Full Documentation of GAMA 1.5.1

Provided by the GAMA development team

http://code.google.com/p/gama-platform/

Table of Contents

Introduction to GAMA 1.5.1 ... **14**
What's new in GAMA 1.5.1 ... 14

Interface Guide ... **15**
Interface Guide ... **16**
Introduction ... 16
GAMA perspectives ... 16
List of GAMA perspectives ... 16
Modeling/Simulation switch perspectives ... 18
Project and model creation ... 20
project creation ... 20
model creation ... 21
Experiment running ... 21
Inspectors ... **22**
Introduction ... 22
Species inspector ... 22
Agent inspector ... 22

Modeling Guide ... **24**
Introduction to the Modeling Guide ... **25**
Introduction ... 25
GAML Language ... 25

Model sections overview ... **26**
Model Sections ... **27**
include ... 27
global ... 27
entities ... 27
environment ... 28
experiment ... 28

Species section ... **30**
Species Definition ... **31**
Introduction ... 31
Species declaration ... 31
Skills: behavioral plug-ins ... 32
Aspects: display properties ... 33
Parent: inheritance of species ... 34

Scheduling description. . . . 34
Topology description. . . . 35
Nesting species. . . . 35
Control: behavioral architecture. . . . 35
Base: Java foundation. . . . 36
Variables definition. . . . 37
Declaration. . . . 37
Basis. . . . 37
init or <-. . . . 37
const. . . . 38
update. . . . 38
Special attributes. . . . 38
function. . . . 38
parameter. . . . 39
max, min. . . . 39
of. . . . 39
Naming variables. . . . 40
Reserved Keywords. . . . 40
Naming conventions. . . . 40
Accessing variables. . . . 40
Direct access. . . . 40
Remote access. . . . 41
Built-in Variables. . . . 43
Introduction. . . . 43
List of built-in agent variables. . . . 43
Behaviors. . . . 44
Choice of an agent behavior architecture. . . . 44
Common behavior structure. . . . 44
reflex. . . . 44
Attributes. . . . 44
Definition. . . . 44
init. . . . 45
Attributes. . . . 45
Definition. . . . 45
Task behavior models. . . . 45
task. . . . 45
Sub elements. . . . 45
Definition. . . . 45
FSM-based behaviors. . . . 45
state. . . . 46
Attributes. . . . 46
Sub elements. . . . 46
Definition. . . . 46
Actions. . . . 47
Definition of an action. . . . 47
Use of an action. . . . 47
Skills. . . . 48

Overview. 48
moving. 48
variables. 48
speed. 48
heading. 49
destination. 49
actions. 49
follow. 49
goto. 49
move. 49
wander. 49
Experiment section. 51
Experiment definition. 52
Experiment definition. 52
Experiment types. 52
GUI. 53
Definition of GUI experiment. 53
Input. 53
Parameters. 53
User command. 53
Output. 54
Display. 54
agents layer. 54
species layer. 55
image layer. 56
chart layer. 56
text layer. 57
File. 57
Monitor. 58
Batch. 59
Definition. 59
The batch experiment facets. 59
Parameter definition. 59
The method element. 60
Batch Methods. 60
Exhaustive exploration of the parameter space. 60
Hill Climbing. 61
Simulated Annealing. 61
Tabu Search. 62
Reactive Tabu Search. 62
Genetic Algorithm. 63
GAML language. 65
Data Types. 66
Table of Contents. 66
Primitive built-in types. 66
bool. 66

float. 66
int. 66
string. 67
Complex built-in types. 67
agent. 67
container. 67
file. 68
geometry. 68
graph. 69
list. 70
map. 70
matrix. 71
pair. 71
path. 71
point. 72
rgb. 72
species. 73
Species names as types. 73
topology. 74
Defining custom types. 74
Statements. 76
Table of Contents. 76
General syntax. 76
add. 76
Attributes. 76
Definition. 76
ask. 77
Attributes. 77
Definition. 77
Notice. 78
capture. 78
Attributes. 78
Definition. 79
create. 79
Attributes. 79
Definition. 79
do. 81
Attributes. 81
Enclosed tags. 81
Definition. 81
error. 82
Attributes. 82
Definition. 82
if. 82
Attributes. 82
Following tags. 82
Definition. 82

let. 83
Attributes. 83
Definition. 83
loop. 83
Attributes. 83
Definition. 84
put. 85
Attributes. 85
Definition. 85
release. 85
Attributes. 85
Definition. 86
remove. 86
Attributes. 86
Definition. 86
return. 87
Attributes. 87
Definition. 87
save. 88
Attributes. 88
Definition. 88
set. 88
Attributes. 88
Definition. 88
switch. 89
Attributes. 89
Embedded tags. 89
Definition. 89
write. 89
Attributes. 89
Definition. 90
Operators. 91
Table of Contents. 91
Definition. 91
Operators by categories. 91
Casting operators. 91
Comparison operators. 92
Containers-related operators. 92
Files-related operators. 92
Graphs-related operators. 92
Logical operators. 92
Mathematics operators. 92
Matrix-related operators. 92
Random operators. 92
Spatial operators. 93
Statistical operators. 93
Strings-related operators. 93

System ... 93
Operators ... 93
- ... 93
: ... 94
:: ... 95
! ... 95
!= ... 95
? ... 95
/ ... 96
. ... 96
^ ... 96
`*` ... 97
`**` ... 97
+ ... 97
< ... 99
<--: ... 99
<= ... 99
<> ... 100
= ... 100
> ... 100
>= ... 101
abs ... 101
accumulate ... 102
acos ... 102
add_edge ... 102
add_point ... 102
add_z ... 103
add_z_pt ... 103
agent ... 103
agent_closest_to ... 104
agent_from_geometry ... 104
agents_at_distance ... 104
agents_inside ... 104
agents_overlapping ... 105
among ... 105
and ... 105
any ... 105
any_location_in ... 106
any_point_in ... 106
around ... 106
as ... 106
as_4_grid ... 106
as_date ... 107
as_distance_graph ... 107
as_edge_graph ... 107
as_grid ... 108
as_int ... 108

as_intersection_graph. 108
as_map. 109
as_matrix. 109
as_time. 109
asin. 109
at. 110
at_distance. 110
at_location. 111
atan. 111
binomial. 111
bool. 111
buffer. 112
ceil. 112
circle. 112
clean. 113
closest_points_with. 113
closest_to. 113
collate. 113
collect. 114
column_at. 114
columns_list. 114
cone. 115
container. 115
contains. 115
contains_all. 116
contains_any. 116
contains_edge. 116
contains_vertex. 117
convex_hull. 117
copy. 117
copy_between. 117
corR. 118
cos. 118
count. 118
crosses. 118
dead. 119
degree_of. 119
directed. 119
direction_between. 119
direction_to. 120
disjoint_from. 120
distance_between. 120
distance_to. 120
div. 121
empty. 121
enlarged_by. 121
eval_gaml. 121

evaluate_with. 122
even. 122
every. 122
exp. 122
fact. 123
farthest_point_to. 123
file. 123
first. 123
first_with. 124
flip. 124
float. 125
floor. 125
folder. 125
frequency_of. 126
gauss. 126
generate_barabasi_albert. 126
generate_watts_strogatz. 127
geometric_mean. 127
geometry. 127
get. 128
graph. 128
grid_at. 128
group_by. 129
harmonic_mean. 129
image. 129
in. 130
in_degree_of. 130
in_edges_of. 130
index_of. 130
inside. 131
int. 131
inter. 132
intersection. 132
intersects. 132
is. 133
is_image. 133
is_number. 133
is_properties. 133
is_shape. 134
is_text. 134
last. 134
last_index_of. 135
last_with. 135
length. 136
line. 136
link. 136
list. 137

ln. 137
load_graph_from_dgs. 137
load_graph_from_dgs_old. 138
load_graph_from_dot. 138
load_graph_from_edge. 138
load_graph_from_gexf. 139
load_graph_from_graphml. 139
load_graph_from_lgl. 140
load_graph_from_ncol. 140
load_graph_from_pajek. 140
load_graph_from_tlp. 141
map. 141
masked_by. 142
matrix. 142
max. 142
max_of. 143
mean. 143
mean_deviation. 144
meanR. 144
median. 144
min. 144
min_of. 145
mod. 145
mul. 145
neighbours_at. 146
neighbours_of. 146
new_folder. 147
norm. 147
not. 147
of. 147
of_generic_species. 147
of_species. 148
one_of. 148
or. 149
out_degree_of. 149
out_edges_of. 149
overlapping. 149
overlaps. 150
pair. 150
partially_overlaps. 151
path. 151
path_between. 152
path_to. 152
point. 152
points_at. 153
poisson. 153
polygon. 153

polyline. 153
predecessors_of. 154
product. 154
properties. 154
R_compute. 154
read. 154
rectangle. 155
reduced_by. 155
remove_duplicates. 155
remove_node_from. 155
reverse. 156
rewire_n. 156
rgb. 156
rnd. 157
rotated_by. 157
round. 158
row_at. 158
rows_list. 158
scaled_by. 158
select. 158
set_verbose. 159
shapefile. 159
shuffle. 159
simple_clustering_by_distance. 159
simple_clustering_by_envelope_distance. 160
simplification. 160
sin. 160
skeletonize. 161
solid. 161
sort. 161
sort_by. 161
source_of. 161
species. 162
species_of. 162
split_at. 162
split_lines. 162
split_with. 163
sqrt. 163
square. 163
standard_deviation. 163
string. 164
successors_of. 164
sum. 164
tan. 165
tanh. 165
target_of. 165
text. 166

TGauss. . . . 166
to_gaml. . . . 166
to_java. . . . 167
tokenize. . . . 167
topology. . . . 167
touches. . . . 167
towards. . . . 168
transformed_by. . . . 168
translated_by. . . . 168
translated_to. . . . 169
triangle. . . . 169
triangulate. . . . 169
truncated_gauss. . . . 169
undirected. . . . 170
union. . . . 170
unknown. . . . 170
user_input. . . . 171
variance. . . . 171
weight_of. . . . 171
where. . . . 172
with_max_of. . . . 172
with_min_of. . . . 172
with_optimizer_type. . . . 173
with_precision. . . . 173
with_weights. . . . 173
without_holes. . . . 173
write. . . . 174
Keywords. . . . 175
constants. . . . 175
nil. . . . 175
true, false. . . . 175
global built-in variables. . . . 175
time. . . . 175
step. . . . 175
seed. . . . 175
agents. . . . 176
pseudo-variables. . . . 176
self. . . . 176
myself. . . . 176
each. . . . 177
units. . . . 177
length. . . . 177
time. . . . 177
mass. . . . 178
surface. . . . 178
volume. . . . 178
Built-in Agents. . . . 179

Introduction. 179
cluster_builder. 179
actions. 179
simple_clustering_by_distance. 179
clustering_cobweb. 179
clustering_DBScan. 180
clustering_em. 180
clustering_farthestFirst. 181
clustering_simple_kmeans. 181
clustering_xmeans. 181
multicriteria_analyzer. 182
actions. 182
weighted_means_DM. 182
promethee_DM. 183
electre_DM. 183
evidence_theory_DM. 184
Built-in Actions. 185
Built-in actions. 185
debug. 185
tell. 185
Global built-in actions. 185
halt. 185
pause. 185
Additional features. 187
Gama 3D. 188
Introduction. 188
How to use OpenGL display in Gama. 188
Gama 3D Simulation perspective. 189
Mouse interaction. 189
Toogle button. 189
Information. 190
Why using 3D in your model?. 190
Multi layer. 190
Digital Model Elevation. 191
Data into form. 191
User Control. 193
Introduction. 193
user_command. 193
user_location. 194
`user_input` operator. 194
user control architecture. 194
user_only, user_first, user_last. 194
user_panel. 195
user_controlled. 198
Database access. 199
Description. 199
Introduction. 199

SQLSkill. 199
Define a species that uses the SQLSKILL skill. 199
Map of connection parameters. 200
Action: testConnection [params:: connection_parameter]. 200
Action: select [params:: connection_parameter, select::select_string]. 201
Action: executeUpdate [params:: connection_parameter, updateComm:: update_string]. 201
AgentDB. 202
Map of connection parameters. 202
Action: testConnection [params:: connection_parameter]. 203
Action: connect [params:: connection_parameter]. 203
Action: isConnected [ ]. 204
Action: close [ ]. 204
Action: getParameter [ ]. 204
Action: select [select:: select_string]. 205
Action: executeUpdate [updateComm:: update_string]. 205
Using SQL features to define environment or create species. 206
Define the boundary of the environment from database. 206
Create agents from the result of a `select` action. 207
Driving Skill. 208
Description. 208
Introduction. 208
driving skill. 208
Define a species that uses the driving skill. 208
variables. 208
living_space. 208
lanes_attribute. 208
tolerance. 209
obstacle_species. 209
action. 209
goto. 209
Model examples. 209

Introduction to GAMA 1.5.1

What's new in GAMA 1.5.1

GAMA version 1.5.1 is the GAMA version that will be used for the Can Tho Tutorial 2012 and the Fall Gama Coding Camp 2012. The version 1.5.1 improves some features of the 1.5 :

- correction of bugs (in particular, no more freezes when reloading an experiment)
- performance improvement (in particular for "big" models)
- improvement of the 3D integration (new operators to add a "z" to geometries, bug corrections...)
- new models (driving_traffic, Vote, 3D models)

—

Interface Guide

Interface Guide

Introduction

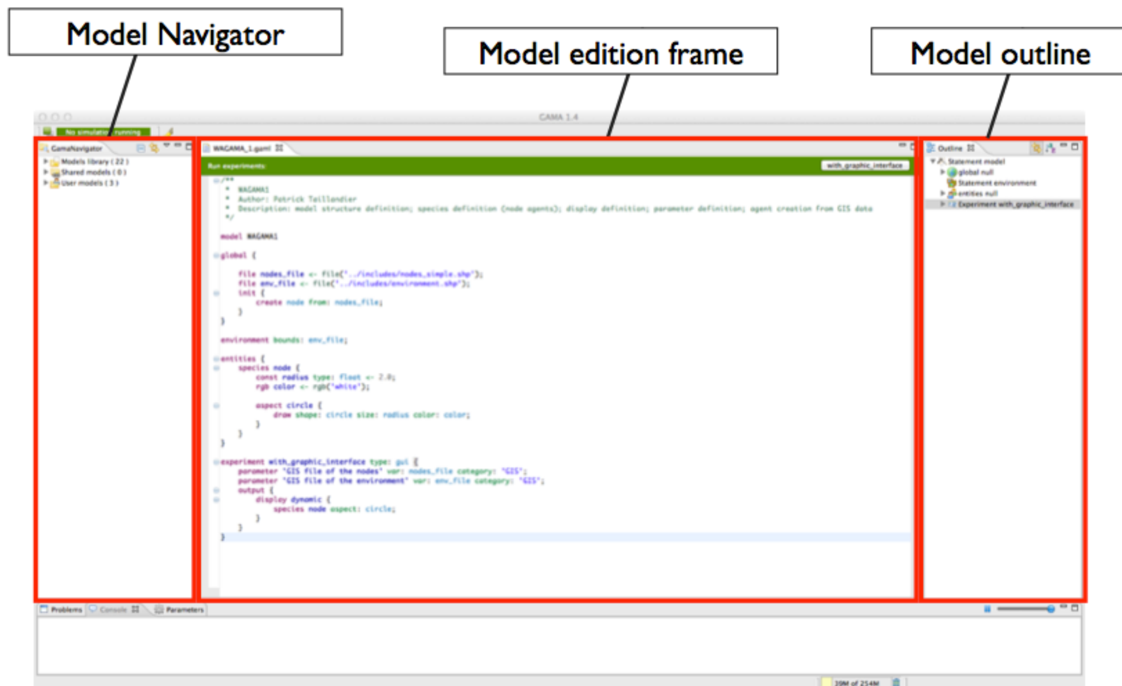
Developed as a plugin of Eclipse, GAMA has an graphical user interface very close to the Eclipse one. It is in particular based on the notions of editor, view and perspective. **Views** and **editors** and tabs in which user can display information (in the case of view) and edit files (in the case of editors). For example, the *Gama Navigator* allows to display the projects libraries. A **Perspective** a visual container in which are organized a set of views and editors.

GAMA perspectives

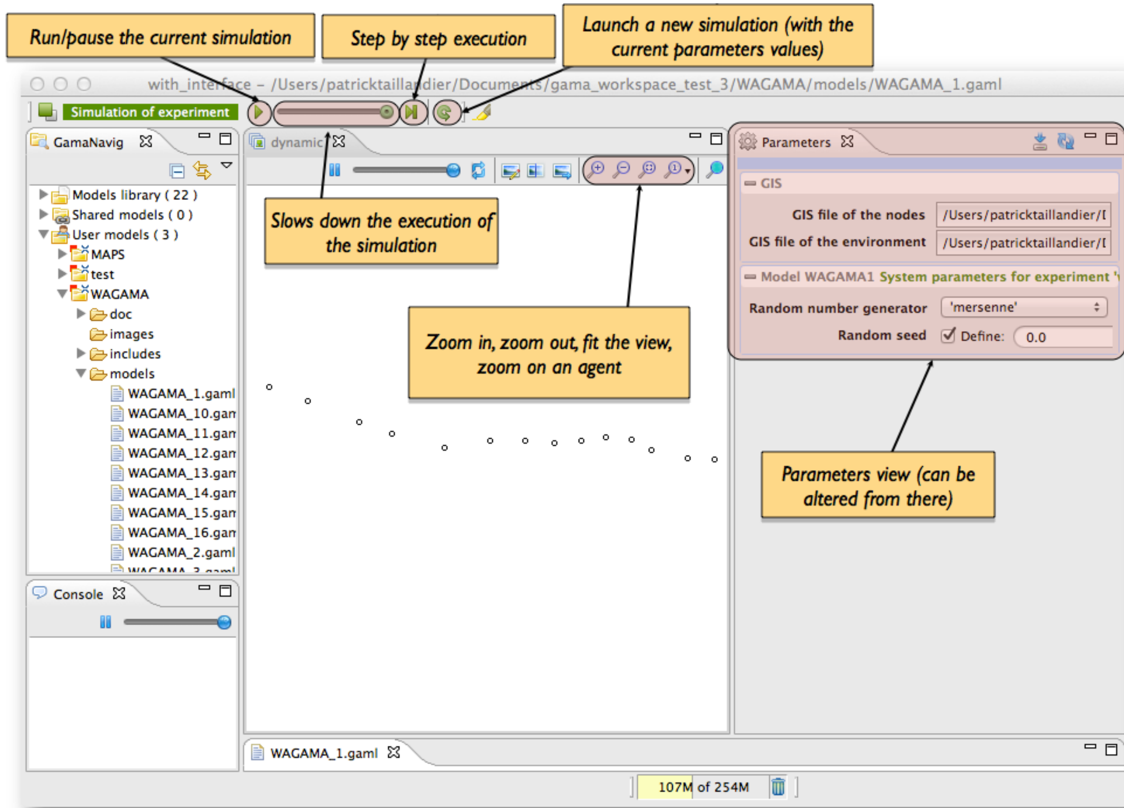
List of GAMA perspectives

GAMA provides five distinct perspectives that we describe in details in dedicated sections:

