Hemmerling)

Features of the language XL:

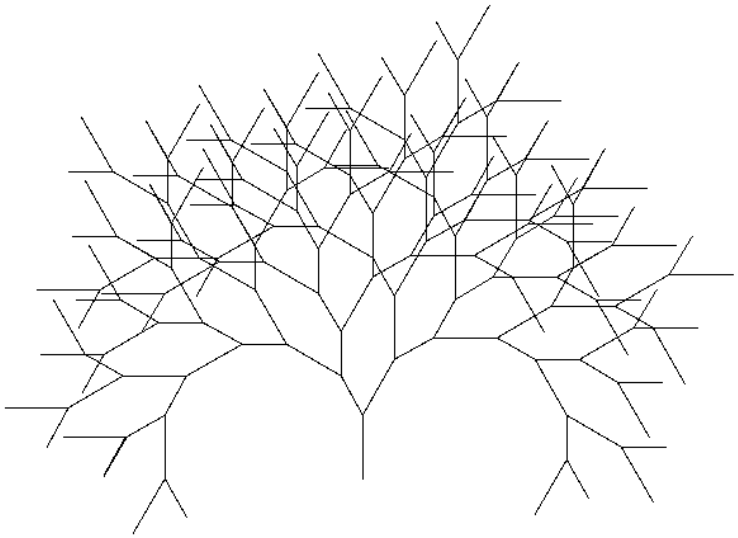
- set-valued expressions (more precisely: producer instead of sets)
- **graph queries** to analyze the actual structure

example for a graph query:

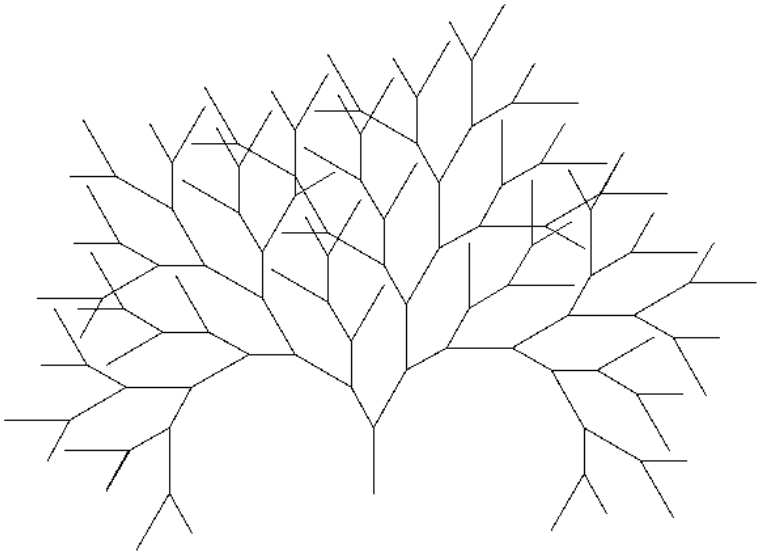
binary tree, growth shall start only if there is enough distance to other **F** objects

```
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



query syntax:

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.