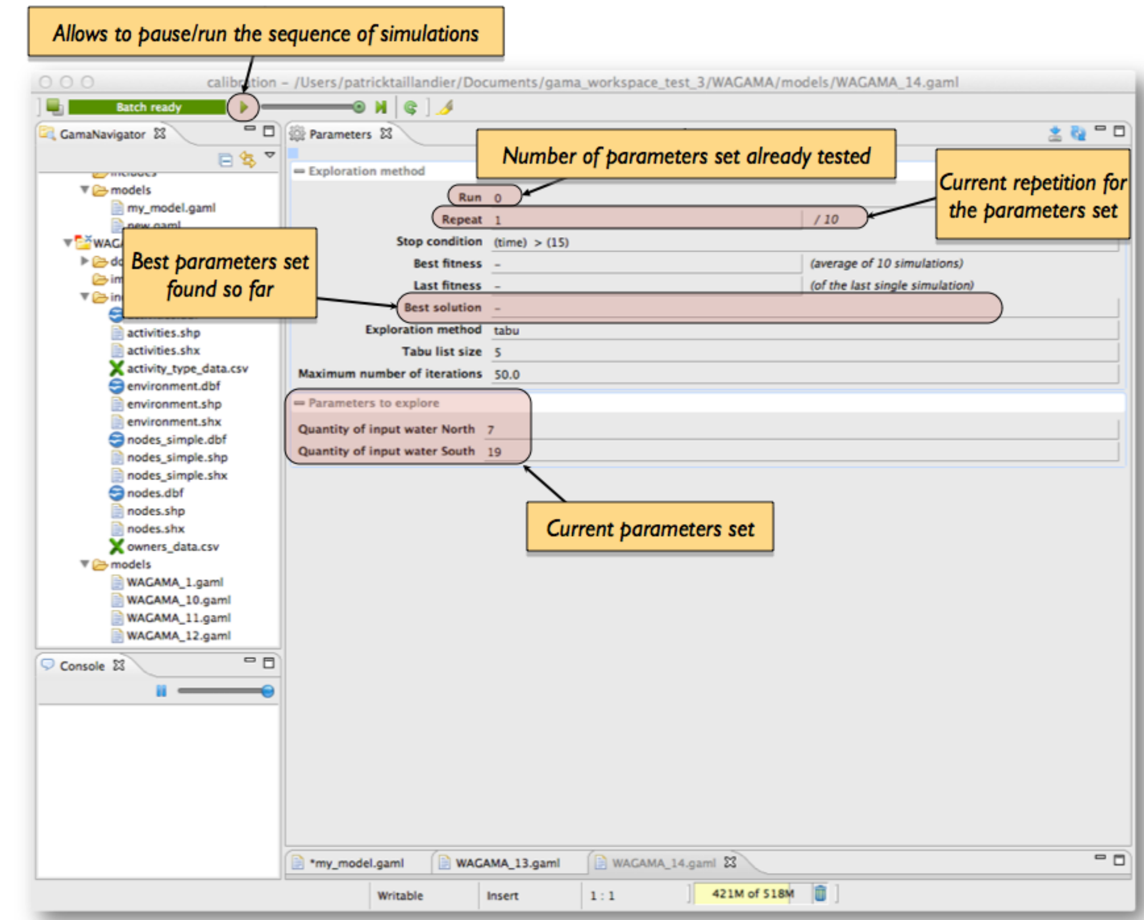
- the **Modeling** perspective (the default perspective);



- the **Simulation** perspective



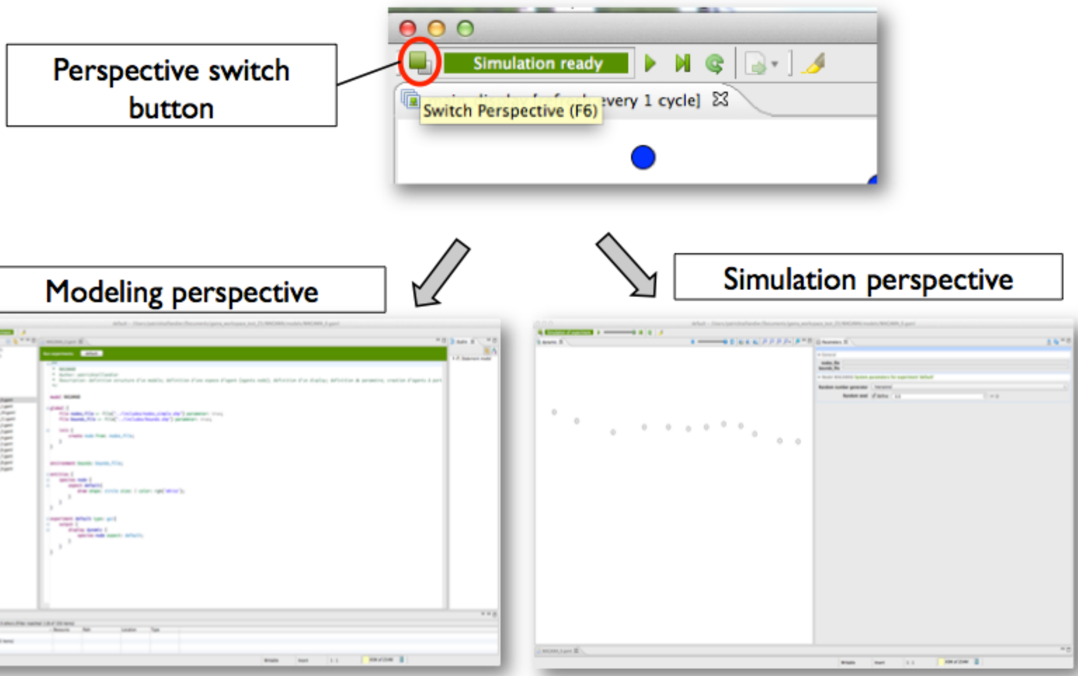
- the **Batch** perspective



- the **SVN Repository Exploring** perspective
- the **Team Synchronizing** perspective

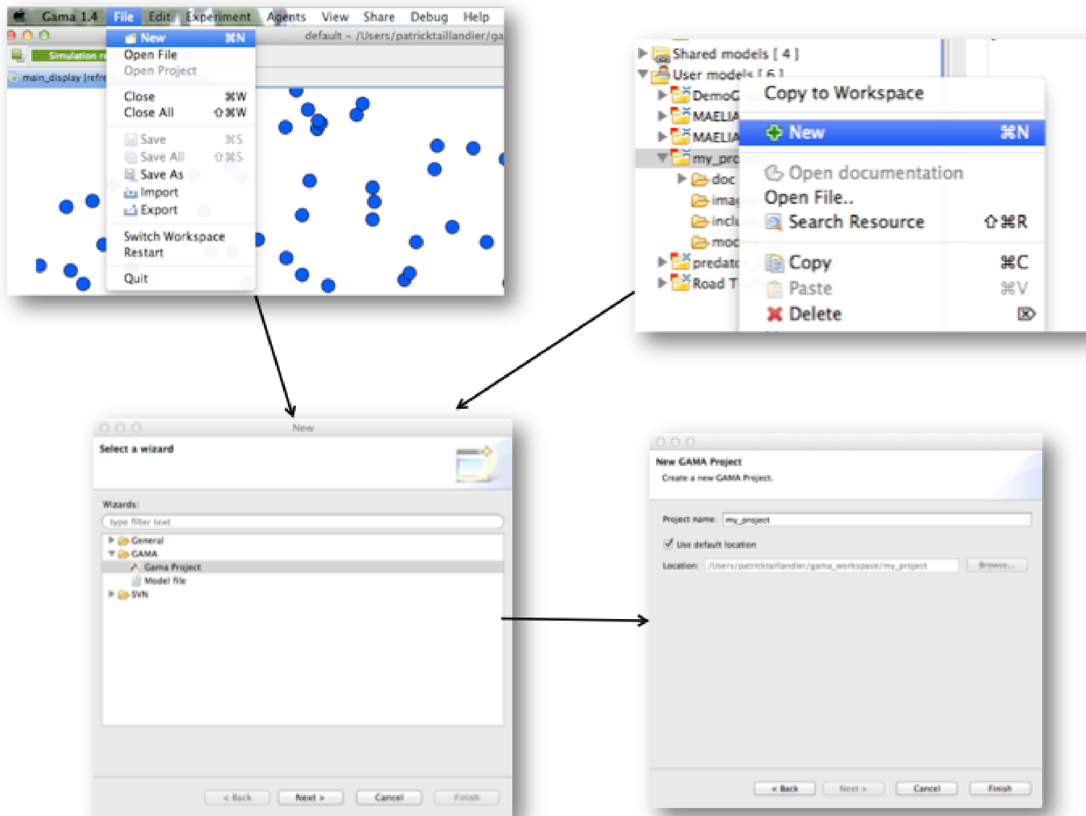
Modeling/Simulation switch perspectives

GAMA offers the possibility to simply switch between the modeling and the simulation perspective.

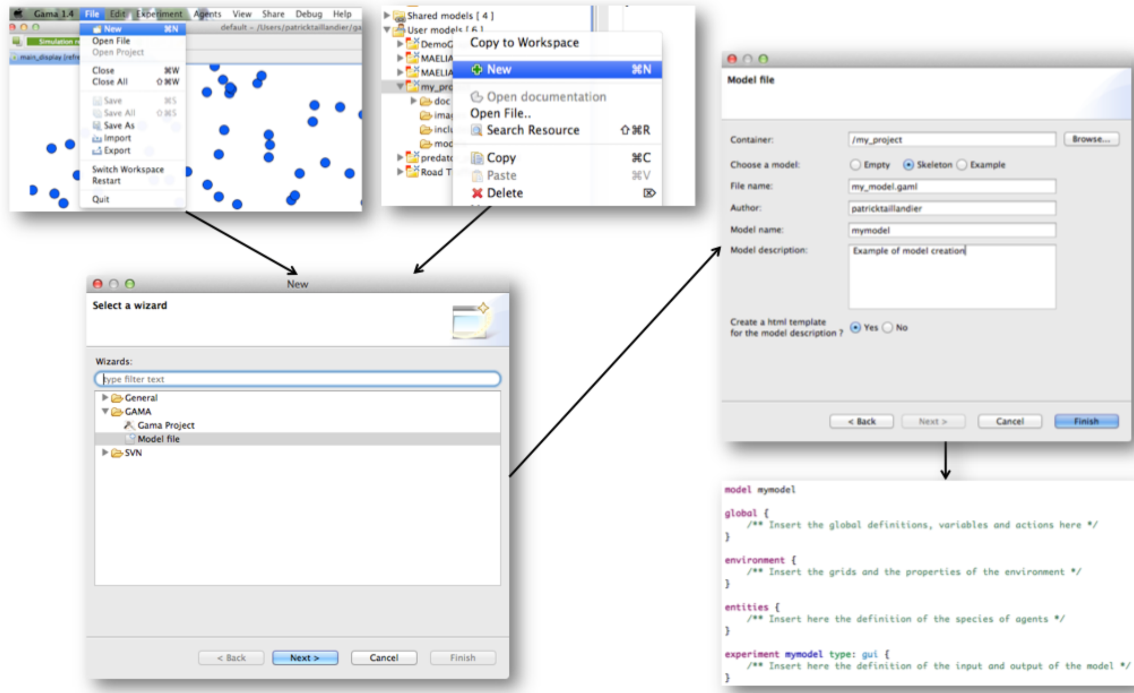

Perspective switch button
Simulation ready
Switch Perspective (F6)
Modeling perspective
Simulation perspective

Project and model creation

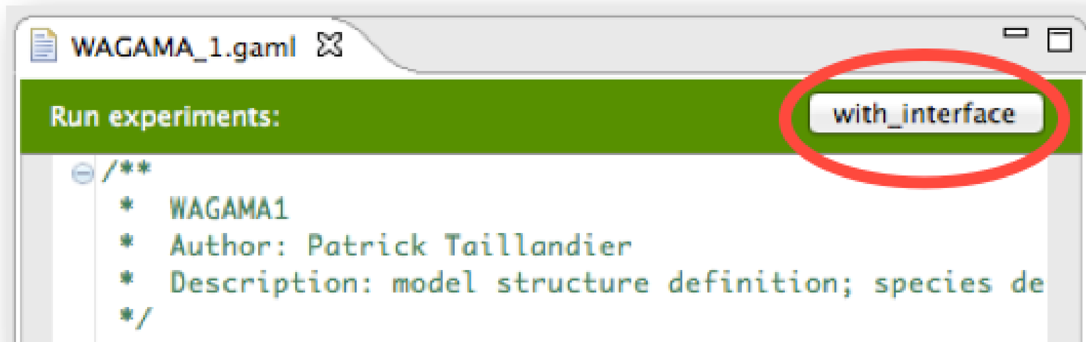
project creation



model creation



Experiment running



Inspectors

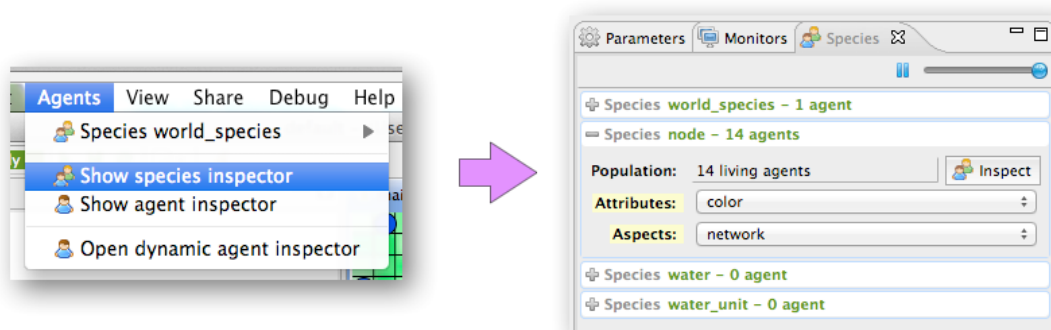
Introduction

The Inspectors allow to obtain informations about a species or an agent. There are two kinds of inspectors :

- species inspector
- agent inspector

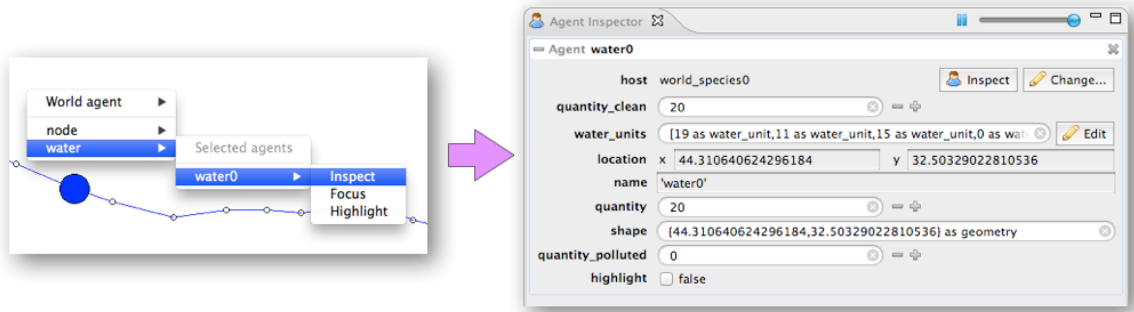
Species inspector

The species inspector provides informations about all the species present in a model. The species inspector is Available through the **Agents** menu.

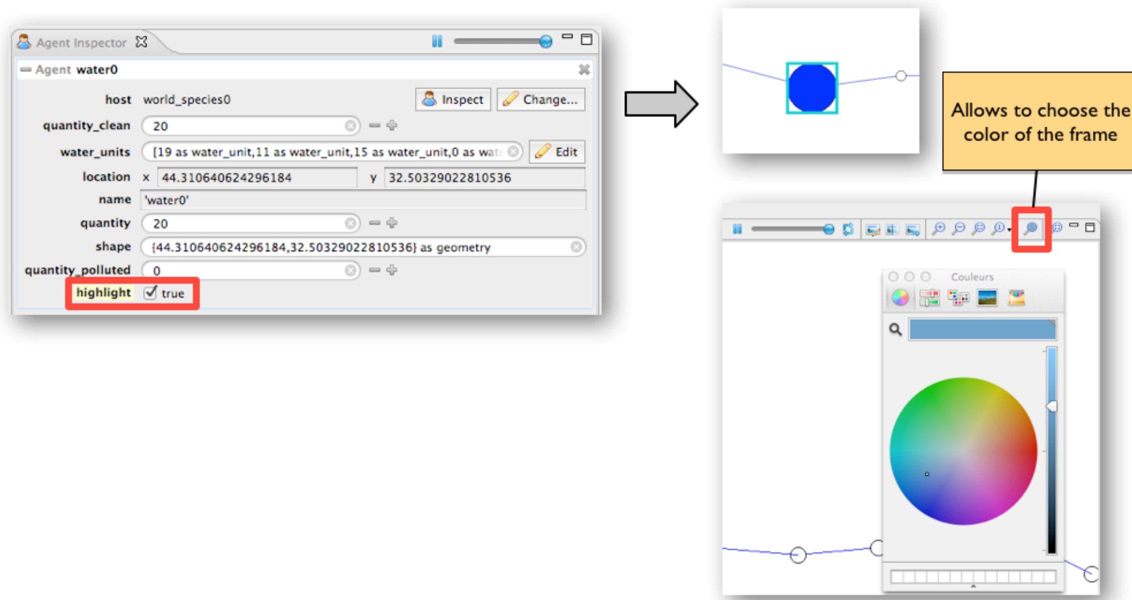


Agent inspector

The agent inspector provides information about one specific agent. It also allows to change the values of its variables during the simulation. The agent inspector is available from the **Agents** menu, by right_clicking on a display, in the species inspector or when inspecting another agent.



It is possible to «highlight» the selected agent.



—

Modeling Guide

Introduction to the Modeling Guide

Introduction

The modeling guide describe how to define a model with GAMA. A model is a GAML file (or set of files, included in each other) that is composed of several sections (a description of these sections is provided [Sections15 here]).

GAML Language

- Actions (except built-in actions) and behaviors are written as sequences of statements.
- Statements use expressions to define conditions, data changes, computations, etc. A statement ends by a ";" (statement without block) and by a {...} (a block of statements).
- An expression is composed of variables, keywords and operators.
- Every expression (and, therefore, every variable) has a type: either a built-in type or a species, as species can be seen as extended data types.
- Species come preloaded with built-in variables and skills variables (and every species can make use of the global built-in variables). All these variables can be redefined in GAML and, of course, new variables can be declared.

Model sections overview

Model Sections

A model is structured in several sections.

include

This section allows to load another model file before loading the current file. This file can contain anything (whole definition of a model, definition of a species, of an environment, of global variables, etc.) as long as it respects the common structure of GAML models. Note that as many files as needed can be included. Example:

```
import "../include/schelling_common.gaml"
import "../include/data_global.gaml"
```

global

This "global" section defines the "world" agent, a special agent of a GAMA model. We can define variables and behaviours for the "world" agent. Variables of "world" agent are global variables thus can refered by agents of other species or other places in the model source code. Example:

```
global {
    int numberBugs <- 100;
    float globalMaxConsumption <- 1;
    float globalMaxFoodProdRate <- 0.01;
    init {
        create bug number: numberBugs;
    }
}
```

entities

Definitions of species are placed int this section. Example:

```
entities {
    species bug  {
        float evol <- 1;
        rgb color <- rgb ([255, 255/evol, 255/evol]);
        float maxConsumption <- globalMaxConsumption;
        stupid_cell myPlace update: location as stupid_grid;
        reflex basic_move {
            let destination <- one_of ((myPlace neighbours_at 4) where empty(each.agents));
            if  destination != nil {
                set location <- destination;
            }
        }
        reflex grow {
            let transfer <- min ([maxConsumption, myPlace.food]);
            set evol <- evol + transfer;
            set myPlace.food <- myPlace.food - transfer;
```

```
        }
        aspect basic {
            draw shape: circle color: color size: 1;
        }
    }
}
```

environment

This section contains definitions of environments. GAMA supports three types of topologies for environments: continuous, grid and graph. By default, the world agent (i.e. the global agent that contains all of the other agents) has a continuous topology and its geometry is a rectangle of size 100mx100m. The size of the rectangle can be defined:

- using the width and height facets:

```
environment width: 200 height: 100
```

- using the bounds facet, with:
  - a point ({x,y}):

```
environment bounds: {200,100}
```

- a shapefile (GIS): envelope of all the data contained in the shapefile:

```
environment bounds: "buildings.shp"
```

- a raster file (asc):

```
environment bounds: "mnt.asc"
```

- a image file (png, jpg, tiff) linked to a world file (respectively pgw, jgw, tfw):

```
environment bounds: "mnt.png"
```

- a list of files (shapefile or ASC) : union of their envelopes:

```
environment bounds: ["buildings.shp", "mnt.asc"]
```

This section could include the definition of one or several environments with grid topology. Example:

```
environment width: 100 height: 100 {
  grid stupid_cell width: 100 height: 100 torus: false {
      rgb color <- rgb('black');
      float maxFoodProdRate <- globalMaxFoodProdRate;
      float maxConsumption <- globalMaxConsumption;
      float foodProd <- (rnd(1000) / 1000) * maxFoodProdRate;
      float food <- 0.0 update: food + foodProd;
  }
}
```

Each cell of a grid will be an agent. A grid represent a particular species of agents with a specific topology (grid topology). It is possible to define for a grid like for other species: variables , actions , behaviors and aspects .

experiment

This section defines experiments to run. Two kinds of experiment are supported: gui (graphic user interface) and batch (exploration of models). Example:

```
experiment my_experimentation type: gui {
  output {
      display stupid_display {
        grid stupid_cell;
        species bug aspect: basic;
      }
  }
}
```

Species section

Species Definition

Introduction

The agents' species are defined in the entities section. A model can contain any number of species. Species are used to specify the structure and behaviors of agents. Although the definitions below apply to all the species, some of them require specific declarations: the species of the world and the species of the environmental places.

Species declaration

The simplest way to declare a species is the following:

```
species a_name {
  [variable declarations]
  [action declarations]
  [behaviors]
}
```

for example:

```
species foo{} //it is also possible to directly write: species foo;
```

The agents that will belong to this species will only be provided with some built-in attributes and actions, a basic behavioral structure and nothing more. So, for instance, it is possible (somewhere else in the model) to write something like:

```
let foo_agents type: list of: foo <- [];
create foo number: 10 return: foo_agents;
ask foo_agents {
  write  ("my name is: " + name);
}
```

Which will result in the 10 agents writing their name, in turn, on the console. If the species declare variables, the structure of the agents is modified consequently. For instance:

```
species foo {
    float energy <- rnd (100) min: 0 max: 100 update: energy - 0.001;
}
```

Will give each agent an amount of energy (between 0 and 100), which will decrease over time until it reaches 0. The species can also declare actions that will supplement the built-in ones and extends the possibilities of the agents. Here, we provide two possible actions for agents of species foo, eating and stealing energy:

```
species foo {
    float energy <- rnd (100) min: 0 max: 100 update: energy - 0.001;
    action eat {
        set energy <- energy + rnd (2);
    }
    action steal {
        let another_agent type:foo <- any ((foo as list) - self);
        if (another_agent != nil) and ((another_agent.energy) > 0){
```

```
          set another_agent.energy <- another_agent.energy - 0.01;
          set energy <- energy + 0.01;
        }
    }
}
```

Of course, these actions do nothing unless they are called either by behaviors or by other agents. One might for example extend the previous example like:

```
let foo_agents type: list of: foo <- [];
create foo number: 1000 return: foo_agents;
ask (100 among (foo)) {
    do eat;
}
ask (100 among (foo)) {
    do steal;
}
```

In this example, we create 1000 foos, ask 100 of them to eat, and another 100 of them to steal energy. If these commands are done repetitively (for example, every turn in the world), they will result in a somewhat complex dynamic distribution of the energy between the foos. Of course, the dynamics of foos can also be declared from within their species. If we change slightly the declaration of foo like this:

```
species foo {
    float energy <- rnd (100) min: 0 max: 100 update: energy - 0.001;
    action eat {
        set energy <- energy + rnd (2);
    }
    action steal {
        let another_agent type:foo <- any ((foo as list) - self);
        if (another_agent != nil) and ((another_agent.energy) > 0){
          set another_agent.energy <- another_agent.energy - 0.01;
          set energy <- energy + 0.01;
        }
  }
  reflex {
        if flip (0.1) {
          do eat;
        }
        if  flip (0.1) {
          do steal;
        }
    }
}
```

We obtain agents that execute the reflex every turn and decide independently to eat or steal energy. Once they are created using

```
create foo number:1000;
```

they behave by their own.

Skills: behavioral plug-ins

Basic agents like the previous ones cannot, however, do many things. That's what skills are for. Example:

```
species foo skills: [moving]{
  ...
}
```

makes foos benefit from a set of variables and behaviors declared by the situated skill. Skills are like plug-ins written in Java and can provide a lot of new functionality to the agents.

Aspects: display properties

The aspect section allows to define the display of of the agents. it is possible to define different displays (i.e. different aspect sections) for a same species. In this context, the user will be able to change the display drawn during the simulation execution. The command **draw** allows to draw a shape (line, circle or square), a icon, a text or the agent geometry. this command has several facets:

- shape: optional, can be either "line", "circle", "square" or "geometry (in this case, the geometry of the agent will be drawn).
- geometry: any arbitrary geometry, that will only be affected by the color facet.
- text: string, optional, the text to draw.
- image: string, optional, path of the icon to draw (JPEG, PNG, GIF).
- color: rgb, optional, the color to use to display the shape/text/icon/geometry.
- size: float, size of the shape/text/icon (not use in the context of the drawing of a geometry).
- at: point, location where the shape/text/icon is drawn (not use in the context of the drawing of a geometry).
- to: point, terminal location of the line (only use in the in the context of the drawing of a line).
- rotate: int, orientation of the shape/text/icon (not use in the context of the drawing of a geometry).
- z: float, optional (only works if the type of the display is opengl) Add a height to the geometry previously defined (a point becomes a sphere, a line becomes a plan, a circle becomes a cylinder, a square becomes a cube,a polygon becomes a polyhedron with height equal to the z value). Note: This only works if a the agent has already a geometry.

For example, the following model allows to define three displays for the agent: one named "info", another named "icon" and the last one named "default".

```
aspect info {
    draw shape: square at: location size: 2 rotate: heading;
    draw geometry: square(2) rotated_by heading;
    draw shape: line at: location to:destination + (destination - location) color:'white';
    draw shape: circle at: location size:4 empty:true color:'white';
    draw text: heading color: 'white' size:1;
    draw text: state color: 'white' size:1 at:my location + {1,1};
}

aspect icon {
    draw image: shape at: location size: 2 rotate: heading;
}
aspect default {
    draw shape: square at: location empty: !hasFood color:'yellow' size: 2 rotate: heading;
}
```

3D aspect with display type:opengl`:

```
aspect sphere {
  draw geometry: geometry (point([location.x,location.y])) z:0.1;
}
aspect plan{
  draw geometry: geometry (line ([{0,0},{10,10}]))  z:10 ;
}
aspect cylinder {
  draw geometry: circle(1) z:1;
```

```
}
aspect cube {
  draw geometry: square(1) z:1;
}
aspect polyhedron{
  draw geometry: polygon([{7,5.5}, {7.5,5}, {8.5,5}, {9,5.5},{9,6.5},{8.5,7},{7.5,7},{7,6.5}])
z:2;
}
```

Parent: inheritance of species

A species can be declared as a child of another species, using the parent property. For instance :

```
species foo skills:[moving] parent:bar {
  ...
}
```

will make foo "inherit" from the definition of bar. What does "inherit" precisely mean in this context ?

- skills declared in bar, together with their built-in attributes and actions, are copied to foo and added to the possible new skills defined in foo.
- variables declared in bar, are identically copied to foo unless a variable with the same name is defined in foo, in which case this redefinition is kept. This also applies to built-in variables. The type of the variable can be changed in this process as well (but be careful when doing it, since inherited behaviors can rely on the previous type).
- actions declared in bar are identically copied to foo unless an action with the same name is defined in foo, in which case this redefinition is kept.
- reflexes declared in bar are identically copied to foo unless a reflex with the same name is defined in foo, in which case this redefinition is kept. Unnamed reflexes from both species are kept in the definition of foo.
- behaviors declared in bar are identically copied to foo unless a behavior of the same type with the same name is defined in foo, in which case this redefinition is kept.
- inits are treated differently : each of the init reflexes defined in bar and foo are kept in foo and they are executed in the order of inheritance (ie. bar 's one first, then foo 's one).

Scheduling description

The modeler can specify the scheduling information of a species. The scheduling information composes of the execution frequency and the list of agent to be scheduled.

- the execution frequency is the frequency which agents of the species are considered to be scheduled.
- "the list of agent to be scheduled" is an expression returning a list of agent dynamically evaluated at runtime.

```
species foo skills:[moving] parent:bar frequency: 2 schedules: (list (foo)) where (each.energy >
50)  {
    var energy type: float init: rnd (100) min: 0 max: 100 value: energy - 0.001;
  ...
}
```

- frequency: consider to schedule agents of the "foo" species every 2 simulation step.
- schedules: is an expression of the list of agent to be scheduled, this expression returns "foo" agents having energy greater than 50.

Hence, every 2 simulation step, "foo" agents having energy greater than 50 are scheduled.

Topology description

The topology describes the spatial organization of the species. This imposes constraint on the movement and perception (neighborhood) of the species' agents. GAMA supports three types of topology: continuous, grid and graph.

```
species foo skills:[moving] parent:bar topology: (square (10)) at_location {50, 50}  {
  ...
}
```

Topology of the "foo" species is a square of 10 meters each side at location {50, 50}.

Nesting species

A species can be defined inside another species. The enclosing species is the macro-species. The enclosed species is the micro-species. A model has "world" species as top-level species. The "world" species has one special agent ("world" agent) playing the role of the global context. The possibility to establish micro-macro relationship, to specify the scheduling description and the topology description enable the modeler to develop multi-scale model.

```
species A  {
  ...
}
species B {
  species C parent: A {
  ...
  }
  species D {
  ...
  }
}
```

- "A" and "B" are micro-species of "world" species.
- "C" and "D" are micro-species of "B" species.
- "C" species is a sub-species of "A" species. So agents of "A" species can be [Commands14 captured] by an agent of "B" species to become a "C" agent, micro-agent of the "B" agent. Vice-versa, a "C" agent, micro-agent of a "C" agent, can be [Commands14 released] from the "C" agent to become an "A" agent.

Control: behavioral architecture

By default, species are created with a minimal behavioral architecture : they only allow the definition of reflexes as a way to define the agents' behaviors. As reflex-based agents are somewhat limited when it comes to maintaining a state between two steps or enabling the selection of behaviors, GAML provides the modeler with two possible behavioral architectures, EMF (for Etho-Modeling Framework) and FSM (Finite State Machines). Each of them gives the possibility to define new elements in addition to reflexes : respectively tasks and states.

Base: Java foundation

The corresponding class used to initialize agent. An advance feature of the GAMA platform allowing the third party developer to develop their own agent architecture using the Java programming language.

Variables definition

Declaration

Basis

Except temporary variables, which are declared with their own syntax within behaviors or actions (see the `let` statement for more details), all other variables are declared using the following one:

```
datatype var_name [optional_attributes: ...];
```

In this declaration, datatype refers to the name of a built-in type or a species declared in the model. The value of var_name can be any combination of letters and digits (plus the underscore, " _ ") that does not begin with a digit and that follows certain rules (see "Naming variables"). Examples of valid declarations are:

```
int i;
list my_list;
my_species name an_agent_of_my_species; // if my_species is declared in the model as a species.
```

These variables are given default values at their creation, depending on their datatype: Default Value

int	float	bool	string	list	matrix	point	rgb	graph	geometry
0	0.0	false	''	[]	nil	nil	black	nil	nil

Deprecated alternative. Another way of declaring variable is also supported (for compatibility sake only). This way is fully equivalent to the previous one. **It is considered as deprecated and should not be used anymore** :

```
var var_name type: datatype [optional_attributes: ...];
var i type:int;
var my_list type:list;
var an_agent_of_my_species type:my_species name; // if my_species is declared in the model as a species.
```

init or <-

When it is necessary to initialize the variable with another value than its default value, the init (or <-) attribute can be used.

```
datatype var_name <- initial_expression [optional_attributes:...];
```

which is equivalent to:

```
var var_name type: datatype <- initial_expression [optional_attributes:...];
```

and to:

```
var var_name type: datatype init: initial_expression [optional_attributes:...];
```

The initial_expression is expected to be of the same type as the variable (otherwise it is casted to the datatype). Its only (obvious) restriction is that it cannot refer to the variable being declared. Examples of valid declarations are:

```
int i <- 0;
var i type:int init: 0;
list my_list <- [i + 1, i + 2, i + 3];
var my_list type:list init: [i + 1, i + 2, i + 3];
agent an_agent <- self;
var an_agent type:agent init: self;
```

const

If the value of the variable is not intended to change over time, it is preferable to declare it as a constant in order to avoid any surprise (and to optimize, a little bit, the compiler's work). This is done with the const attribute set to true (if const is not declared, it is considered as false by default):

```
var var_name type: datatype init: initial_expression const: true [optional_attributes:...];
```

With this declaration, the variable var_name will keep the result of initial_expression as its value for the whole execution of the simulation.

update

What if, on the contrary, the value of the variable is supposed to change over time and the modeller wants to define this evolution? The update attribute is precisely available for this purpose. Basically, its contents is evaluated every time step and assigned to the variable. It means that, unless the contents of this attribute refers to the variable itself, every modification made in the model to the value of the variable will be lost and replaced with the evaluation of the expression.

```
datatype var_name <- initial_expression update: value_expression [optional_attributes:...];
```

All the variables of all the agents are updated at the same time, before they are given a chance to execute behaviors. Some examples of use for value:

- Automatically evolving variables:

```
int my_int <- 0 update: my_int + 1: // -> my_int is incremented by 1 every time step.
float my_float <- 100 update: my_float - (my_float / 100); // -> my_float is decremented by 1%
every time step.
```

- Sticky variables:

```
int sticky_int update: 100; // -> whatever the changes made in the model to sticky_int, its value
returns to 100 at the beginning of every step.
```

- Conditionally evolving variables:

```
int cond_int update: (my_int < 100) ? 0 : my_int / 10; // -> the value of cond_int depends on
that of my_int.
float log_my_int update: ln (my_int); // -> the value of "cond_int" is always coupled to that of
my_int.
```

Special attributes

function

The update attribute is computed only every step. But somethimes, we need more accurate updates (i.e. that the value of the variable evaluated each time we use it). The function facet (attribute) has been introduced to this purpose and has the following syntax:

```
type1 var1 function: {an_expression} [optional_attributes:...];
```

Once a function is declared, whenever the variable is used somewhere, the function is computed (so the value of the variable always remains accurate). The declaration of function is **incompatible** with both init or update (an error will be raised). A shortcut has also been introduced:

```
type1 var1 -> {an_expression} [optional_attributes:...];
```

parameter

This way of defining parameters is deprecated. It is still supported for compatibility sake. The parameters should now be defined into the experiment section (see parameters definition in GUI or parameters definition in batch experiment). This attribute can only be used in the context of global variables, i.e. variables declared in the world species in the global section. Indicates that the value of the variable will (or can) be defined by an external input : either a file or an optimization process (in the case of batch simulations), or the user (in the case of interactive simulations with a user interface). Makes **const** turns to false if it has been defined. For example, declaring:

```
int max_energy <- 300 parameter:true;
```

will translate to this in the user interface:

[Parameter Editor] < http://gama-platform.googlecode.com/files/parameter.png> In several cases, this interface will allow the user to change the value of the variable during a simulation. If behaviors depend on it, the outcome of the simulation will then be affected by these changes, which can be a great way to manually and interactively explore the effect of parameters on a model. More on this in the presentation of the interface. The value of parameter can be used to name the variable on the interface. Any sequence of characters will do. If true is used, then the name of the variable itself is used for the label. Example:

```
int max_energy <- 300 parameter: "Maximum energy for the agents";
```

max, min

These two attributes are only available in the context of int or float variables. They allow the modeler to control the values of the variable, by specifying a maximum and minimum value. The variable will never be allowed to be lower than the minimum or greater than the maximum.

```
int max_energy <- 300 min: 100 max: 3000;
```

min and max combine gracefully with the parameter attribute and allow to control what the user can enter, or the limits between which exploring the values of variables. from For int variables, when declared in a "grid species". Not documented, because it will be added to the declaration of matrices instead (and removed from int). An example of the current use can be found in the "ants_from_file.xml" model.

of

Only defined in the context of matrix and list variables. Allows to define the type/species of values contained in the list. For instance, it can be handy, sometimes, to fix the species of the agents in a list at once rather than having to use the of_species operator every time. An example of that with the re-declaration of the built-in neighbours variable in a model with only one species of agents:

```
list neighbours of:species (self);
```

Doing so enables the use of neighbours, in the following expressions, without having to specify which kind of agents are manipulated in it.

Naming variables

Reserved Keywords

In GAML, some keywords are already reserved with a specific meaning and cannot be used for naming variables (and also species, actions, etc.). They are :

- The names of the global built-in variables
- The names of the primitive data types and new species defined in the model.
- The special keywords used by the language.
- The names of the variables found in every species.
- The names of the variables defined in skills when a species declares their use.
- The names of the units that can attached to numeric values.

Naming conventions

A variable name can be sequence of alphanumeric characters (plus the underscore, " _ "). It should follow certain rules:

- it should not begin by a digit;
- it should not contain space caracters.

By convention, it is recommended that:

- variable name begins by a lower case letter.

Accessing variables

Direct access

Variables can be directly accessed differently depending on their status (i.e. the place where they were declared: global variable, species variable or temporary/local variables):

- global variables can be directly accessed everywhere,
- species variables can be directly accessed in the scope of the species declaration,
- temporary variables can be directly accessed only in the scope in which they have been declared.

For instance, we can have a look at the following example:

```
species animal {
  float energy <- 1000 min: 0 max: 2000 <- energy - 0.001;
  int age_in_years <- 1 value: age_in_years + int (time / 365);

  action eat {
      arg amount default: 0;
      let gain type: float <- amount / age_in_years;
      set energy <- energy + gain;
  }
  reflex feed {
```

```
      let food_found type: int <- rnd(100);
      do eat amount: food_found;
  }
}
```

- **Species declaration** Everywhere in the species declaration, we are able to directly name and use:
  - time , a global built-in variable,
  - energy and age_in_years , the two species variables.

  Nevertheless, in the species declaration, but outsite of the action eat and the reflex feed , we **cannot** name the variables:
  - amount , the argument of eat action,
  - gain , a local variable defined into the eat action,
  - food_found , the local variable defined into the feed reflex.
- **Eat action declaration** In the eat action declaration, we can directly name and use:
  - time , a global built-in variable,
  - energy and age_in_years , the two species variables,
  - amount , which is an argument to the action eat ,
  - gain , a temporary variable within the action.

  We **cannot** name and use the variables:
  - food_found , the local variable defined into the feed reflex.
- **feed reflex declaration** Similarly, in the feed reflex declaration, we can directly name and use:
  - time , a global built-in variable,
  - energy and age_in_years , the two species variables,
  - food_found , the local variable defined into the reflex.

  But we **cannot** access to variables:
  - amount , the argument of eat action,
  - gain , a local variable defined into the eat action.