Queries as generalized contexts:

test the examples **`sm09_e28.rgg`**, **`sm09_e29.rgg`**, **`sm09_e30.rgg`**

Features of the language XL:

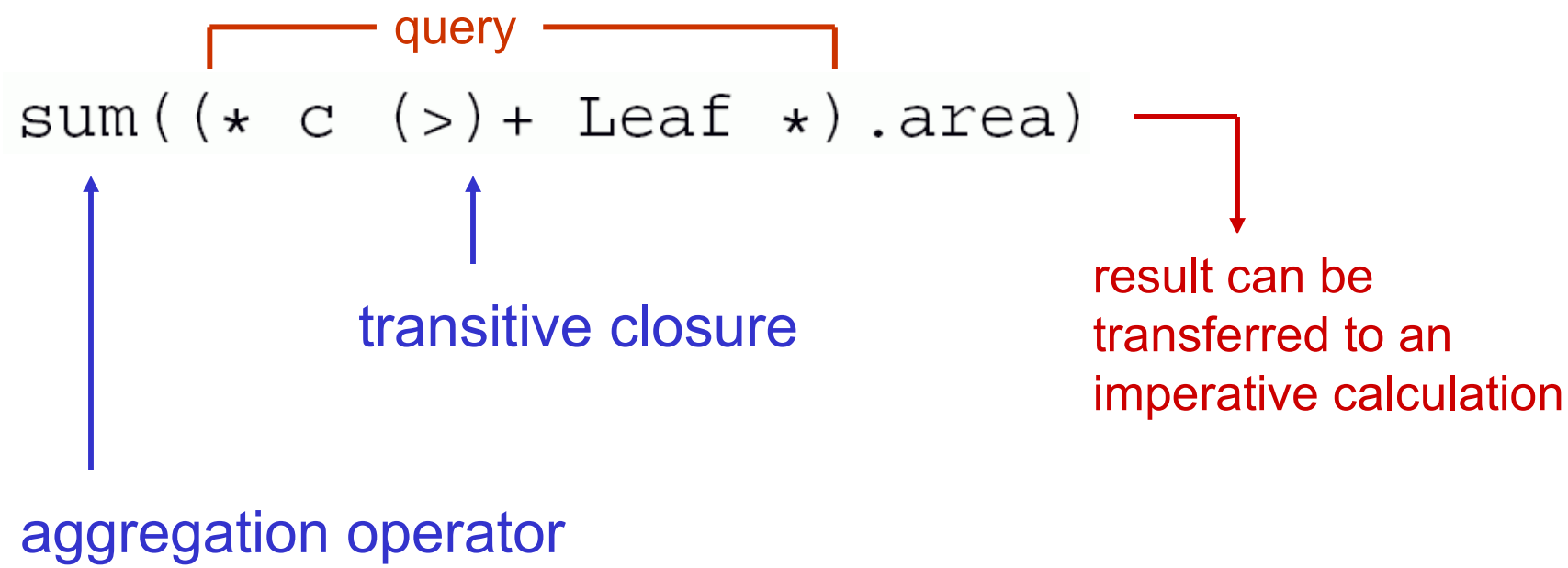
- aggregating operators (e.g., „sum“, „mean“, „empty“, „forall“, „selectWhereMin“)

can be applied to set-valued results of a query

Queries and aggregating operators

provide possibilities to connect structure and function

example: search for all leaves which are successors of node `c` and sum up their surface areas



Queries in XL

test the examples

`sm09_e31.rgg`,
`sm09_e35.rgg`,
`sm09_e36.rgg`

for light interception / photosynthesis:

a simple model of overshadowing

using a query referring to a geometric region in space

model approach (strongly simplifying):
overshadowing of an object occurs when there are further objects in an imagined cone with its apex in the object, opened into z direction (to the sky).

example:

`sm09_e42.rgg` competition of three 2-dimensional model plants for light

```
module Segment(int t, int ord) extends F0;
module TBud(int t) extends F(1, 1, 1);
module LBud extends F(0.5, 0.5, 1);

Vector3d z = new Vector3d(0, 0, 1);

protected void init()
   [
   Axiom ==> P(2) D(5) V(-0.15) [ TBud(-4) ] RU(90) M(600) RU(-90)
                                [ TBud(0) ] RU(-90) M(1200) RU(90)
                                [ TBud(-8) ];
   ]

public void run()
   [
   TBud(t), (t < 0) ==> TBud(t+1);
   x:TBud(t), (t >= 0 && empty( (* s:Segment, (s in cone(x, z, 45)) *) ) ) ==>
      L(random(80, 120)) Segment(0, 0)
      [ MRel(random(0.5, 0.9)) RU(60) LBud ]
      [ MRel(random(0.5, 0.9)) RU(-60) LBud ] TBud(t+1);
   y:LBud,
      (empty( (* s:Segment, (s in cone(y, z, 45)) *) ) ) ==>
      L(random(60, 90) Segment(0, 1) RV0 LBud;
   Segment(t, o), (t < 8) ==> Segment(t+1, o);
   Segment(t, o), (t >= 8 && o == 1) ==>> ;   /* removal of the whole branch */
   ]
```

Representation of graphs in XL

- node types must be declared with „**`module`**“
- nodes can be all Java objects.