Remote access

When an agent needs to get access to the variable of another agent, a special notation (similar to that used in Java) has to be used:

```
remote_agent.variable
```

where remote_agent can be the name of an agent, an expression returning an agent, self, myself, context or each. For instance, if we modify the previous species by giving its agents the possibility to feed another agent found in its neighbourhood, the result would be:

```
species animal {
  var energy type: float init: 1000 min: 0 max: 2000 value: energy - 0.001;
  var age_in_years type: int init: 1 value: age_in_years + int (time / 365);
  action eat {
      arg amount default: 0;
      let gain type: float <- amount / age_in_years;
      set energy <- energy + gain;
  }
  action feed {
      arg target type: animal;
      if (agent_to_feed != nil) and (agent_to_feed.energy < energy { // verifies that the agent
exists and that it need to be fed
            ask agent_to_feed {
                do eat amount: myself.energy / 10; // asks the agent to eat 10% of our own energy
            }
            set energy <- energy - (energy / 10); // reduces the energy by 10%
      }
  }
```

```
  reflex {
      let candidates type: animal value: agents_overlapping (10 around agent.shape); gathers all
the neighbours
      set agent_to_feed value: candidates with_min_of (each.energy); //grabs one agent with the
lowest energy
      do action: feed target: agent_to_feed; // tries to feed it
  }
}
```

In this example, agent_to_feed.energy, myself.energy and each.energy show different remote accesses to the variable energy. The dotted notation used here can be employed in assignments as well. For instance, an action allowing two agents to exchange their energy could be defined as:

```
action random_exchange {//exchanges our energy with that of the closest agent
    let one_agent type: animal <- agent_closest_to (self)/>
    let temp type: float <-one_agent.energy; // temporary storage of the agent's energy
    set one_agent.energy <- energy; // assignment of the agent's energy with our energy
    set energy <- temp;
}
```

Built-in Variables

Introduction

In GAMA, every agents have some built-in variables linked to their identification, their location and their components (an agent can contains other agents). These variables can be used like normal variables.

List of built-in agent variables

=== location ===

- type: point
- description: centroid of the agent shape ({x, y})
- default value: random point in the environment

- type: string
- description: name of the agent; used to describe it
- default value: species name + creation index

- type: geometry
- description: geometry of the agent. It can be point, a polyline or a polygon
- default value: by default, a point geometry located at the location

- type: list
- description: list of agents that have the current agent for macro agent
- default value: []

Behaviors

Choice of an agent behavior architecture

GAMA integrates several agent behavior architectures that can be used in addition to the common behavior structure:

- weighted_tasks
- sorted_tasks
- probabilistic_tasks
- fsm

The choice of an architecture (that is optional) is made through the **control** facet:

```
species ant control: fsm {
  ...
}
```

Common behavior structure

All agents (including the world and grid cells) are provided with a simple behavioral structure, based on reflexes. Species can define any number of reflexes within their body.

reflex

Attributes

- when: a boolean expression

Definition

A reflex is a sequence of statements that can be executed, at each time step, by the agent. If no attribute when are defined, it will be executed every time step. If there is an attribute **when** , it is executed only if the boolean expression evaluates to true. It is a convenient way to specify the behavior of the agents. Example:

```
reflex my_reflex { //Executed every time step
    write 'Executing the unconditional reflex';
}
```

```
reflex my_reflex when: flip (0.5){ //Only executed when flip returns true
    write 'Executing the conditional reflex';
```

init

Attributes

- when: a boolean expression

Definition

A special form of reflex that is evaluated only once when the agent is created, after the initialization of its variables, and before it executes any reflex. Only one instance of init is allowed in each species (except in case of inheritance, see this section). Useful for creating all the agents of a model in the definition of the world, for instance.

Task behavior models

GAMA integrated several task behaviors model architectures. Species can define any number of tasks within their body. At any given time, only one or several tasks are executed according to the architecture chosen:

- **weighted_tasks** : in this architecture, only the task with the maximal weight is executed.
- **sorted_tasks** : in this architecture, the tasks are all executed in the order specified by their weights (biggest first)
- **probabilistic_tasks** : this architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen each step.

task

Sub elements

Besides a sequence of statements like reflex, a task contains the following sub elements:

- weight: Mandatory. The priority level of the task.

Definition

Like reflex, a task is a sequence of statement that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler choses a task to execute basing on its current priority weight value.

FSM-based behaviors

FSM (Finite State Machine) is a finite state machine based behavior model. During its life cycle, agent possesses several states. At any given time step, an agent is in one state.

state

Attributes

- initial: a boolean expression, indicates the initial state of agent.
- final: a boolean expression, indicates the final state of agent.

Sub elements

- enter: a sequence of statements to execute upon entering the state.
- exit: a sequence of statements to execute right before exiting the state.
- transition: specifies the next state of the life cycle.

Definition

A state like a reflex can contains several statements that can be executed, at each time step, by the agent. For an exemple of state, please have a look at the definition of "ant" species in this FSM exemple.

Actions

Definition of an action

An action is a capability available to the agents of a species (what they can do). It is a block of statements that can be used and reused whenever needed. An action can accept arguments (statement **arg** nom_arg type: type). An action can return a result (statement **return**). From a general point of view, an action is declared with:

```
action action_name {
  [arg arg_name type: var_type;]
  [sequence_of_statements]
  [return value;]
}
```

For example:

```
action action_addition {
  arg arg1 type: int;
  arg arg2 type: int;
  return arg1 + arg2;
}
```

Some actions, called primitives, are directly coded in Java : for instance, the write action defined for all the agents.

Use of an action

There are two ways to call an action: using a statement or as part of an expression

- action that does not return a result:

```
do action_name arg1: v1 arg2: v2;
```

- action that returns a result:

```
set my_var <- self action_name [arg1::v1, arg2::v2];
```

The **self** keyword denotes the agent that will execute the action (the action must be defined in its species). For action that returns a result, use of a map to pass arguments.

Skills

Overview

Skills are built-in modules that provide a set of related built-in variables and built-in actions (in addition to those already provided by GAMA) to the species that declare them. A declaration of skill is done by filling the skills attribute in the species definition:

```
species my_species skills: [skill1, skill2] {
    ...
}
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. The list of available skills in GAMA is:

- moving: for agents that need to move.

So, for instance, if a species is declared as:

```
species foo skills: [moving]{
...
}
```

its agents will automatically be provided with the following variables : "speed, heading, destination (r/o)" and the following actions: "move, goto, wander, follow" in addition to those built-in in species and declared by the modeller. Most of these variables, except the ones marked read-only, can be customized and modified like normal variables by the modeller. For instance, one could want to set a maximum for the speed; this would be done by redeclaring it like this:

```
float speed max:100 min:0;
```

Or, to obtain a speed increasing at each simulation step:

```
float speed max:100 min:0  <- 1 update: speed * 1.01;
```

Or, to change the speed in a behavior:

```
if speed = 5 {
    set speed <- 10;
}
```

moving

variables

speed

- float, the speed of the agent, in meter/second.

heading

- int, the absolute heading of the agent in degrees (in the range 0-359).

destination

- point, read-only, continuously updated destination of the agent with respect to its speed and heading.

actions

follow

moves the agent along a given path passed in the arguments.

- → **speed** : float, optional, the speed to use for this move (replaces the current value of speed).
- → **path** : a path to be followed

```
do follow speed: speed * 2 path: road_path;
```

goto

moves the agent towards the target passed in the arguments.

- → **target** : point or agent, mandatory, the location or entity towards which to move.
- → **speed** : float, optional, the speed to use for this move (replaces the current value of speed).
- → **on** : list, agent, graph, geometry, optional, that restrains this move (the agent moves inside this geometry).
- → **return_path** : bool, optional, if true, the action returns the path followed
- ← return: null or the path followed if **return_path** is set to *true*

```
do goto target: one_of (list (species (self))) speed: speed * 2 on: road_network;
```

move

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- → **speed** : float, optional, the speed to use for this move (replaces the current value of speed).
- → **heading** : int, optional, the direction to take for this move (replaces the current value of heading).
- → **bounds** : localized entity, optional, the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry).

```
do move speed: speed - 10 heading: heading + rnd (30) bounds: agentA;
```

wander

moves the agent towards a random location at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- → **speed** : float, optional, the speed to use for this move (replaces the current value of speed).
- → **amplitude** : int, optional, a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2).
- → **bounds** : localized entity or geometry, optional, the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry).

```
do wander speed: speed - 10 amplitude: 120 bounds: agentA;
```

Experiment section

Experiment definition

Experiment definition

An experiment block defines how a model can be simulated (executed). Several experiments can be defined for a model. They are defined using:

```
experiment exp_name type: gui/batch {
  …
}
```

Experiment types

There are two type of experiments :

- **gui** : experiment with a graphical interface, which displays its input parameters and outputs.
- **batch** : Allows to setup a series of simulations (w/o graphical interface).

GUI

Definition of GUI experiment

A GUI experiment allows to display a graphical interface with input parameters and outputs (display, file, monitor...). A GUI experiment is defined by:

```
experiment exp_name type: gui {
  [input]
  [output]
}
```

Input

Parameters

Experiments can define input, i.e. parameters. Defining parameters allows to make the value of a global variable definable by the user through the user graphic interface. A parameter is defined as follow:

```
  parameter title var: global_var category: cat;
```

With:

- title: string to display
- var: reference to a global variable (defined in the global section)
- category: string used to «store» the operators on the UI (optional)

Example:

```
  parameter "Value of toto: " var: toto;
```

User command

Experiments can also define some commands (buttons in the GUI) allowing the user to interact with the simulation, i.e. to call an action defined in the model. Commands can either call directly an existing action (with or without arguments) or be followed by a block that describes what to do when this command is run. The syntax is:

```
  user_command cmd_name action: action_name;
```

or

```
  user_command cmd_name action: action_name with: [arg1::val1, arg2::val2, ...];
```

or

```
  user_command cmd_name {
    [statements]
  }
```

These commands are not executed when an agent runs. Instead, they are collected and appear as buttons above the parameters of the simulation.

Output

Output blocks define how to visualize a simulation (with one or more display blocks that define separate windows)

```
experiment exp_name type: gui {
  [input]
  output {
    [display statements]
    [monitor statements]
    [file statements]
  }
}
```

Display

A display refers to a part of the independent and mobile interface part that can display species, images, texts or charts. There exist several kinds of display:

- classical **displays** (without specific type) used to species, text, image, charts...

```
display my_display { ... }
```

- **opengl displays** (display with type: opengl) used to display species, text or image. It allows to display 3D models.

```
display my_display type: opengl { ... }
```

- **graphdisplay** is a special kind of display dedicated to the dirplay of graphs. It is based on the grapstream library. Note that it provides a pretty way of displaying graphs without regard of spatiality.

```
graphdisplay monNom2 graph: my_graph lowquality:true ;
```

Each display can be refreshed independently by defining the facet **refresh_every: nb (int)** (the display will be refreshed every nb steps of the simulation) Each display can include different layers (like in a GIS). Although every combinaison of any number of following layers are allowed in GAML, it is recommended to distinguish displays with species, image and/or text and display with charts (and text).

agents layer

agents allows the modeler to display only the agents that fulfill a given condition.

```
display my_display {
  agents layer_name value: expression [additional options];
}
```

Additional options include:

- **value** (type = container) the set of agents to display
- **aspect** : the name of the aspect that should be used to display the species.
- **transparency** (type = float, from 0 to 1): the transparency rate of the agents (1 means no transparancy)

- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the species is refreshed. (usefull in case of agents that do not move)
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying agents
- **focus** (type = agent)

For instance, in a segregation model, agents will only display un happy agents:

```
display Segregation {
  agents agentDisappear value : people as list where (each.is_happy = false) aspect:
with_group_color;
}
```

species layer

species allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them. The general syntax is:

```
display display_name {
  species species_name [additional options];
}
```

Additional options include:

- **aspect** : the name of the aspect that should be used to display the species.
- **transparency** (type = float, from 0 to 1): the transparency rate of the agents (1 means no transparancy)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the species is refreshed. (usefull in case of agents that do not move)
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying agents

For instance it could be:

```
display my_display{
  species agent1 aspect: base ;
}
```

Species can be superposed on the same plan:

```
display my_display{
  species agent1 aspect: base;
  species agent2 aspect: base;
  species agent3 aspect: base;
}
```

Species can be placed on different z values for each layer using the opengl display. z:0 means the layer will be placed on the ground and z=1 means it will be placed at an height equal to the maximum size of the environment.

```
display my_display type: opengl{
  species agent1 aspect: base z:0;
  species agent2 aspect: base z:0.5;
```

```
  species agent3 aspect: base z:1;
}
```

image layer

image allows modeler to display an image (e.g. as background of a simulation). The general syntax is:

```
display display_name {
  image layer_name file: image_file [additional options];
}
```

Additional options include:

- **file** (type = string): the name/path of the image (in the case of a raster image)
- **gis** (type = string): the name/path of the shape file (to display a shapefile as background, without creating agents from it)
- **color** (type = color): in the case of a shapefile, this the color used to fill in geometries of the shapefile
- **transparency** (type = float, from 0 to 1): the transparency rate (1 means no transparancy)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the image is refreshed.
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying the image

For instance:

```
display my_display{
  image name:'background' file:'../images/my_backgound.jpg';
}
```

Or

```
display city_display refresh_every: 1 {
  image testGIS gis: "../includes/building.shp" color: rgb('blue');
}
```

It is also possible to superpose images on different layers in the same way as for species using opengl display.

```
display my_display type:opengl{
  image name:'image1' file:'../images/image1.jpg';
  image name:'image2' file:'../images/image2.jpg' z:0.5;
}
```

chart layer

chart allows modeler to display a chart: this enables to display a specific value of the model at each iteration. GAMA can display various chart types: time series (**series**), pie charts (**pie**) and histograms (**histogram**).

```
display chart_display [additional options] {
  chart  "chart name" type: series {
  data data1 value: mydata1 color: rgb('blue') ;
  data data2 value: mydata2 color: rgb('blue') ;
}
```

Additional options include:

- **transparency** (type = float, from 0 to 1): the transparency rate of the layer (1 means no transparancy)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **background** (type = color): the background color
- **axes** (type = color): the axes color
- **type** : the type of chart. It could be **histogram** , **series** or **pie** .
- **style** : the style of the chart. It could be: **exploded** , **stack** , **bar** or **3d**
- **font** : the font used for legends.
- **color** (type = color)

text layer

text allows the modeler to display a string (that can change at each step) in a given position of the display. The general syntax is:

```
display my_display {
  text  my_text value: expression [additional options];
```

Additional options include:

- **value** (type = string) the string to display
- **transparency** (type = float, from 0 to 1): the transparency rate of the layer (1 means no transparancy)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **font** : the font used for the text
- **color** (type = color): the color used to display the text
- **refresh** (type = boolean, **opengl only**): specify whether the layer is refreshed.
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying text

For instance:

```
display my_display {
  text agents value : 'Carrying ants : ' + string ( int ( ant as list count (each . has_food ) )
+ int ( ant as list count ( each . state =    'followingRoad' ) ) ) position : { 0.5 , 0.03 }
color : rgb ( 'black' ) size: { 1 , 0.02 };
}
```

File

There are several ways to save data in a file in GAMA:

- The simplest way is to use the save statement (not in the experiment section)
- Use of the **file** output in the output section. In this case, a new line is added to the end of the file at each simulation step.

```
  file name: "file_name" type: file_type data: data_to_write;
```

with :

- file_type: text, csv or xml
- file_name: string
- data_to_write: string

Example:

```
file name: "results" type: text data: time + "; " + nb_preys + ";" + nb_predators;
```

Monitor

A monitor allows to follow the value of an arbitrary expression in GAML. Definition of a monitor:

```
monitor monitor_name value: an_expression refresh_every: nb_steps;
```

with:

- value: mandatory, its value will be displayed in the monitor.
- refresh_every: int, optional : number of simulation steps between two computations of the expression (default is 1).

Example:

```
monitor "nb preys" value: length(prey as list);
```

Batch

Definition

Batch experiment allows to execute numerous successive simulation runs. It is used to explore the parameter space of a model or to optimize a set of model parameters. A Batch experiment is defined by:

```
experiment exp_title type: batch {
  [parameter to explore]
  [exploration method]
}
```

The batch experiment facets

Batch experiment have the following three facets:

- until: (expression) Specifies when to stop each simulations. Its value is a condition on variables defined in the model. The run will stop when the condition is evaluated to true. If omitted, the first simulation run will go forever, preventing any subsequent run to take place (unless a halt command is used in the model itself).
- repeat: (integer) A parameter configuration corresponds to a set of values assigned to each parameter. The attribute repeat specifies the number of times each configuration will be repeated, meaning that as many simulations will be run with the same parameter values. Different random seeds are given to the pseudo-random number generator. This allows to get some statistical power from the experiments conducted. Default value is 1.
- keep_seed: (boolean) If true, the same series of random seeds will be used from one parameter configuration to another. Default value is false.

```
experiment my_batch_experiment type: batch repeat: 5 keep_seed: true until: time = 300 {
  [parameter to explore]
  [exploration method]
}
```

Parameter definition

The **parameter** elements specifies which model parameters will change through the successive simulations. A parameter is defined as follows:

```
parameter title var: global_variable + possible_values
```

There are 2 ways to describe the range in which the value of the parameter will be explored :

- Explicit list: **among** : values_list

```
  parameter "Value of toto:" var: toto among: [1, 3, 7, 15, 100];
```

- Range : **min** : min_value **max** : max_value **step** : increment_step

```
  parameter "Value of toto:" var: toto min: 1 max: 100 step: 2;
```

For Strings and Booleans, you can only use the Explicit List. Each Batch methods may accept only some kind of definitions and parameter types. See the description of each of them for details.

The method element

If this element is omitted, the batch will run in a classical way, changing one parameter at each step until all the possible combinations of parameter values have been covered. See #Exhaustive exploration of the parameter space for details. The optional method element controls the algorithm which drives the batch. It must contain at least a name attribute to specify the algorithm to use, and for an optimization method, a minimize or a maximize attribute defining the expression to be optimized. Each combination of parameter values is tested repeat times. The fitness of one combination is the average of fitness values obtained with those repetitions. There might be additional attributes for tuning the exploration algorithm. See below for a description of the available methods.

Batch Methods

Several batch methods are currently available. Each is described below.

Exhaustive exploration of the parameter space

Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way. Example (models/ants/batch/ant_exhaustive_batch.xml)

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
}
```

The order of the simulations depends on the order of the param. In our example, the first combinations will be the followings:

- evaporation_rate = 0.1, diffusion_rate = 0.1, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 0.4, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 0.7, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 1.0, (2 times)
- evaporation_rate = 0.2, diffusion_rate = 0.1, (2 times)
- ...

Note: this method can also be used for optimization by adding an method element with maximize or a minimize attribute:

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
    method exhaustive maximize : food_gathered;
}
```

Hill Climbing

Name: hill_climbing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article. Algorithm:

```
Initialization of an initial solution s
iter = 0
While iter <= iter_max, do:
  Choice of the solution s' in the neighborhood of s that maximize the fitness function
  If f(s') > f(s)
    s = s’
  Else
    end of the search process
  EndIf
  iter = iter + 1

EndWhile
```

Method parameters:

- iter_max: number of iterations

Example (models/ants/batch/ant_hill_climbing_batch.xml):

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
    method hill_climbing iter_max: 50 maximize : food_gathered;
}
```

Simulated Annealing

Name: annealing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article. Algorithm:

```
Initialization of an initial solution s
temp = temp_init
While temp > temp_end, do:
  iter = 0
  While iter < nb_iter_cst_temp, do:
    Random choice of a solution s2 in the neighborhood of s
    df = f(s2)-f(s)
    If df > 0
      s = s2
    Else,
      rand = random number between 0 and 1
      If rand > exp(df/T)
        s = s2
      EndIf
    EndIf
    iter = iter + 1
  EndWhile

EndWhile
```

Method parameters:

- temp_init: Initial temperature
- temp_end: Final temperature
- temp_decrease: Temperature decrease coefficient
- nb_iter_cst_temp: Number of iterations per level of temperature

Example (models/ants/batch/ant_simulated_annealing_batch.xml):

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
    method annealing temp_init: 100  temp_end: 1 temp_decrease: 0.5 nb_iter_cst_temp: 5
maximize : food_gathered;
}
```

Tabu Search

Name: tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article. Algorithm:

```
Initialization of an initial solution s
tabuList = {}
iter = 0
While iter <= iter_max, do:
  Choice of the solution s2 in the neighborhood of s such that:
    s2 is not in tabuList
    the fitness function is maximal for s2
  s = s2
  If size of tabuList = tabu_list_size
    removing of the oldest solution in tabuList
  EndIf
  tabuList = tabuList + s
  iter = iter + 1

EndWhile
```

Method parameters:

- iter_max: number of iterations
- tabu_list_size: size of the tabu list

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
    method tabu iter_max: 50 tabu_list_size: 5 maximize : food_gathered;
}
```

Reactive Tabu Search

Name: reactive_tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti

et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle. Method parameters:

- iter_max: number of iterations
- tabu_list_size_init: initial size of the tabu list
- tabu_list_size_min: minimal size of the tabu list
- tabu_list_size_max: maximal size of the tabu list
- nb_tests_wthout_col_max: number of movements without collision before shortening the tabu list
- cycle_size_min: minimal size of the considered cycles
- cycle_size_max: maximal size of the considered cycles

```
experiment Batch type : batch repeat : 2 keep_seed : true until : (food_gathered = food_placed )
or ( time > 400 ) {
    parameter name: 'Evaporation:' var : evaporation_rate among : [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter name: 'Diffusion:' var : diffusion_rate min : 0.1 max : 1.0 unit : 'rate every
cycle (1.0 means 100%)' step : 0.3;
    method tabu iter_max: 50 tabu_list_size_init: 5 tabu_list_size_min: 2 tabu_list_size_max: 10
nb_tests_wthout_col_max: 20 cycle_size_min: 2 cycle_size_max: 20 maximize : food_gathered;
}
```

Genetic Algorithm

Name: genetic Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all. This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article. The principle of GA is to search an optimal solution by applying evolution operators on an initial population of solutions There are three types of evolution operators:

- Crossover: Two solutions are combined in order to produce new solutions
- Mutation: a solution is modified
- Selection: only a part of the population is kept. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Representation of the solutions:

- Individual solution: {Param1 = val1; Param2 = val2; …}
- Gene: Parami = vali

Initial population building: the system builds nb_prelim_gen random initial populations composed of pop_dim individual solutions. Then, the best pop_dim solutions are selected to be part of the initial population. Selection operator: roulette-wheel selection: the probability to choose a solution is equals to: fitness(solution)/ Sum of the population fitness. A solution can be selected several times. Ex: population composed of 3 solutions with fitness (that we want to maximize) 1, 4 and 5. Their probability to be chosen is equals to 0.1, 0.4 and 0.5. Mutation operator: The value of one parameter is modified. Ex: The solution {Param1 = 3; Param2 = 2} can mute to {Param1 = 3; Param2 = 4} Crossover operator: A cut point is randomly selected and two new solutions are built by taking the half of each parent solution. Ex: let

{Param1 = 4; Param2 = 1} and {Param1 = 2; Param2 = 3} be two solutions. The crossover operator builds two new solutions: {Param1 = 2; Param2 = 1} and {Param1 = 4; Param2 = 3}. Method parameters:

- pop_dim: size of the population (number of individual solutions)
- crossover_prob: crossover probability between two individual solutions
- mutation_prob: mutation probability for an individual solution
- nb_prelim_gen: number of random populations used to build the initial population
- max_gen: number of generations

```
experiment Genetic type : batch repeat : 2 keep_seed : true until : (food_gathered =
food_placed ) or ( time > 400 ) {
    parameter  'Evaporation:' var : evaporation_rate among: [ 0.1 , 0.2 , 0.5 , 0.8 , 1.0 ]
unit : 'rate every cycle (1.0 means 100%)';
    parameter  'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:'rate every cycle (1.0
means 100%)' step: 0.3;
    method genetic maximize: food_gathered pop_dim: 5 crossover_prob: 0.7 mutation_prob: 0.1
nb_prelim_gen: 1 max_gen: 20;
}
```

GAML language

—

Data Types

Table of Contents

Primitive built-in types

bool

- **Definition:** primitive datatype providing two values: true or false .
- **Litteral declaration:** both true or false are interpreted as boolean constants.
- **Other declarations:** expressions that require a boolean operand often directly apply a casting to bool to their operand. It is a convenient way to directly obtain a bool value.

```
bool (0) -> false
```

Top of the page

float

- **Definition:** primitive datatype holding floating point values comprised between -(2-252) * 21023 and -(2-252) * 21023.
- **Comments:** this datatype is internally backed up by the Java double datatype.
- **Litteral declaration:** decimal notation 123.45 or exponential notation 123e45 are supported.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to float to their operand. Using it is a way to obtain a float constant.

```
float (12) -> 12.0
```

Top of the page

int

- **Definition:** primitive datatype holding integer values comprised between -231 and 231 - 1
- **Comments:** this datatype is internally backed up by the Java int datatype.
- **Litteral declaration:** decimal notation like 1, 256790 or hexadecimal notation like #1209FF are automatically interpreted.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to int to their operand. Using it is a way to obtain an integer constant.

```
int (234.5) -> 234.
```

Top of the page

string

- **Definition:** a datatype holding a sequence of characters.
- **Comments:** this datatype is internally backed up by the Java String class. However, contrary to Java, strings are considered as a primitive type, which means they do not contain character objects. This can be seen when casting a string to a list using the list operator: the result is a list of one-character strings, not a list of characters.
- **Litteral declaration:** a sequence of characters enclosed in quotes, like 'this is a string' . If one wants to literally declare strings that contain quotes, one has to double these quotes in the declaration. Strings accept escape characters like \n (newline), \r (carriage return), \t (tabulation), as well as any Unicode character (\uXXXX`).
- **Other declarations:** see string
- **Example:** see [Operators_14 string operators] .

Top of the page

Complex built-in types

Contrarily to primitive built-in types, complex types have often various attributes. They can be accessed in the same way as attributes of agents:

```
let nom_var type: complex_type <- init_var;
let attr_var type: type_attr <- nom_var.attr_name;
```

For example:

```
let fileText type: file <- "../data/cell.Data";
let fileTextReadable type: bool <- fileText.readable;
```

agent

- **Definition:** a generic datatype that represents an agent whatever its actual species.
- **Comments:** This datatype is barely used, since species can be directly used as datatypes themselves.
- **Declaration:** the agent casting operator can be applied to an int (to get the agent with this unique index), a string (to get the agent with this name).

Top of the page

container

- **Definition:** a generic datatype that represents a collection of data.
- **Comments:** a container variable can be a list, a matrix, a map... Conversely each list, matrix and map is a kind of container. In consequence every container can be used in container-related operators.
- **See also:** [Operators_14 Container operators]
- **Declaration:**

```
container c  <- [1,2,3];
container c  <- matrix [[1,2,3],[4,5,6]];
container c  <- map ["x"::5, "y"::12];
```

```
container c  <- list species1;
```

Top of the page

file

- **Definition:** a datatype that represents a file.
- **Built-in attributes:**
  - name (type = string): the name of the represented file (with its extension)
  - extension(type = string): the extension of the file
  - path (type = string): the absolute path of the file
  - readable (type = bool, read-only): a flag expressing whether the file is readable
  - writable (type = bool, read-only): a flag expressing whether the file is writable
  - exists (type = bool, read-only): a flag expressing whether the file exists
  - is_folder (type = bool, read-only): a flag expressing whether the file is folder
  - contents (type = container): a container storing the content of the file
- **Comments:** a variable with the file type can handle any kind of file (text, image or shape files...). The type of the content attribute will depend on the kind of file. Note that the allowed kinds of file are the followings:
  - text files: files with the extensions .txt, .data, .csv, .text, .tsv, .asc. The content is by default a list of string.
  - image files: files with the extensions .pgm, .tif, .tiff, .jpg, .jpeg, .png, .gif, .pict, .bmp. The content is by default a matrix of int.
  - shapefiles: files with the extension .shp. The content is by default a list of geometry.
  - properties files: files with the extension .properties. The content is by default a map of string::string .
  - folders. The content is by default a list of string.
- **Remark:** Files are also a particular kind of container and can thus be read, written or iterated using the container operators and commands.
- **See also:** [Operators_14 File operators]
- **Declaration:** a file can be created using the generic file (that opens a file in read only mode and tries to determine its contents), folder or the new_folder (to open an existing folder or create a new one) unary operators. But things can be specialized with the combination of the read / write and image / text / shapefile / properties unary operators.

```
folder(a_string)  // returns a file managing a existing folder
file(a_string) // returns any kind of file in read-only mode
read(text(a_string)) // returns a text file in read-only mode
read(image(a_string)) // does the same with an image file.
write(properties(a_string)) // returns a property file which is available for writing
                            // (if it exists, contents will be appended unless it is cleared
                            // using the standard container operations).
```

Top of the page

geometry

- **Definition:** a datatype that represents a vector geometry, i.e. a list of georeferenced points.
- **Built-in attributes:**
  - location (type = point): the centroid of the geometry
  - area (type = float): the area of the geometry
  - perimeter (type = float): the perimeter of the geometry

  - holes (type = list of geometry): the list of the hole inside the given geometry
  - contour (type = geometry): the exterior ring of the given geometry and of his holes
  - envelope (type = geometry): the geometry bounding box
  - width (type = float): the width of the bounding box
  - height (type = float): the height of the bounding box
  - points (type = list of point): the set of the points composing the geometry
- **Comments:** a geometry can be either a point, a polyline or a polygon. Operators working on geometries handle transparently these three kinds of geometry. The envelope (a.k.a. the bounding box) of the geometry depends on the kind of geometry:
  - If this Geometry is the empty geometry, it is an empty point.
  - If the Geometry is a point, it is a non-empty point.
  - Otherwise, it is a Polygon whose points are (minx, miny), (maxx, miny), (maxx, maxy), (minx, maxy), (minx, miny).
- **See also:** [Operators_14 Spatial operators]
- **Declaration:** geometries can be built from a point, a list of points or by using specific operators (circle, square, triangle...).

```
geometry varGeom <- circle(5);
geometry polygonGeom <- polygon([{3,5}, {5,6},{1,4}]);
```

Top of the page

graph

- **Definition:** a datatype that represents a graph composed of vertices linked by edges.
- **Built-in attributes:**
  - edges(type = list of agent/geometry): the list of all edges
  - vertices(type = list of agent/geometry): the list of all vertices
  - circuit (type = path): an approximate minimal traveling salesman tour (hamiltonian cycle)
  - spanning_tree (type = list of agent/geometry): minimum spanning tree of the graph, i.e. a sub-graph such as every vertex lies in the tree, and as much edges lies in it but no cycles (or loops) are formed.
  - connected(type = bool): test whether the graph is connected
- **Remark:**
  - graphs are also a particular kind of container and can thus be manipulated using the container operators and commands.
  - This algorithm used to compute the circuit requires that the graph be complete and the triangle inequality exists (if x,y,z are vertices then d(x,y)+d(y,z) < d(x,z) for all x,y,z) then this algorithm will guarantee a hamiltonian cycle such that the total weight of the cycle is less than or equal to double the total weight of the optimal hamiltonian cycle.
  - The computation of the spanning tree uses an implementation of the Kruskal's minimum spanning tree algorithm. If the given graph is connected it computes the minimum spanning tree, otherwise it computes the minimum spanning forest.
- **See also:** [Operators_14 Graph operators]
- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create road from: shape_file_road;
let the_graph type: graph <- as_edge_graph(list(road));
graph([1,9,5])        --: ([1: in[] + out[], 5: in[] + out[], 9: in[] + out[]], [])
graph([node(0), node(1), node(2)]      // if node is a species
```

```
graph(['a'::345, 'b'::13])  --:  ([b: in[] + out[b::13], a: in[] + out[a::345], 13: in[b::13] +
out[], 345: in[a::345] + out[]], [a::345=(a,345), b::13=(b,13)])
graph(a_graph)  --: a_graph
graph(node1)    --: null
```

Top of the page

list

- **Definition:** a composite datatype holding an ordered collection of values.
- **Comments:** lists are more or less equivalent to instances of [ArrayList] in Java (although they are backed up by a specific class). They grow and shrink as needed, can be accessed via an index (see @ or index_of), support set operations (like union and difference), and provide the modeller with a number of utilities that make it easy to deal with collections of agents (see, for instance, shuffle, reverse,where,sort_by,...).
- **Remark:** lists can contain values of any datatypes, including other lists. Note, however, that due to limitations in the current parser, lists of lists cannot be declared litteraly; they have to be built using assignments. Lists are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of expressions separated by commas, enclosed in square brackets, like [12, 14, 'abc', self] . An empty list is noted [].
- **Other declarations:** lists can be build litteraly from a point, or a string, or any other element by using the list casting operator.

```
list (1) -> [1]
```

```
let myList <-list [1,2,3,4];
myList at 2 => 3
```

Top of the page

map

- **Definition:** a composite datatype holding an ordered collection of pairs (a key, and its associated value).
- **Built-in attributes:**
  - keys (type = list): the list of all keys
  - values (type = list): the list of all values
  - pairs (type = list of pairs): the list of all pairs key::value
- **Comments:** maps are more or less equivalent to instances of Hashtable in Java (although they are backed up by a specific class).
- **Remark:** maps can contain values of any datatypes, including other maps or lists. Maps are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of pair expressions separated by commas, enclosed in square brackets; each pair is represented by a key and a value sperarated by '::'. An example of map is [agentA::'big', agentB::'small', agentC::'big'] . An empty map is noted [].
- **Other declarations:** lists can be built litteraly from a point, or a string, or any other element by using the map casting operator.

```
map (1) -> [1::1]
map ({1,5}) -> [x::1, y::5]
[]   // empty map
```

Top of the page

matrix

- **Definition:** a composite datatype that represents either a two-dimension array (matrix) or a one-dimension array (vector), holding any type of data (including other matrices).
- **Comments:** Matrices are fixed-size structures that can be accessed by index (point for two-dimensions matrices, integer for vectors).
- **Litteral declaration:** Matrices cannot be defined literally. One-dimensions matrices can be built by using the matrix casting operator applied on a list. Two-dimensions matrices need to to be declared as variables first, before being filled.

```
//builds a one-dimension matrix, of size 5
let mat1 type:matrix <- matrix ([10, 20, 30, 40, 50]);
//  builds a two-dimensions matrix with 10 columns and 5 lines, where each cell is initialized to
0.0
let mat2 type:matrix <- 0.0 as_matrix({10,5});
// builds a two-dimensions matrix with 2 columns and 3 lines, with initialized cells
let mat3 type:matrix <- matrix([["c11","c12","c13"],["c21","c22","c23"]]);
    -> c11;c21
      c12;c22
      c13;c23
```

Top of the page

pair

- **Definition:** a datatype holding a key and its associated value.
- **Built-in attributes:**
  - key (type = string): the key of the pair, i.e. the first element of the pair
  - value (type = string): the value of the pair, i.e. the second element of the pair
- **Remark:** pairs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a pair is defined by a key and a value sperarated by '::'.
- **Other declarations:** a pair can also be built from:
  - a point,
  - a map (in this case the first element of the pair is the list of all the keys of the map and the second element is the list of all the values of the map),
  - a list (in this case the two first element of the list are used to built the pair)

```
pair testPair <- "key"::56;
pair testPairPoint <- {3,5};            // 3::5
pair testPairList2 <- [6,7,8];          // 6::7
pair testPairMap <- [2::6,5::8,12::45];  // [12,5,2]::[45,8,6]
```

Top of the page

path

- **Definition:** a datatype representing a path (i.e. a polyline) linking two agents or geometries in a graph or more generally two points
- **Built-in attributes:**
  - source (type = point): the source point, i.e. the first point of the path
  - target (type = point): the target point, i.e. the last point of the path
  - graph (type = graph): the current topology (in the case it is a spatial graph), null otherwise
  - segments (type = list of geometry): the list of the geometries composing the path

- **Comments:** the path created between two agents/geometries or locations will strongly depend on the topology in which it is created.
- **Remark:** paths are particular cases of geometries. Thus they have also all the built-in attributes of the geometry datatype and can be used with every kind of operator or command admitting geometry.
- **Remark:** a path is **immutable** , i.e. it can not be modified after it is created.
- **Declaration:** paths are very barely defined litterally. We can nevertheless use the path unary operator on a list of points to build a path. Operators dedicated to the computation of paths (such as path_to or path_between) are often used to build a path.

```
path([{1,5},{2,9},{5,8}]) // a path from {1,5} to {5,8} through {2,9}

geometry rect <- rectangle(5);
geometry poly <- polygon([{10,20},{11,21},{10,21},{11,22}]);
path pa <- rect path_to poly;  // built a path between rect and poly, in the topolopy
                                              // of the current agent (i.e. a line in a& continuous
topology,
                                              // a path in a graph  in a graph topology )
a_topology path_between a_container_of_geometries // idem with an explicit topology and the
possiblity
                                                    // to have more than 2 geometries
                                                    // (the path is then built inscrementally)
```

Top of the page

point

- **Definition:** a datatype normally holding two positive float values. Represents the absolute coordinates of agents in the model.
- **Built-in attributes:**
  - x (type = float): coordinate of the point on the x-axis
  - y (type = float): coordinate of the point on the y-axis
- **Comments:** point coordinates should be positive, if a negative value is used in its declaration, the point is built with the absolute value.
- **Remark:** points are particular cases of geometries and containers. Thus they have also all the built-in attributes of both the geometry and the container datatypes and can be used with every kind of operator or command admitting geometry and container.
- **Litteral declaration:** two numbers, separated by a comma, enclosed in braces, like {12.3, 14.5}
- **Other declarations:** points can be built litteraly from a list, or from an integer or float value by using the point casting operator.

```
point ([12,123.45]) -> {12.0, 123.45}
point (2) -> {2.0, 2.0}
```

Top of the page

rgb

- **Definition:** a datatype that represents a color in the RGB space.
- **Built-in attributes:**
  - red(type = int): the red component of the color
  - green(type = int): the green component of the color
  - blue(type = int): the blue component of the color
  - darker(type = rgb): a new color that is a darker version of this color
  - brighter(type = rgb): a new color that is a brighter version of this color

- **Remark:** rgbs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** there exist lot of ways to declare a color. We use the rgb casting operator applied to:
  - a string. The allowed color names are the constants defined in the Color Java class, i.e.: black, blue, cyan, darkGray, lightGray, gray, green, magenta, orange, pink, red, white, yellow.
  - a list. The integer value associated to the three first elements of the list are used to define the three red (element 0 of the list), green (element 1 of the list) and blue (element 2 of the list) components of the color.
  - a map. The red, green, blue compoeenents take the value associated to the keys "r", "g", "b" in the map.
  - an integer < - the decimal integer is translated into a hexadecimal < - OxRRGGBB. The red (resp. green, blue) component of the color take the value RR (resp. GG, BB) translated in decimal.
- **Declaration:**

```
rgb testColor <- rgb('white');                 // rgb [255,255,255]
rgb test <- rgb([3,5,67]);                     // rgb [3,5,67]
rgb te <- rgb(340);                            // rgb [0,1,84]
rgb tete <- rgb(["r"::34, "g"::56, "b"::345]); // rgb [34,56,255]
```

Top of the page

species

- Definition: a generic datatype that represents a species
- **Built-in attributes:**
  - topology (type=topology): the topology is which lives the population of agents
- Comments: this datatype is actually a "meta-type". It allows to manipulate (in a rather limited fashion, however) the species themselves as any other values.
- Litteral declaration: the name of a declared species is already a litteral declaration of species.
- Other declarations: the species casting operator, or its variant called species_of can be applied to an agent in order to get its species.

Top of the page

Species names as types

Once a species has been declared in a model, it automatically becomes a datatype. This means that :

- It can be used to declare variables, parameters or constants,
- It can be used as an operand to commands or operators that require species parameters,
- It can be used as a casting operator (with the same capabilities as the built-in type agent)

In the simple following example, we create a set of "humans" and initialize a random "friendship network" among them. See how the name of the species, human, is used in the create command, as an argument to the list casting operator, and as the type of the variable named friend.