  In user-made **`module`** declarations, methods (functions) and additional variables can be introduced, like in Java
- notation for nodes in a graph:
  `Node_type`, optionally preceded by: **`label:`**
  Examples: **`A`**, **`Meristem(t)`**, **`b:Bud`**
- notation for edges in a graph:

  `-`*edgetype*`->`, `<-`*edgetype*`-`
- special edge types:
  successor edge: **`-successor->`**, **`>`** or *(blank)*
  branch edge: **`-branch->`**, **`+>`** or **`[`**
  refinement edge: **`/>`**

Notations for special edge types

`>`	successor edge forward
`<`	successor edge backward
`---`	successor edge forward or backward
`+>`	branch edge forward
`<+`	branch edge backward
`-+-`	branch edge forward or backward
`/>`	refinement edge forward
`</`	refinement edge backward
`-->`	arbitrary edge forward
`<--`	arbitrary edge backward
`--`	arbitrary edge forward or backward

(cf. Kniemeyer 2008, p. 150 and 403)

user-defined edge types

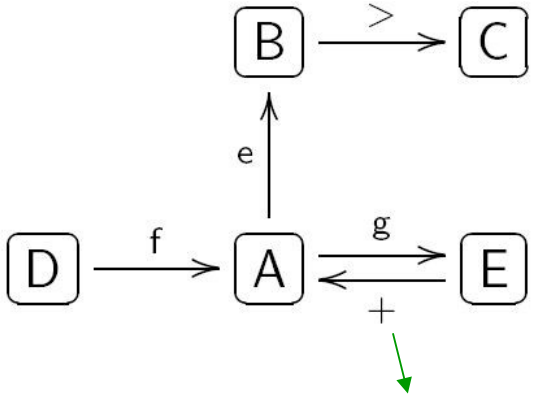
`const int xxx = EDGE_0;` **`//`** oder **`EDGE_1`**, ..., **`EDGE_14`**

...

usage in the graph: **`-xxx->`**, **`<-xxx-`**, **`-xxx-`**

Notation of graphs in XL

example:



(>: successor edge, +: branch edge)

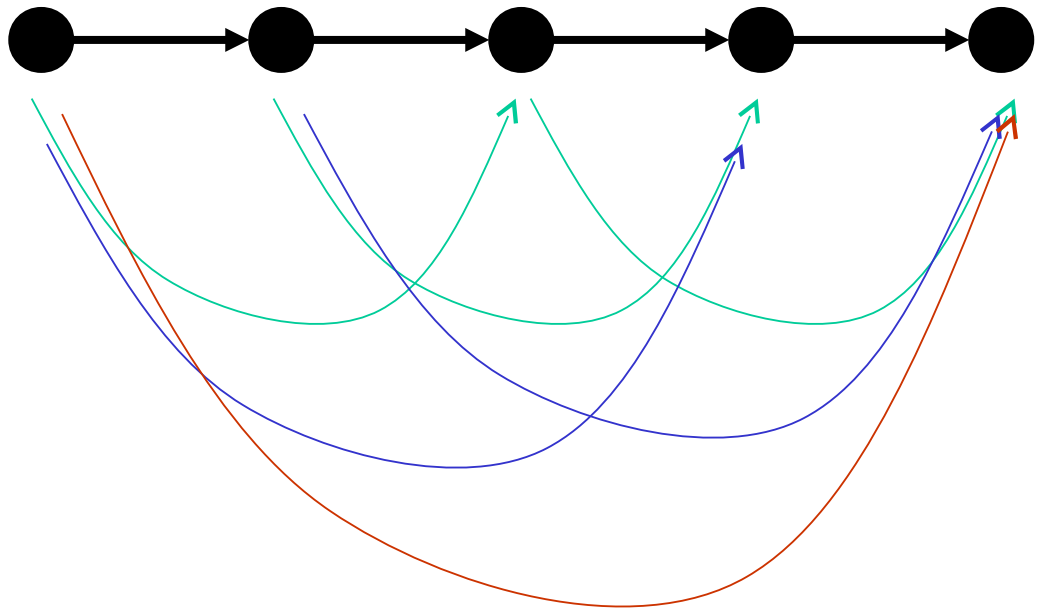
is represented in programme code as

```
a:A [-e-> B C] [<-f- D] -g-> E [a]
```

(the representation is not unique!)