```
global {
    init {
        create human number: 10;
        ask list (human) {
              set friend <- one_of (list (human) - self);
        }
```

```
    }
}
entities {
    species human {
        human friend <- nil;
    }
}
```

Top of the page

topology

- **Definition:** a topology is basically on neighbourhoods, distance,... structures in which agents evolves. It is the environment or the context in which all these values are computed. It also provides the access to the spatial index shared by all the agents. And it maintains a (eventually dynamic) link with the 'environment' which is a geometrical border.
- **Built-in attributes:**
  - places(type = container): the collection of places (geometry) defined by this topology.
  - environment(type = geometry): the environment of this topology (i.e. the geometry that defines its boundaries)
- **Comments:** the attributes places depends on the kind of the considered topolopy. For continuous topologies, it is a list with their environment. For discrete topologies, it can be any of the container supporting the inclusion of geometries (list, graph, map, matrix)
- **Remark:** There exist various kinds of topology: continous topology and discrete topology (e.g. grid, graph...)
- **See also:** [Operators_14 Topology operators]
- **Declaration:** To create a topology, we can use the topology unary casting operator applied to:
  - an agent: returns a continuous topology built from the agent's geometry
  - a species name: returns the topology defined for this species population
  - a geometry: returns a continuous topology built on this geometry
  - a geometry container (list, map, shapefile): returns an half-discrete (with corresponding places), half-continuous topology (to compute distances...)
  - a geometry matrix (i.e. a grid): returns a grid topology which computes specifically neighbourhood and distances
  - a geometry graph: returns a graph topology which computes specifically neighbourhood and distances

More complex topologies can also be built using dedicated operators, e.g. to decompose a geometry...
Top of the page

Defining custom types

Sometimes, besides the species of agents that compose the model, it can be necessary to declare custom datatypes. Species serve this purpose as well, and can be seen as "classes" that can help to instantiate simple "objects". In the following example, we declare a new kind of "object", bottle, that lacks the skills habitually associated with agents (moving, visible, etc.), but can nevertheless group together attributes and behaviors within the same closure. The following example demonstrates how to create the species:

```
species bottle {
    float volume <- 0.0 max:1 min:0.0;
    bool is_empty function: {volume = 0.0};
```

```
    action fill {
        set volume <- 1;
    }
}
```

How to use this species to declare new bottles :

```
create bottle number: 1 {
    set volume <- 0.5;
}
```

And how to use bottles as any other agent in a species (a drinker owns a bottle; when he gets thirsty, it drinks a random quantity from it; when it is empty, it refills it):

```
species drinker {
    ...
    bottle my_bottle<- nil;
    float quantity <- rnd (100) / 100;
    bool thirsty <- false update: flip (0.1);
    ...
    action drink {
        if condition: ! bottle.is_empty {
              set bottle.volume <- max [bottle.volume - quantity, 0.0];
              set thirsty <- false;
        }
    }
    ...
    init {
          let cretated_bottle type:list of:bottle <- [];
          create bottle number: 1 return: created_bottle;
              set volume <- 0.5;
          }
          set my_bottle <- first(created_bottle);
    }
    ...
    reflex filling_bottle when: bottle.is_empty {
        ask  my_bottle {
              do fill;
        }
    }
    ...
    reflex drinking when: thirsty {
        do drink;
    }
}
```

Top of the page

_

Statements

Table of Contents

General syntax

A statement is a keyword, followed by specific attributes, some of them mandatory (in bold), some of them optional. One of the attribute names can be ommitted (the one that is omissible in the sequel). It has to be the first one.

```
statement_keyword expression1 attribute2: expression2 ... ;
or
statement_keyword attribute1: expression1 attribute2: expression2 ...;
```

If the statement encloses other statements, they are declared between curly brakets, as in:

```
statement_keyword1 expression1 attribute2: expression2... {
    statement_keyword2 expression1 attribute2: expression2...;
    statement_keyword3 expression1 attribute2: expression2...;
}
```

add

Attributes

- item (omissible): any expression
- **to** : an expression that evaluates to a container
- at: any expression

Definition

Allows to add, i.e. to insert, a new element in a container (a list, matrix, map, ...). The new element can be added either at the end of the container or at a particular position. **Incorrect use:** The addition of a new element at a position out of the bounds of the container will produce a warning and let the container unmodified.

```
  add expr to: expr_container;    // Add at the end
  add expr at: expr to: expr_container;   // Add at position expr
```

Note that the behavior and the type of the attributes depends on the specific kind of container.

- Case of a **list**

In the case of list, the expression in the attribute at: should be an integer.

```
let emptyList type: list <- [];
```

```
add 0 at: 0 to: emptyList ;   // emptyList now equals [0]
add 10 at: 0 to: emptyList ;   // emptyList now equals [10,0]
add 25 at: 2 to: emptyList ;   // emptyList now equals [10,0,20]
add 50 to: emptyList;    // emptyList now equals [10,0,20,50]
```

- Case of a **matrix**

This statement can not be used on matrix. Please refer to the statement `put` .

- Case of a **map**

As a map is basically a list of pairs key::value , we can also use the add statement on it. It is important to note that the behavior of the statement is slightly different, in particular in the use of the at attribute.

```
let emptyMap type: map <- [];
add "val1" at: "x" to: emptyMap;   // emptyList now equals [x::val1]
```

If the at: attribute is ommitted, a pair null::expr_item will be added to the map. An important exception is the case where the is a pair expression: in this case the pair is added.

```

add "val2" to: emptyMap;   // emptyList now equals [null::val2, x::val1]
add 5::"val4" to: emptyMap;   // emptyList now equals [null::val2, 5::val4, x::val1]
```

Notice that, as the key should be unique, the addition of an item at an existing position (i.e. existing key) will only modify the value associated with the given key.

```

add "val3" at: "x" to: emptyMap;   // emptyList now equals [null::value2, 5::val4, x::val3]
```

Top of the page

ask

Attributes

- **target** (omissible): an expression that evaluates to an agent or a list of agents
- as: an expression that evaluates to a species

Definition

Allows an agent, the sender agent (that can be the world agent), to ask another (or other) agent(s) to perform a set of statements. It obeys the following syntax, where the target attribute denotes the receiver agent(s):

```
ask receiver_agent(s) {
    [statements]
}
```

If the value of the target attribute is nil or empty, the statement is ignored. The species of the receiver agents must be known in advance for this statement to compile. If not, it is possible to cast them using the as attribute, like :

```
ask receiver_agent(s) as: a_species_expression {
    [statement_set]
}
```

Alternative forms for this casting are :

- if there is only a single receiver agent:

```
ask species_name (receiver_agent) {
    [statement_set]
}
```

- if receiver_agent(s) is a list of agents:

```
ask receiver_agents of_species species_name {
    [statement_set]
}
```

Any statement can be declared in the block statements . All the statements will be evaluated in the context of the receiver agent(s), as if they were defined in their species, which means that an expression like *self* will represent the receiver agent and not the sender. If the sender needs to refer to itself, some of its own attributes (or temporary variables) within the block statements , it has to use the keyword *myself* .

```
species animal {
    float energy <- rnd (1000) min: 0.0 {
    reflex when: energy > 500 { // executed when the energy is above the given threshold
        let others type: list of: animal <- (self neighbours_at 5) of_species animal; // find all
the neighbouring animals in a radius of 5 meters
        let shared_energy type: float  <- (energy - 500) / length (others); // compute the amount
of energy to share with each of them
        ask others { // no need to cast, since others has already been filtered to only include
animals
              if (energy < 500) { // refers to the energy of each animal in others
                  set energy <- energy + myself.shared_energy; // increases the energy of each
animal
                  set myself.energy <- myself.energy - myself.shared_energy; // decreases the
energy of the sender
              }
        }
    }
}
```

Notice

If the target is an addition of list like "target = (list speciesA) + (list speciesB)", the temporary built list will use the default cast from the first list and won't add the second list as the elements are from a different type. Top of the page

capture

Attributes

- **target** (omissible): an expression that is evaluated as an agent or a list of the agent to be captured.
- **as** : the species that the captured agent(s) will become, this is a micro-species of the calling agent's species.
- returns: a list of the newly captured agent(s).

Definition

Allows an agent to capture other agent(s) as its micro-agent(s). The preliminary for an agent A to capture an agent B as its micro-agent is that the A's species must defined a micro-species which is a sub-species of B's species (cf. [Species14 Nesting species]).

```
species B {
...
}
species A {
...
  species C parent: B {
  ...
  }
...
}
```

- To capture all "B" agents as "C" agents, we can ask an "A" agent to execute the following statement:

```
capture list(A) as: C;
```

Deprecated writing:

```
capture target: list (A) as: C;
```

See also the release statement . Top of the page

create

Attributes

- species (omissible): an expression that evaluates to a species
- number: an expression that evaluates to an int
- from: an expression that evaluates to a localized entity, a list of localized entities or a string
- with: an expression that evaluates to a map
- type: an expression that evaluates to a string
- size: an expression that evaluates to a float
- return: a temporary variable name

Definition

Allows an agent to create *number* agents of species *species* , to create agents of species *species* from a shapefile or to create agents of species *species* from one or several localized entities (discretization of the localized entity geometries). Its simple syntax is:

- To create *an_int* agents of species *a_species* :

```
create a_species number: an_int;
```

Deprecated writing:

```
create species: a_species number: an_int;
```

If *number* equals 0 or species is not a species, the statement is ignored.

- To create agents of species *a_species* (with two attributes type and nature) from a shapefile *the_shapefile* while reading attributes 'TYPE_OCC' and 'NATURE' of the shapefile:

```
create a_species from: the_shapefile with: [type:: 'TYPE_OCC', nature::'NATURE'];
```

One agent will be created by object contained in the shapefile. In this example, we assume that for the species *a_species* , two variables *type* and *nature* are declared and that their type corresponds to the types of the shapefile attributes.

- To create agents of species *a_species* by discretizing the geometry of one or several localized entities:

```
create a_species from: [agentA, agentB, agentC];
```

Two types of discretization exist:

- **'Triangles'** : default discretization. The agent geometries are decomposed into triangles; for each triangle, an agent is created. If a size is declared by attribute *size* , the geometries are first decomposed into squares of size *size* , then each square is decomposed into triangles.

```
create a_species from: [agentA, agentB, agentC] type: 'Triangles' size: 10.0;
```

- **'Squares'** : agent geometries are decomposed into squares of size *size* ; for each square, an agent is created:

```
create a_species from: [agentA, agentB, agentC] type: 'Squares' size: 10.0;
```

The agents created are initialized following the rules of their species. If one wants to refer to them after the statement is executed, the returns keyword has to be defined: the agents created will then be referred to by the temporary variable it declares. For instance, the following statement creates 0 to 4 agents of the same species as the sender, and puts them in the temporary variable children for later use.

```
create species (self) number: rnd (4) returns: children;
ask children {
    ...
}
```

If one wants to specify a special initialization sequence for the agents created, create provides the same possibilities as ask . This extended syntax is:

```
create a_species number: an_int {
    [statements]
}
```

The same rules as in ask apply. The only difference is that, for the agents created, the assignments of variables will bypass the initialization defined in species. For instance:

```
create species(self) number: rnd (4) returns: children {
    set location <- myself.location + {rnd (2), rnd (2)}; // tells the children to be initially located close to me
    set parent <- myself; // tells the children that their parent is me (provided the variable parent is declared in this species)
}
```

Top of the page

do

Attributes

- **action** (omissible): the name of an action or a primitive
- with: a map expression

Enclosed tags

- arg name : specify the arguments expected by the action/primitive to execute.

Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this [Built_in_14 page]; for the list of primitives defined by the different skills, see this [Skills_14 page]. Finally, see this [Species_14 page] to know how to declare custom actions. The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

```
do name_of_action_or_primitive;
```

Deprecated writing:

```
do action: name_of_action_or_primitive;
```

In case the result of the action needs to be made available to the agent, the returns keyword has to be defined; the result will then be referred to by the temporary variable declared in this attribute:

```
let result <- self name_of_action_or_primitive [];
```

Deprecated writing:

```
let result <- name_of_action_or_primitive(self, []);
```

In case the action expects one or more arguments to be passed, they are defined by using facets, enclosed tags or a map. We can have the three following notations:

```
do name_of_action_or_primitive arg1: expression1 arg2: expression2;
```

Deprecated writing:

```
do name_of_action_or_primitive with: [arg1::expression1, arg2::expression2];
or
do name_of_action_or_primitive {
    arg arg1 value: expression1;
    arg arg2 value: expression2;
    ...
}
```

In the case of an action returning a value, we can only use facets or a map as follows:

```
let result <- self name_of_action_or_primitive [arg1::expression1, arg2::expression2];
```

Deprecated writing:

```
do name_of_action_or_primitive arg1: expression1 arg2: expression2 returns: result;
or
let result <- name_of_action_or_primitive(self, [arg1::expression1, arg2::expression2]);
```

Top of the page

error

Attributes

- **message** (omissible): string, the message to display. Modelers can add some formatting characters to the message (carriage returns, tabs, or Unicode characters), which will be used accordingly in the error dialog.

Definition

makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

```
error 'This is an error raised by ' + self;
```

Top of the page

if

Attributes

- **condition** (omissible): a boolean expression

Following tags

- else : encloses alternative statements

Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true. The generic syntax is:

```
if bool_expr {
    [statements]
}
```

Deprecated writing:

```
if condition: bool_expr { [statements] }
```

Optionally, the statements to execute when the condition evaluates to false can be defined in a following statement else . The syntax then becomes:

```
if bool_expr {
    [statements]
}
else {
```

```
    [statements]
}
```

ifs and elses can be imbricated as needed. For instance:

```
if bool_expr {
    [statements]
}
else if bool_expr2 {
    [statements]
}
else {
        [statements]
}
```

Top of the page

let

Attributes

- name (omissible): the name of the temporary variable
- type: the datatype of the temporary variable
- **value** : an expression

Definition

Allows the agent to declare a temporary variable, local to the scope in which it is defined. The naming rules follow those of the variable declarations. In addition, a temporary variable cannot be declared twice in the same scope. The generic syntax is:

```
let temp_var1 type: a_datatype <- an_expression;
```

If the datatype of the variable is not specified, it is inferred from that of the expression (which can be enforced using casting operators if necessary). After it has been declared this way, a temporary variable can be used like regular variables (for instance, the set statement should be used to assign it a new value within the same scope). Top of the page

loop

Attributes

- var (omissible): a temporary variable name
- times: an int expression
- while: a boolean expression
- over: a list, point, matrix or map expression
- from: an int expression
- to: an int expression

- step: an int expression

Definition

Allows the agent to perform the same set of statements either a fixed number of times, or while a condition is true, or by progressing in a collection of elements or along an interval of integers. The basic syntax for each of these usages are:

```
loop times: an_int_expression {
    [statements]
}
```

Or:

```
loop while: a_bool_expression {
    [statements]
}
```

Or:

```
loop a_temp_var over: a_list_expression {
    [statements]
}
```

Or:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 {
    [statements]
}
loop a_temp_var from: int_expression_1 to: int_expression_2 step: int_expression3{
    [statements]
}
```

In these latter two cases, the var attribute designates the name of a temporary variable, whose scope is the loop, and that takes, in turn, the value of each of the element of the list (or each value in the interval). For example, in the first instance of the "loop over" syntax :

```
let a type: int <- 0;
loop i over: [10, 20, 30] {
    set a <- a + i;
} // a now equals 60
```

The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to attributes take a a integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the interval. The step is assumed equal to 1.

```
let the_list <-list (species_of (self)) {
loop i from: 0 to: length (the_list){
    ask target: the_list at i {
        ...
    }
} // every  agent of the list is asked to do something
```

Be aware that there are no prevention of infinite loops. As a consequence, open loops should be used with caution, as one agent may block the execution of the whole model. Top of the page

put

Attributes

- **in** : an expression that evaluates to a container
- item (omissible): any expression
- at: any expression
- key: any expression
- all: any expression

Definition

Allows the agent to replace a value in a container at a given position (in a list or a map) or for a given key (in a map). The allowed parameters configurations are the following:

```
put expr at: expr in: expr_container;
put all: expr in: expr_container;
```

Note that the behavior and the type of the attributes depends on the specific kind of container:

- In the case of a **list** , the position should an integer in the bound of the list. The attribute all is used to replace all the elements of the list by the given value.

```
let testList type: list <- [1,2,3,4,5];   // testList now contains [1,2,3,4,5]
put -10 at: 1 in: testList;    // testList now contains [1,-10,3,4,5]
put all: 10 in: testList;    // testList now contains [10,10,10,10,10]
```

- In the case of a **matrix** , the position should be a point in the bound of the matrix. The attribute all is used to replace all the elements of the matrix by the given value.

```
let testMat type: matrix <- [[0,1],[2,3]];  //testMat now contains [[0,1],[2,3]]
put -10 at: {1,1} in: testMat;  //testMat now contains [[-10,1],[2,3]]
put all: 10 in: testMat;  //testMat now contains [[10,10],[10,10]]
```

- In the case of a **map** , the position should be one of the key values of the map. Notice that if the given key value does not exist in the map, the given pair key::value will be added to the map. The attribute all is used to replace the value of all the pairs of the map.

```
let testMap type: map <- ["x"::4,"y"::7]; //testMap now contains ["x"::4,"y"::7]
put -10 key: "y" in: testMap;    //testMap now contains ["x"::4,"y"::10]
put -20 key: "z" in: testMap;    //testMap now contains ["x"::4,"y"::7, "z"::-20]
put all: -30 in: testMap;    //testMap now contains ["x"::-30,"y"::-30, "z"::-30]
```

Top of the page

release

Attributes

- **target** (omissible): an expression that is evaluated as an agent or a list of the agent to be released.
- returns: a list of the newly released agent(s).

Definition

Allows an agent to release its micro-agent(s). The preliminary for an agent to release its micro-agents is that species of these micro-agents are sub-species of other species (cf. [Species14 Nesting species]). The released won't be micro-agents of the calling agent anymore. Being released from a macro-agent, the micro-agents will change their species and host (macro-agent) .

```
species A {
}
species B {
  species C parent: A {
  }
  species D {
  }
}
```

Agents of "C" species can be released from a "B" agent to become agents of A species. Agents of "D" species cannot be released from the "A" agent because species "D" has no parent species.

- To release all "C" agents from a "B" agent, we can ask the "C" agent to execute the following statement:

```
release target: list (C);
```

The "C" agent will change to "A" agent. The won't consider "B" agent as their macro-agent (host) anymore. Their host (macro-agent) will the be the host (macro-agent) of the "B" agent. See also the capture statement . Top of the page

remove

Attributes

- **from** : an expression that evaluates to a container
- item (omissible): any expression
- index: any expression
- key: any expression
- all: any expression

Definition

Allows the agent to remove an element from a container (a list, matrix, map...). This statement should be used in the following ways, depending on the kind of container used and the expected action on it:

```
remove expr from: expr_container
remove index: expr from: expr_container
remove key: expr from: expr_container
remove all: expr from: expr_container
```

- Case of **list** .

In the case of list, the attribut item: is used to remove the first occurence of a given expression, whereas all is used to remove all the occurences of the given expression.

```
let testList type: list <- [3,2,1,2,3];  //testList now contains [3,2,1,2,3]
remove 2 from: listTest;  // testList now contains [3,1,2,3]
remove all: 3 from: listTest;  // testList now contains [1,2]
remove index: 1 from: listTest;
```

- Case of **matrix**

This statement can not be used on **matrix** .

- In the case of **map**

In the case of map, the attribute key: is used to remove the pair identified by the given key.

```
let mapTest type: map <- ["x"::5, "y"::7];    // mapTest now contains ["x"::5, "y"::7]
remove key: "x" from: mapTest;   // mapTest now contains ["y"::7]
```

Top of the page

return

Attributes

- **value** (omissible): an expression

Definition

Allows to specify which value to return from the evaluation of the surrounding statement. Usually used within the declaration of an action. Contrary to other languages, using return does not stop the evaluation of the surrounding statement (for instance, a loop). It simply indicates what value to return: if it is inside a loop, then, only the last evaluation of return will be returned. Example:

```
action foo {
    return 'foo';
}
...
reflex {
    let foo_result type: string <- self.foo [];
    do write {
        arg message <- foo_result;
    }
}
// the agent will print  foo  on the console at each step
```

In the specific case one wants an agent to ask another agent to execute a statement with a return, it can be done similarly to:

```
Species A:
action foo_different {
    return 'foo_not_same';
}
---
Species B
reflex writing{
    let temp type: string <- some_agent_A.foo_different [];
    write temp;
}
// the agent will print  foo_not_same  on the console at each step
```

Top of the page

save

Attributes

- **to** : an expression that evaluates to an string
- species: an expression that evaluates to a species
- type: an expression that evaluates to an string

Definition

Allows to save the localized entities of species *species* into a particular kind of file (shapefile, text or csv...). The type can be "shp", "text" or "csv". Its simple syntax is:

```
save a_species to: the_shapefile type: a_type_file;
```

e.g for a .shp file:

```
save a_species to: (path + "shapefile.shp") type: "shp" with: [attribut1::ATT1];
```

It can be use at the end of the init or in a user command. Do not use it in experiment Top of the page

set

Attributes

- an expression that either returns a variable or an element of a composite type
- **value** : an expression

Definition

Allows the agent to assign a value to a variable. See this section to know how to access variables. Examples:

```
set my_var <- expression;
set temp_var <- expression;
set global_var <- expression;
```

The variable assigned can be accessed in the value atribute. In that case, it represents the value of the variable before it has been modified. Examples (with temporary variables):

```
let my_int type: int <- 1000;
set my_int <- my_int + 1; // my_int now equals 1001
```

Top of the page

switch

Attributes

- **value** (omissible): an expression

Embedded tags

- match value {...}
- match_one list_values {... }
- match_between [value1, value2] {...}
- default {...}

Definition

The "switch... match" statement is a powerful replacement for imbricated "if ... else ..." constructs. All the blocks that match are executed in the order they are defined. The block prefixed by default is executed only if none have matched (otherwise it is not). Examples:

```
switch an_expression {
    match value1 {...}
    match_one [value1, value2, value3] {...}
    match_between [value1, value2] {...}
    default {...}
}
```

Example:

```
switch 3 {
  match 1 {write "Match 1"; }
  match 2  {write "Match 2"; }
  match 3  {write "Match 3"; }
  match_one [4,4,6,3,7]  {write "Match one_of"; }
  match_between [2, 4] {write "Match between"; }
  default {write "Match Default"; }
}
```

Top of the page

write

Attributes

- **message** (omissible): string, the message to display. Modelers can add some formatting characters to the message (carriage returns, tabs, or Unicode characters), which will be used accordingly in the console.

Definition

makes the agent output an arbitrary message in the console.

```
write 'This is a message from ' + self;
```

Top of the page

Operators

This file is automatically generated from java files. Do Not Edit It.

Table of Contents

Definition

An operator performs a function on one, two, or three operands. An operator that only requires one operand is called a unary operator. An operator that requires two operands is a binary operator. And finally, a ternary operator is one that requires three operands. The GAML programming language has only one ternary operator, ? :, which is a short-hand if-else statement. Unary operators are written using aprefix parenthesized notation. Prefix notation means that the operator appears before its operand. Note that unary expressions should always been parenthesized:

```
unary_operator (operand)
```

Most of binary operators can use two notations:

- the fonctional notation, which used a parenthesized notation around the operands (this notation cannot be used with arithmetic and relational operators such as: +, -, /, * , ^, =, !=, < , > , > =, < =...)
- the infix notation, which means that the operator appears between its operands

```
binary_operator(op1, op2)
Or
op1 binary_operator op2
```

The ternary operator is also infix; each component of the operator appears between operands:

```
op1 ? op2 : op3
```

In addition to performing operations, operators are functional, i.e. they return a value. The return value and its type depend on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers - the result of the arithmetic operation. Moreover, operators are strictly functional, i.e. they have no side effects on their operands. For instance, the shuffle operator, which randomizes the positions of elements in a list, does not modify its list operand but returns a new shuffled list. Top of the page

Operators by categories

Casting operators

- agent , as , as_int , as_matrix , bool , container , float , geometry , graph , int , is , list , pair , path , point , rgb , species , species_of , string , to_gaml , to_java , topology , unknown ,

Comparison operators

- != , < , <= , <> , = , > , >= ,

Containers-related operators

- :: , accumulate , among , any , as_map , at , collate , collect , contains , contains_all , copy_between , count , empty , first_with , grid_at , group_by , index_of , last_with , map , matrix , max , max_of , min , min_of , mul , of_generic_species , of_species , one_of , product , remove_duplicates , select , sort , sort_by , sum , where , with_max_of , with_min_of ,

Files-related operators

- file , folder , get , image , is_image , is_properties , is_shape , is_text , new_folder , properties , read , shapefile , text , write ,

Graphs-related operators

- add_edge , agent_from_geometry , as_distance_graph , as_edge_graph , as_intersection_graph , contains_edge , contains_vertex , degree_of , directed , generate_barabasi_albert , generate_watts_strogatz , in_degree_of , in_edges_of , load_graph_from_dgs , load_graph_from_dgs_old , load_graph_from_dot , load_graph_from_edge , load_graph_from_gexf , load_graph_from_graphml , load_graph_from_lgl , load_graph_from_ncol , load_graph_from_pajek , load_graph_from_tlp , out_degree_of , out_edges_of , predecessors_of , remove_node_from , rewire_n , set_verbose , source_of , successors_of , target_of , undirected , weight_of , with_optimizer_type , with_weights ,

Logical operators

- : , ! , ? , and , not , or ,

Mathematics operators

- / , ^ , ** , abs , acos , asin , atan , ceil , cos , div , even , exp , fact , floor , ln , mod , round , sin , sqrt , tan , tanh , with_precision ,

Matrix-related operators

- column_at , columns_list , row_at , rows_list ,

Random operators

- binomial , flip , gauss , poisson , rnd , shuffle , TGauss , truncated_gauss ,

Spatial operators

- - , * , + , <--: , add_point , add_z , add_z_pt , agent_closest_to , agents_at_distance , agents_inside , agents_overlapping , any_location_in , any_point_in , around , as_4_grid , as_grid , at_distance , at_location , buffer , circle , clean , closest_points_with , closest_to , cone , convex_hull , covered_by , covers , crosses , direction_between , direction_to , disjoint_from , distance_between , distance_to , enlarged_by , farthest_point_to , inside , inter , intersection , intersects , line , link , masked_by , neighbours_at , neighbours_of , norm , overlapping , overlaps , partially_overlaps , path_between , path_to , points_at , polygon , polyline , rectangle , reduced_by , rotated_by , scaled_by , simple_clustering_by_distance , simple_clustering_by_envelope_distance , simplification , skeletonize , solid , split_at , split_lines , square , touches , towards , transformed_by , translated_by , translated_to , triangle , triangulate , union , without_holes ,

Statistical operators

- corR , frequency_of , geometric_mean , harmonic_mean , mean , mean_deviation , meanR , median , R_compute , standard_deviation , variance ,

Strings-related operators

- as_date , as_time , contains_any , first , in , is_number , last , last_index_of , length , reverse , split_with , tokenize ,

System

- . , copy , dead , eval_gaml , evaluate_with , every , of , user_input ,

Operators

-

- Possible use:
  - OP(int) --- > int
  - OP(float) --- > float
  - shape OP list of shapes --- > shape
  - float OP int --- > float
  - rgb OP int --- > rgb
  - list OP any --- > list
  - shape OP float --- > shape
  - point OP float --- > point
  - point OP point --- > point
  - shape OP species --- > shape
  - list OP list --- > list
  - rgb OP rgb --- > rgb
  - int OP int --- > int
  - point OP int --- > point

- int OP float --- > float
- shape OP shape --- > shape
- float OP float --- > float

- Result: a new color resulting from the substraction of each component of the color with the right operandreturns a new list in which all the elements of the right operand have been removed from the left onea new color resulting from the substraction of the two operands, component by componentthe difference of the two operands
- Comment: The behavior of the operator depends on the type of the operands.
- Special cases:
  - if the right-operand is a list of points, geometries or agents, returns the geometry resulting from the difference between the left-geometry and all of the right-geometries
  - if the right operand is an object of any type (except list), - returns a copie of the left operand without this object
  - if the left-hand operand is a geometry and the rigth-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) reduced by the right-hand operand distance
  - if left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the difference of the operand coordinate with this number.
  - if both operands are points, returns their difference.
  - if the right-operand is a species, returns the geometry resulting from the difference between the left-geometry and all of geometries all agents of the right-species
  - when it is used as an unary operator, - returns the opposite or the operand.
  - if the right operand is empty or nil, - returns the left operand
  - if both operands are numbers, performs a normal arithmetic difference and returns a float if one of them is a float.
  - if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries
- See also: + ,

```
geom1 - [geom2, geom3, geom4] --: a geometry corresponding to geom1 - (geom2 + geom3 + geom4)
1.0 - 1 --: 0.0
rgb([255, 128, 32]) - 3     --:      rgb([252,125,29])
[1,2,3,4,5,6] - 2         --:     [1,3,4,5,6]
[1,2,3,4,5,6] - 0         --:         [1,2,3,4,5,6]
shape - 5 --: returns a geometry corresponding to the geometry of the agent applying the operator
reduced by a distance of 5
{1, 2} - 4.5     --:     {-3.5, -2.5}
{1, 2} - {4, 5}     --:     {-3.0;-3.0}
geom1 - speciesA --: a geometry corresponding to geom1 - (the geometry of all agents of species
speciesA)
- (-56)     --:     56
[1,2,3,4,5,6] - [2,4,9]     --:     [1,3,5,6]
[1,2,3,4,5,6] - [0,8]         --:         [1,2,3,4,5,6]
rgb([255, 128, 32]) - rgb('red')       --:      rgb([0,128,32])
1 - 1     --:     0
{1, 2} - 4     --:     {-3.0;-2.0}
geom1 - geom2 --: a geometry corresponding to difference between geom1 and geom2
```

Top of the page

- Possible use:
  - any OP any --- > any

- See also: ? ,

Top of the page

::

- Possible use:
  - any OP any --- > pair
- Result: produces a new pair combining the left and the right operands

Top of the page

!

- Possible use:
  - OP(bool) --- > bool
- Result: opposite boolean value.
- Special cases:
  - if the parameter is not boolean, it is casted to a boolean value.
- See also: bool ,

```
! (true)         --:         false
```

Top of the page

!=

- Possible use:
  - float OP float --- > bool
  - any OP any --- > bool
- Result: true if both operands are different, false otherwise
- Comment: this operator will return false if the two operands are identical (i.e., the same object) or equal. Comparisons between nil values are permitted.
- See also: = ,

```
4.5 = 4.7      --:  false
[2,3] != [2,3] --: false
[2,4] != [2,3] --: true
```

Top of the page

?

- Possible use:
  - bool OP any expression --- > any
- Result: if the left-hand operand evaluates to true, returns the value of the left-hand operand of the :, otherwise that of the right-hand operand of the :
- Comment: These functional tests can be combined together.
- See also: : ,

```
[10, 19, 43, 12, 7, 22] collect ((each > 20) ? 'above' : 'below')    --:  ['below', 'below', 'above', 'below', 'below', 'above']
```

```
set color value:(food > 5) ? 'red' : ((food >= 2)? 'blue' : 'green');
```

Top of the page

/

- Possible use:
  - int OP int --- > float
  - point OP int --- > point
  - point OP float --- > point
  - float OP int --- > float
  - int OP float --- > float
  - rgb OP int --- > rgb
  - float OP float --- > float
  - rgb OP float --- > rgb
- Result: a float, equal to the division of the left-hand operand by the rigth-hand operand.a new color resulting from the division of each component of the color by the right operanda new color resulting from the division of each component of the color by the right operand. The result on each component is then truncated.
- Special cases:
  - if the right-hand operand is equal to zero, raises a "Division by zero" exception
  - if the left-hand operator is a point and the right-hand a number, returns a point with coordinates divided by the number
- See also: * ,

```
{2,5} / 4    --:  {0.5;1.25}
rgb([255, 128, 32]) / 2     --:      rgb([127,64,16])
rgb([255, 128, 32]) / 2.5     --:      rgb([102,51,13])
```

Top of the page

.

- Possible use:
  - agent OP any expression --- > any
- Result: returns an evaluation of the expresion (right-hand operand) in the scope the given agent.
- Special cases:
  - if the agent is nil or dead, throws an exception

```
agent.location         --:     returns the location of the agent
```

Top of the page

^

- Possible use:
  - int OP int --- > int
  - float OP int --- > float
  - int OP float --- > float
  - float OP float --- > float
- Result: the left-hand operand raised to the power of the right-hand operand.
- Special cases:

    - if the right-hand operand is equal to 0, returns 1
    - if it is equal to 1, returns the left-hand operand.
- See also: * , sqrt ,

Top of the page

`*`

- Possible use:
    - point OP int --- > point
    - float OP float --- > float
    - float OP int --- > float
    - int OP int --- > int
    - int OP float --- > float
    - shape OP float --- > shape
    - rgb OP int --- > rgb
    - point OP float --- > point
    - point OP point --- > float
- Result: the product of the two operandsa new color resulting from the product of each component of the color with the right operand
- Special cases:
    - if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number
    - if both operands are int, returns the product as an int
    - if the left-hand operand is a geometry and the rigth-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) scaled by the right-hand operand coefficient
    - if both operands are points, returns their scalar product
- See also: / ,

```
{2,5} * 4     --: {8.0; 20.0}
shape * 2 --: returns a geometry corresponding to the geometry of the agent applying the operator
scaled by a coefficient of 2
rgb([255, 128, 32]) * 2     --:      rgb([255,255,64])
{2,5} * {4.5, 5}     --:  34.0
```

Top of the page

`**`

Same signification as ^ operator.

Top of the page

+

- Possible use:
    - shape OP float --- > shape
    - shape OP shape --- > shape
    - shape OP map --- > shape
    - string OP any --- > string
    - list OP list --- > list

- rgb OP int --- > rgb
- float OP int --- > float
- string OP string --- > string
- point OP int --- > point
- point OP float --- > point
- float OP float --- > float
- int OP float --- > float
- int OP int --- > int
- point OP point --- > point
- list OP any --- > list
- rgb OP rgb --- > rgb

- Result: returns a new list containing all the elements of both operandsa new color resulting from the sum of each component of the color with the right operandthe sum, union or concatenation of the two operands.a new color resulting from the sum of the two operands, component by component
- Comment: + is only defined with a list as left operand
- Special cases:
  - if the left-hand operand is a geometry and the rigth-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the right-hand operand distance
  - if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the union between both geometries
  - if the left-hand operand is a geometry and the rigth-hand operand a map (with [distance::float, quadrantSegments:: int (the number of line segments used to represent a quadrant of a circle), endCapStyle::int (1: (default) a semi-circle, 2: a straight line perpendicular to the end segment, 3: a half-square)]), returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged considering the right-hand operand parameters
  - if the right operand is nil, + returns the left operand
  - if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one beind casted into a string
  - if left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the sum of the operand coordinate with this number.
  - if both operands are numbers (float or int), performs a normal arithmetic sum and returns a float if one of them is a float.
  - if both operands are points, returns their sum.
  - if the right operand is an object of any type (except list), + returns a copie of the left operand with this object
- See also: - , - ,

```
shape + 5 --: returns a geometry corresponding to the geometry of the agent applying the operator
enlarged by a distance of 5
geom1 + geom2 --: a geometry corresponding to union between geom1 and geom2
shape + [distance::5.0, quadrantSegments::4, endCapStyle:: 2] --: returns a geometry
corresponding to the geometry of the agent applying the operator enlarged by a distance of 5,
with 4 segments to represent a quadrant of a circle and a straight line perpendicular to the end
segment
"hello " + 12     --:     "hello 12"
[1,2,3,4,5,6] + [2,4,9]     --:     [1,2,3,4,5,6,2,4,9]
[1,2,3,4,5,6] + [0,8]         --:     [1,2,3,4,5,6,0,8]
rgb([255, 128, 32]) + 3     --:      rgb([255,131,35])
1.0 + 1 --: 2.0
{1, 2} + 4     --:     {5.0;6.0}
{1, 2} + 4.5     --:     {5.5, 6.5}
1 + 1     --:     2
{1, 2} + {4, 5}     --:     {5.0;7.0}
```

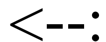
```
[1,2,3,4,5,6] + 2          --:     [1,2,3,4,5,6,2]
[1,2,3,4,5,6] + 0          --:        [1,2,3,4,5,6,0]
rgb([255, 128, 32]) + rgb('red')     --:      rgb([255,128,32])
```

Top of the page

- Possible use:
  - int OP float --- > bool
  - string OP string --- > bool
  - point OP point --- > bool
  - int OP int --- > bool
  - float OP float --- > bool
  - float OP int --- > bool
- Result: true if the left-hand operand is less than the right-hand operand, false otherwise.
- Special cases:
  - if the operands are strings, a lexicographic comparison is performed
  - if both operands are points, returns true if only if left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.
  - if one of the operands is nil, returns false

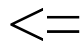
```
3 < 2.5  --: false
abc < aeb  --: true
{5,7} < {4,6}  --: false
{5,7} < {4,8}  --: false
3 < 7  --:  true
3.5 < 7.6  --: true
3.5 < 7  --: true
```

Top of the page



Same signification as disjoint_from operator.

Top of the page

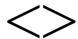


- Possible use:
  - string OP string --- > bool
  - point OP point --- > bool
  - float OP float --- > bool
  - int OP float --- > bool
  - int OP int --- > bool
  - float OP int --- > bool
- Result: true if the left-hand operand is less or equal than the right-hand operand, false otherwise.
- Special cases:
  - if the operands are strings, a lexicographic comparison is performed

- if both operands are points, returns true if only if left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.
- if one of the operands is nil, returns false

```
abc <= aeb  --: true
{5,7} <= {4,6}  --: false
{5,7} <= {4,8}  --: false
3.5 <= 3.5  --: true
3 <= 2.5  --: false
3 <= 7  --:  true
7.0 <= 7  --: true
```

Top of the page



Same signification as != operator.