derived relations

relation between nodes connected by several edges (one after the other) of the same type:



„transitive hull“ of the original relation (edge)

Notation for the transitive hull in XL:

`(-edgetype->)+`

reflexive-transitive hull („node stands in relation to itself“ also permitted):

`(-edgetype->)*`

e.g., for the successor relation: **`(>)*`**

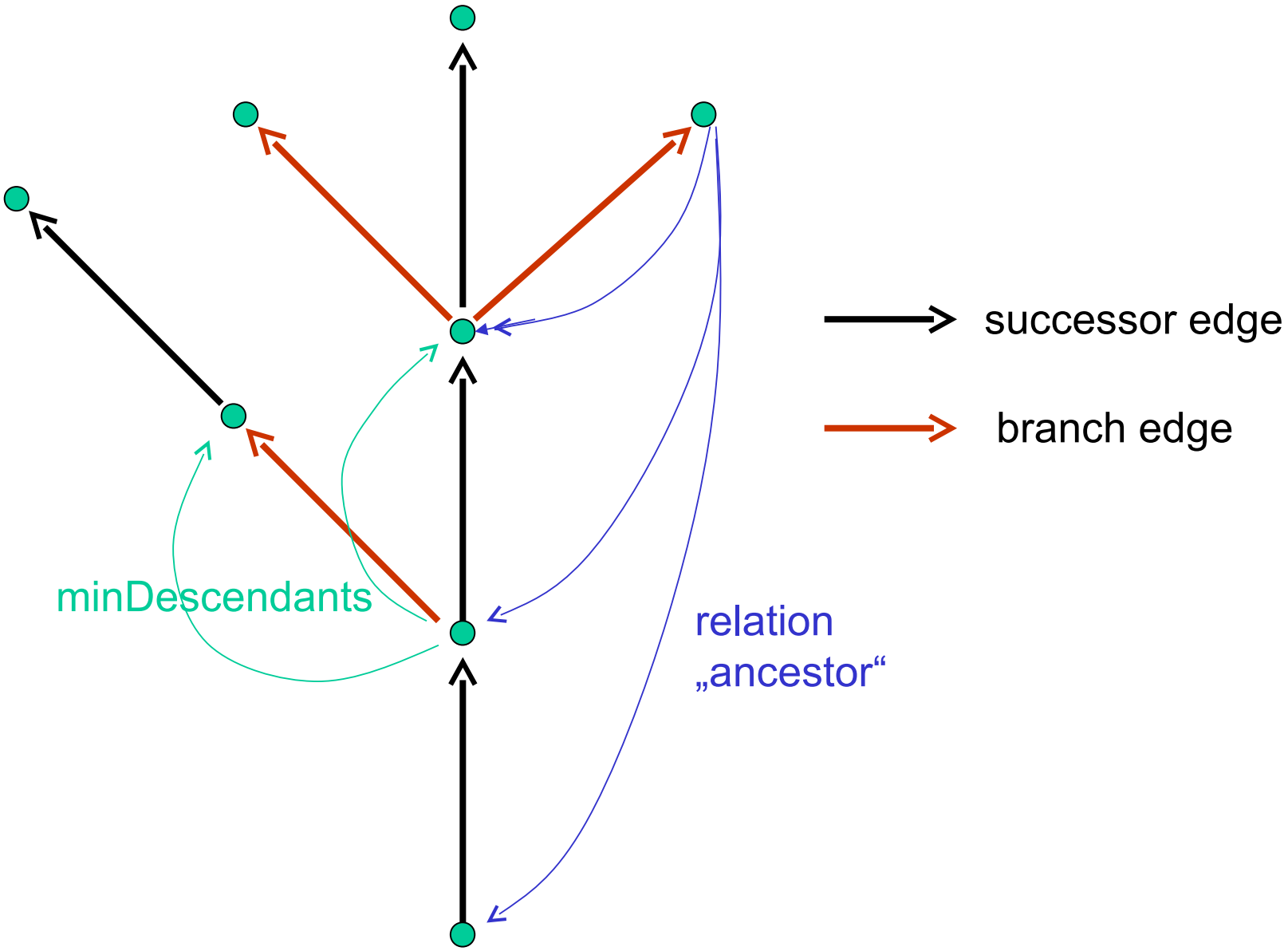
common transitive hull of the special relations „successor“ and „branch“, in reverse direction:

`-ancestor->`

interpretation: this relation is valid to all „preceding nodes“ in a tree along the path to the root.

nearest successors of a certain node type:

`-minDescendants->` (nodes of other types are skipped)


successor edge
branch edge
minDescendants
relation
„ancestor“

The current graph

GroIMP maintains always a graph which contains the complete current structural information. This graph is transformed by application of the rules.

Attention: Not all nodes are visible objects in the 3-D view of the structure!

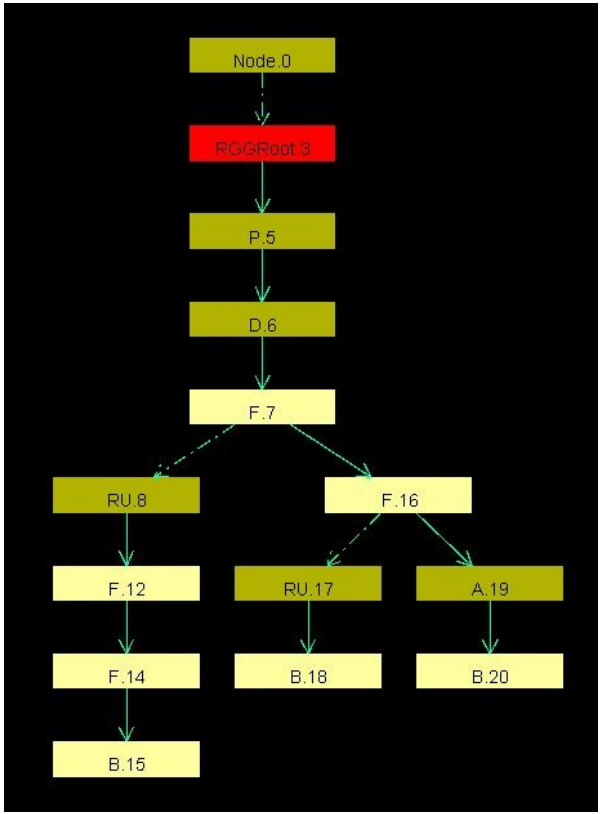
- **`F0`**, **`F(x)`**, **`Box`**, **`Sphere`**: yes
- **`RU(30)`**, **`A`**, **`B`**: normally not (if not derived by „**`extends`**“ from visible objects)

The graph can be completely visualized in the **2-D graph view** (in GroIMP: Panels - 2D - Graph).

Load an example RGG file in GroIMP and execute some steps (do not work with a too complex structure).

Open the 2-D graph view, fix the window with the mouse in the GroIMP user interface and test different layouts (Layout - Edit).

Keep track of the changes of the graph when you apply the rules (click on „redraw“)!



which parts of the current graph of GroIMP are visible (in the 3-d view) ?

all geometry nodes which can be accessed from the root (denoted ^) of the graph by exactly one path, which consists only of "successor" and "branch" edges

How to enforce that an object is visible in any case:

```
==>> ^ Object
```

a further type of rules:

actualization rules

often, nothing at the graph structure has to be changed, but only attributes of one single node are to be modified (e.g., calculation of photosynthesis in one leaf).

For this purpose, there is an extra rule type:

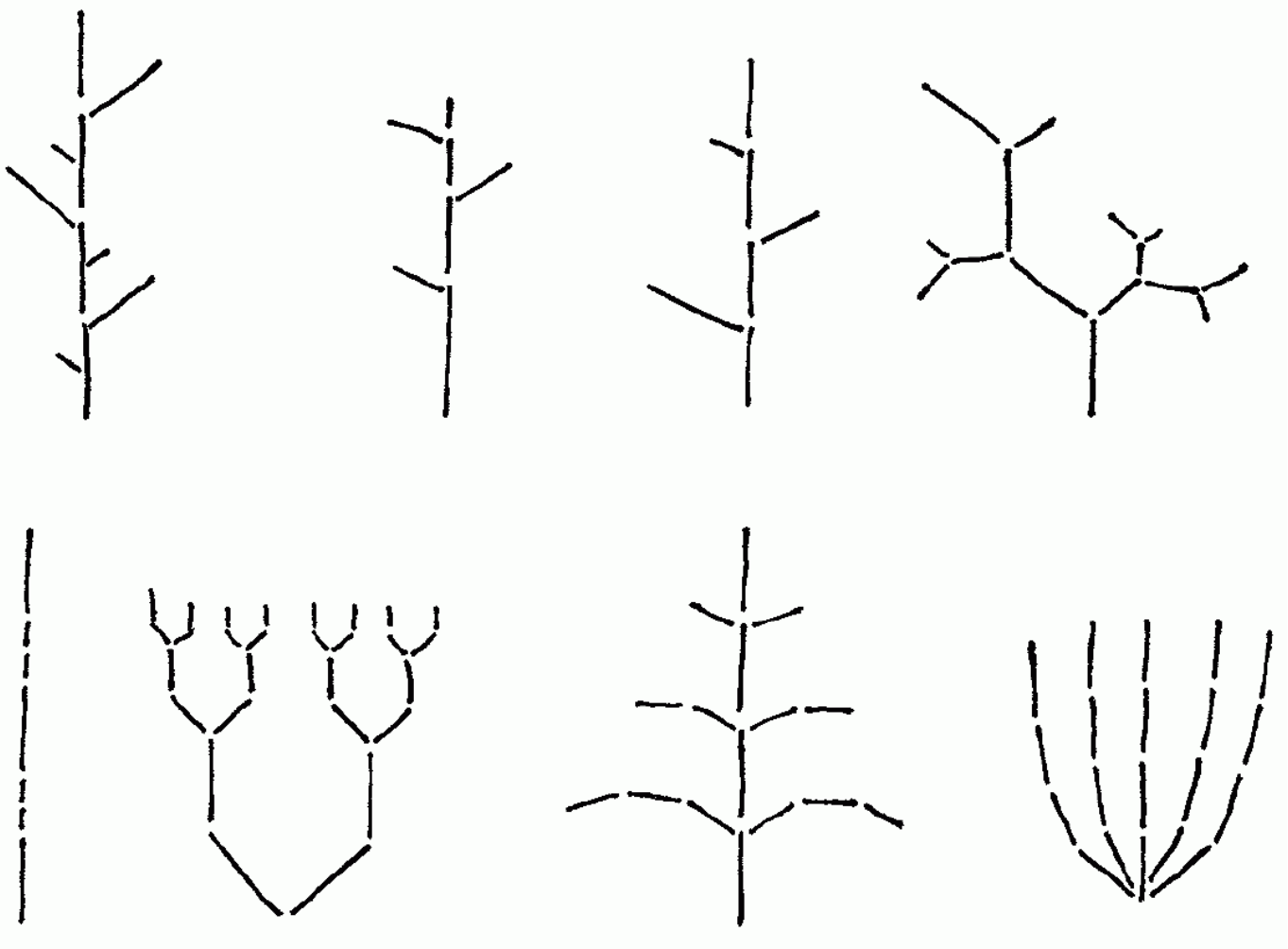
`A :::> {` *imperative code* **`};`**

Test the examples **`sm09_e25.rgg`**, **`sm09_e16.rgg`**, **`sm09_e17.gsz`**, **`sm09_e18.rgg`**

and concerning the access to node attributes: **`sm09_e26.rgg`**

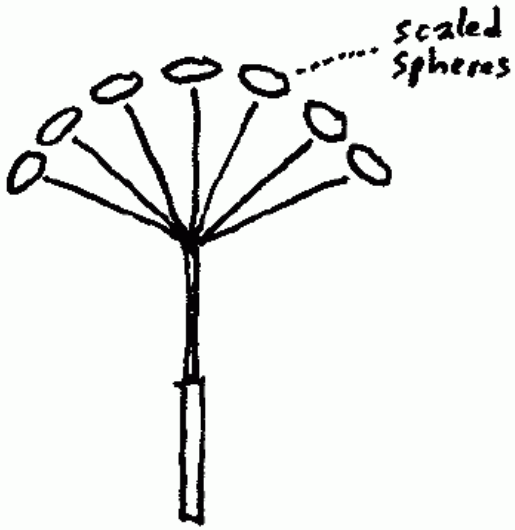
Suggestions for team session:

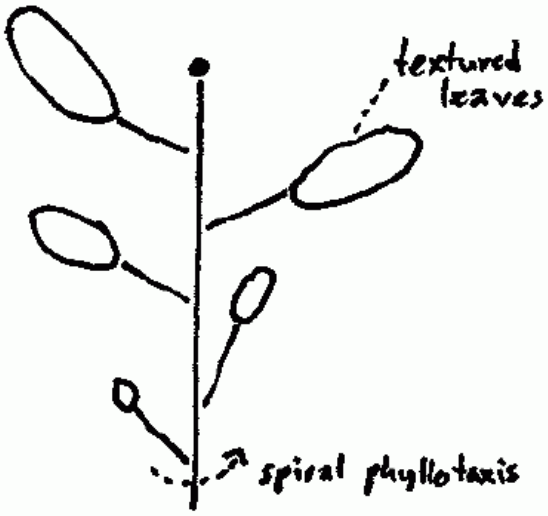
Create the following simple branching patterns with XL

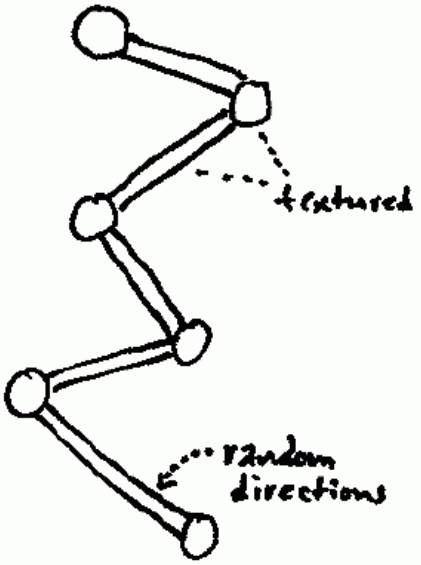


Suggestions for team session:

Create the following patterns as textured structures with XL







make queries in the generated structures