Top of the page

- Possible use:
  - float OP float --- > bool
  - any OP any --- > bool
- Result: true if both operands are equal, false otherwise
- Comment: this operator will return true if the two operands are identical (i.e., the same object) or equal. Comparisons between nil values are permitted.
- See also: != ,

```
3 = 3         --: true
4.5 = 4.7       --:  false
3.0 = 3       --:  true
[2,3] = [2,3] --: true
```

Top of the page

- Possible use:
  - int OP int --- > bool
  - float OP int --- > bool
  - string OP string --- > bool
  - float OP float --- > bool
  - point OP point --- > bool
  - int OP float --- > bool
- Result: true if the left-hand operand is greater than the right-hand operand, false otherwise.
- Special cases:
  - if one of the operands is nil, returns false
  - if the operands are strings, a lexicographic comparison is performed

- if both operands are points, returns true if only if left component (x) of the left operand if greater than x of the right one and if the right component (y) of the left operand is greater than y of the right one.

```
3 > 7  --:  false
3.5 > 7  --: false
abc > aeb  --: false
3.5 > 7.6  --: false
{5,7} > {4,6}  --: true
{5,7} > {4,8}  --: false
3 > 2.5  --: true
```

Top of the page

>=

- Possible use:
  - point OP point --- > bool
  - string OP string --- > bool
  - float OP float --- > bool
  - int OP int --- > bool
  - float OP int --- > bool
  - int OP float --- > bool
- Result: true if the left-hand operand is greater or equal than the right-hand operand, false otherwise.
- Special cases:
  - if both operands are points, returns true if only if left component (x) of the left operand if greater than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.
  - if the operands are strings, a lexicographic comparison is performed
  - if one of the operands is nil, returns false

```
{5,7} >= {4,6}  --: true
{5,7} >= {4,8}  --: false
abc >= aeb  --: false
abc >= abc  --: true
3.5 >= 3.5  --: true
3 >= 7  --:  false
3.5 >= 7  --: false
3 >= 2.5  --: true
```

Top of the page

abs

- Possible use:
  - OP(int) --- > int
  - OP(float) --- > float
- Result: the absolute value of the operand (so a positive int or float depending on the type of the operand).

```
abs (200 * -1 + 0.5) --: 200.5
```

Top of the page

accumulate

- Possible use:
  - container OP any expression --- > list
- Result: returns a new flat list, in which each element is the evaluation of the right-hand operand. If this evaluation returns a list, the elements of this result are added directly to the list returned
- Comment: accumulate is dedicated to the application of a same computation on each element of a container (and returns a list) In the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is nil, accumulate returns an empty list
- See also: collect ,

```
[a1,a2,a3] accumulate (each neighbours_at 10)          --:     a flat list of all the neighbours
of these three agents
[1,2,4] accumulate ([2,4])          --:     [2,4,2,4,2,4]
```

Top of the page

acos

- Possible use:
  - OP(float) --- > float
  - OP(int) --- > float
- Result: the arccos of the operand (which has to be expressed in decimal degrees).
- See also: asin , atan ,

```
acos (90)     --:     0
```

Top of the page

add_edge

- Possible use:
  - graph OP pair --- > graph
- Result: add an edge between source vertex and the target vertex
- Comment: If the edge alrzdy exists the graph is unchanged
- See also: [#],

```
set graph <- graph add_edge (source::target);
```

Top of the page

add_point

- Possible use:
  - shape OP point --- > shape
- Result: A geometry resulting from the adding of a right-point (coordinate) to the right-geometry

```
square(5) add_point {10,10} --: returns a hexagon
```

Top of the page

add_z

- Possible use:
  - point OP float --- > point
  - shape OP float --- > shape
- Result: add_z
- Comment: Return a geometry with a z valueThe add_z operator set the z value of the whole shape.For each point of the cell the same z value is set.
- See also: add_z_pt ,

```
set shape <- shape add_z rnd(100);
```

Top of the page

add_z_pt

- Possible use:
  - shape OP point --- > shape
- Result: add_z_pt
- Comment: Return a geometry with a z value
- See also: add_z ,

```
loop i from: 0 to: length(shape.points) - 1{set shape <- shape add_z_pt {i,valZ};}
```

Top of the page

agent

- Possible use:
  - OP(any) --- > agent
- Result: casting of the operand to an agent (if a species name is used, casting to an instance of species name).
- Special cases:
  - if the operand is a point, returns the closest agent (resp. closest instance of species name) to that point (computed in the topology of the calling agent);
  - if the operand is an agent, returns the agent (resp. tries to cast this agent to species name and returns nil if the agent is instance of another species);
  - if the operand is an int, returns the agent (resp. instance of species name) with this unique index;
- See also: of_species , species ,

```
species node {}
node(0)      --: node0
node(3.78)          --: null
node(true)          --: null
node({23, 4.0}          --: node2
node(5::34)          --: null
node(green)          --: null
node([1,5,9,3])      --: null
node(node1)         --: node1
node('4')        --: null
```

Top of the page

agent_closest_to

- Possible use:
  - OP(any) --- > agent
- Result: A agent, the closest to the operand (casted as a geometry).
- Comment: the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.
- See also: neighbours_at , neighbours_of , agents_inside , agents_overlapping , closest_to , inside , overlapping ,

```
agent_closest_to(self) --: return the closest agent to the agent applying the operator.
```

Top of the page

agent_from_geometry

- Possible use:
  - path OP shape --- > agent
- Result: returns the agent corresponding to given geometry (right-hand operand) in the given path (left-hand operand).
- Special cases:
  - if the left-hand operand is nil, returns nil

```
let line type: geometry <- one_of(path_followed.segments);
let ag type: road <- road(path_followed agent_from_geometry line);
```

Top of the page

agents_at_distance

- Possible use:
  - OP(float) --- > list
- Result: A list of agents situated at a distance < = the right argument.
- Comment: Equivalent to neighbours_at with a left-hand argument equal to 'self'
- See also: neighbours_at , neighbours_of , agent_closest_to , agents_inside , closest_to , inside , overlapping , at_distance ,

```
agents_at_distance(20) --: all the agents (excluding the caller) which distance to the caller is
<= 20
```

Top of the page

agents_inside

- Possible use:
  - OP(any) --- > list of agents
- Result: A list of agents covered by the operand (casted as a geometry).
- See also: agent_closest_to , agents_overlapping , closest_to , inside , overlapping ,

```
agents_inside(self) --: return the agents that are covered by the shape of the agent applying the
operator.
```

Top of the page

agents_overlapping

- Possible use:
  - OP(any) --- > list of agents
- Result: A list of agents overlapping the operand (casted as a geometry).
- See also: neighbours_at , neighbours_of , agent_closest_to , agents_inside , closest_to , inside , overlapping , at_distance ,

```
agents_overlapping(self) --: return the agents that overlap the shape of the agent applying the operator.
```

Top of the page

among

- Possible use:
  - int OP map --- > map
  - int OP container --- > list
- Result: a list of length the value of the left-hand operand, containing random elements from the right-hand operand
- Special cases:
  - if the right-hand operand is a map, among returns a map of right-hand operand element instead of a list
  - if the right-hand operand is empty or nil, among returns a new empty list
  - if the left-hand operand is greater than the length of the right-hand operand, among returns the right-hand operand.

```
2 among [1::2, 3::4, 5::6]     --:     [1::2, 3::4]
3 among [1,2,4,3,5,7,6,8]         --:     [1,2,8]
3 among g2                     --:     [node6,node11,node7]
3 among list(node)                --:      [node1,node11,node4]
```

Top of the page

and

- Possible use:
  - bool OP any expression --- > bool
- Result: a bool value, equal to the logical and between the left-hand operand and the rigth-hand operand.
- Comment: both operands are always casted to bool before applying the operator. Thus, an expression like (1 and 0) is accepted and returns false.
- See also: bool , or ,

Top of the page

any

Same signification as one_of operator.

Top of the page

any_location_in

- Possible use:
  - OP(shape) --- > point
- Result: A point inside (or touching) the operand-geometry.
- See also: closest_points_with , farthest_point_to , points_at ,

```
any_location_in(square(5)) --: a point of the square, for example : {3,4.6}.
```

Top of the page

any_point_in

Same signification as any_location_in operator.

Top of the page

around

- Possible use:
  - float OP any --- > shape
- Result: A geometry resulting from the difference between a buffer around the right-operand casted in geometry at a distance left-operand (right-operand buffer left-operand) and the right-operand casted as geometry.
- Special cases:
  - returns a circle geometry of radius right-operand if the left-operand is nil
- See also: circle , cone , line , link , norm , point , polygon , polyline , rectangle , square , triangle ,

```
10 around circle(5) --: returns a the ring geometry between 5 and 10.
```

Top of the page

as

- Possible use:
  - any OP species --- > agent
- Result: casting of the left-hand operand to a species.
- Special cases:
  - if the right-hand operand is nil, transforms the left-hand operand into an agent
  - if the left-hand operand is nil, returns nil
  - if the left-hand operand is an agent, if the right-hand is a agent, returns it, otherwise returns nil
  - if the left-oprand is a integer, returns an agent with the right-operans as id
  - if the left-operand is a poiny, returns the agent the closest to right-hand operand
  - otherwise, returns nil
- See also: agent ,

Top of the page

as_4_grid

- Possible use:

  - shape OP point --- > matrix
- Result: A matrix of square geometries (grid with 4-neighbourhood) with dimension given by the rigth-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

```
self as_grid {10, 5} --: returns matrix of square geometries (grid with 4-neighbourhood) with
10 columns and 5 lines corresponding to the square tessellation of the geometry of the agent
applying the operator.
```

Top of the page

as_date

- Possible use:
  - OP(double) --- > string
  - double OP string --- > string
- Result: converts a number into a string with year, month, day, hour, minutes, second following a given pattern (right-hand operand)
- Comment: Pattern should include : "%Y %M %D %h %m %s" for outputting years, months, days, hours, minutes, seconds
- Special cases:
  - used as an unary operator, uses a defined pattern with years, months, days
- See also: as_time ,

```
as_date(22324234)     --: 8 months, 18 days
22324234 as_date "%M m %D d %h h %m m %s seconds"     --: 8 m 18 d 9 h 10 m 34 seconds
```

Top of the page

as_distance_graph

- Possible use:
  - container OP float --- > graph
- Result: creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices close enough (less than a distance, right-hand operand).
- Comment: as_distance_graph is more efficient for a list of points than as_intersection_graph.
- See also: as_intersection_graph , as_edge_graph ,

```
list(ant) as_distance_graph 3.0;
```

Top of the page

as_edge_graph

- Possible use:
  - OP(container) --- > graph
  - OP(map) --- > graph
- Result: creates a graph from the list/map of edges given as operand
- Special cases:
  - if the operand is a list, the graph will be built with elements of the list as vertices
  - if the operand is a map, the graph will be built by creating edges from pairs of the map
- See also: as_intersection_graph , as_distance_graph ,

```
as_edge_graph([{1,5},{12,45},{34,56}])  --:  build a graph with these three vertices and
reflexive links on each vertices
as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}])  --:  build a graph with these three vertices
and two edges
```

Top of the page

as_grid

- Possible use:
  - shape OP point --- >   matrix
- Result: A matrix of square geometries (grid with 8-neighbourhood) with dimension given by the rigth-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

```
self as_grid {10, 5} --: returns a matrix of square geometries (grid with 8-neighbourhood) with
10 columns and 5 lines corresponding to the square tessellation of the geometry of the agent
applying the operator.
```

Top of the page

as_int

- Possible use:
  - string OP int --- >   int
- Result: parses the string argument as a signed integer in the radix specified by the second argument.
- Special cases:
  - if the left operand is nil or empty, as_int returns 0
  - if the left operand does not represent an integer in the specified radix, as_int throws an exception
- See also: int ,

```
'20' as_int 10          --: 20;
'20' as_int 8          --: 16;
'20' as_int 16          --: 32
'1F' as_int 16         --: 31
'hello' as_int 32      --: 18306744
```

Top of the page

as_intersection_graph

- Possible use:
  - container OP float --- >   graph
- Result: creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices with an intersection (with a given tolerance).
- Comment: as_intersection_graph is more efficient for a list of geometries (but less accurate) than as_distance_graph.
- See also: as_distance_graph , as_edge_graph ,

```
list(ant) as_intersection_graph 0.5;
```

Top of the page

as_map

- Possible use:
  - container OP any expression --- > map
- Result: produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand
- Comment: the right-hand operand should be pair or a map.
- Special cases:
  - if the left-hand operand is nil or empty, as_map returns a new empty map.

```
[1,2,3,4,5,6,7,8] as_map (each::(each * 2)     --:     [1::2, 2::4, 3::6, 4::8, 5::10, 6::12,
7::14, 8::16]
[1::2,3::4,5::6] as_map (each::(each * 2))        --:     [2::4, 4::8, 6::12]
```

Top of the page

as_matrix

- Possible use:
  - any OP point --- > matrix
  - container OP point --- > matrix
- Result: casts the left operand into a matrix with right operand as preferrenced size
- Comment: This operator is very useful to cast a file containing raster data into a matrix.Note that both components of the right operand point should be positive, otherwise an exception is raised.The operator as_matrix creates a matrix of preferred size. It fills in it with elements of the left operand until the matrix is full If the size is to short, some elements will be omitted. Matrix remaining elements will be filled in by nil.
- Special cases:
  - if the right operand is nil, as_matrix is equivalent to the matrix operator
- See also: matrix ,

Top of the page

as_time

- Possible use:
  - OP(double) --- > string
- Result: converts the given number into a string with hours, minutes and seconds
- Comment: as_time operator is a particular case (using a particular pattern) of the as_date operator.
- See also: as_date ,

```
as_time(22324234)     --: 09:10:34
```

Top of the page

asin

- Possible use:
  - OP(int) --- > float
  - OP(float) --- > float

- Result: the arcsin of the operand (which has to be expressed in decimal degrees).
- See also: acos , atan ,

```
acos (90) --: 1
```

Top of the page

at

- Possible use:
  - container OP [KeyType] --- > [ValueType]
  - map OP any --- > any
  - string OP int --- > string
- Result: the element at the right operand index of the container
- Comment: The first element of the container is located at the index 0. In addition, if the user tries to get the element at an index higher or equals than the length of the container, he will get an [IndexOutOfBoundException] .The at operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list or a matrix, at returns the element at the index specified by the right operand
  - if it is a file, at returns the element of the file content at the index specified by the right operand
  - if it is a population, at returns the agent at the index specified by the right operand
  - if it is a graph and if the right operand is a node, at returns the in and out edges corresponding to that node
  - if it is a graph and if the right operand is an edge, at returns the pair node_out::node_in of the edge
  - if it is a graph and if the right operand is a pair node1::node2 , at returns the edge from node1 to node2 in the graph
  - if it is a map, at returns the value corresponding the right operand as key. If the right operand is not a key of the map, at returns nil
- See also: contains_any contains_all, contains_any ,

```
[1, 2, 3] at 2                  --:    3
[{1,2}, {3,4}, {5,6}] at 0     --: {1.0;2.0}
'abcdef' at 0     --: 'a';
```

Top of the page

at_distance

- Possible use:
  - list OP float --- > list
  - species OP float --- > list
- Result: A list of agents among the left-operand list that are located at a distance < = the right operand from the caller agent (in its topology)
- Special cases:
  - If the left operand is a species, return agents of the specified species (slightly more efficient than using list(species1), for instance)
- See also: neighbours_at , neighbours_of , agent_closest_to , agents_inside , closest_to , inside , overlapping , neighbours_at , neighbours_of , agent_closest_to , agents_inside , closest_to , inside , overlapping ,

```
[ag1, ag2, ag3] at_distance 20 --: return the agents of the list located at a distance <= 20 from the caller agent (in the same order).
```

```
species1 at_distance 20 --: return the agents of species1 located at a distance <= 20 from the caller agent.
```

Top of the page

at_location

- Possible use:
  - shape OP point --- > shape
- Result: A geometry resulting from the tran of a translation to the right-hand operand point of the left-hand operand (geometry, agent, point)

```
self at_location {10, 20} --: returns the geometry resulting from a translation to the location {10, 20} of the geometry of the agent applying the operator.
```

Top of the page

atan

- Possible use:
  - OP(float) --- > float
  - OP(int) --- > float
- Result: the arctan of the operand (which has to be expressed in decimal degrees).
- See also: acos , asin ,

```
atan (45) --: 1
```

Top of the page

binomial

- Possible use:
  - OP(point) --- > int
- Result: A value from a random variable following a binomial distribution. The operand {n,p} represents the number of experiments n and the success probability p.
- Comment: The binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p, cf. Binomial distribution on Wikipedia.
- See also: poisson , gauss ,

```
binomial({15,0.6})  --:  a random positive integer
```

Top of the page

bool

- Possible use:
  - OP(any) --- > bool
- Result: casting of the operand to a boolean value.
- Special cases:
  - if the operand is null, returns false;
  - if the operand is an agent, returns true if the agent is not dead;
  - if the operand is an int or a float, returns true if it is not equal to 0 (or 0.0);

- if the operand is a file, bool is formally equivalent to exists;
- if the operand is a container, bool is formally equivalent to not empty (a la Lisp);
- if the operand is a string, returns true is the operand is true;
- Otherwise, returns false.

```
bool(3.78)          --: true
bool(true)          --: true
bool({23, 4.0}      --: false
bool(5::34)          --: false
bool(green)          --: false
bool([1,5,9,3])      --: true
bool(node1)         --: true
bool('4')             --: false
bool('4.7')         --: false
```

Top of the page

buffer

Same signification as + operator.

Top of the page

ceil

- Possible use:
  - OP(double) --- > double
- Result: maps the operand to the smallest following integer.
- Comment: More precisely, ceiling(x) is the smallest integer not less than x.
- See also: floor , round ,

```
ceil(3)          --:  4.0
ceil(3.5)          --:  4.0
ceil(-4.7)      --:  -4.0
```

Top of the page

circle

- Possible use:
  - OP(float) --- > shape
- Result: A circle geometry which radius is equal to the operand.
- Comment: the centre of the circle is by default the location of the current agent in which has been called this operator.
- Special cases:
  - returns a point if the operand is lower or equal to 0.
- See also: around , cone , line , link , norm , point , polygon , polyline , rectangle , square , triangle ,

```
circle(10) --: returns a geometry as a circle of radius 10.
```

Top of the page

clean

- Possible use:
  - OP(shape) --- > shape
- Result: A geometry corresponding to the cleaning of the operand (geometry, agent, point)
- Comment: The cleaning corresponds to a buffer with a distance of 0.0

```
cleaning(self) --: returns the geometry resulting from the cleaning of the geometry of the agent
applying the operator.
```

Top of the page

closest_points_with

- Possible use:
  - shape OP shape --- > list of points
- Result: A list of two closest points between the two geometries.
- See also: any_location_in , any_point_in , farthest_point_to , points_at ,

```
geom1 closest_points_with(geom2) --: [pt1, pt2] with pt1 the closest point of geom1 to geom2 and
pt1 the closest point of geom2 to geom1
```

Top of the page

closest_to

- Possible use:
  - container of shapes OP shape --- > any
  - species OP shape --- > agent
- Result: An agent among the left-operand list, the closest to the operand (casted as a geometry).
- Comment: the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.
- Special cases:
  - if the left-operand is a species, return an agent of the specified species.
- See also: neighbours_at , neighbours_of , neighbours_at , neighbours_of , inside , overlapping , agents_overlapping , agents_inside , agent_closest_to ,

```
[ag1, ag2, ag3] closest_to(self) --: return the closest agent among ag1, ag2 and ag3 to the agent
applying the operator.
neighbours_at
neighbours_of
species1 closest_to(self) --: return the closest agent of species species1 to the agent applying
the operator.
```

Top of the page

collate

- Possible use:
  - OP(list) --- > list
- Result: a new list containing interleaved elements of the operand
- Comment: the operand should be a list of lists of elements. The result is a list of elements.

- Special cases:
  - if the operand is nil or a list of (non-list) elements, accumulate returns an empty list

```
collate([1,2,4,3,5,7,6,8])     --:     []
collate([['e11','e12','e13'],['e21','e22','e23'],['e31','e32','e33']])  --:
[e11,e21,e31,e12,e22,e32,e13,e23,e33]
```

Top of the page

collect

- Possible use:
  - container OP any expression --- > list
- Result: returns a new list, in which each element is the evaluation of the right-hand operand.
- Comment: collect is very similar to accumulate except. Nevertheless if the evaluation of the right-hand operand produces a list,the returned list is a list of list of elements. In contrarily, the list produces by accumulate is only a list of elements (all the lists) produced are concaneted. In addition, collect can be applied to any container.
- Special cases:
  - if the left-hand operand is nil, accumulate returns an empty list
- See also: accumulate ,

```
[1,2,4] collect (each *2)      --:     [2,4,8]
[1,2,4] collect ([2,4])          --:     [[2,4],[2,4],[2,4]]
[1::2, 3::4, 5::6] collect (each + 2) --: [8,4,6]
(list(node) collect (node(each).location.x * 2)    --:  [25.65, 158.99, 140.80, 80.11, 125.47,
37.830, 4.62,...]
```

Top of the page

column_at

- Possible use:
  - matrix OP int --- > list
- Result: returns the column at a num_col (rigth-hand operand)
- See also: row_at , rows_list ,

```
matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]) column_at 2  --:
["el31","el32","el33"]
```

Top of the page

columns_list

- Possible use:
  - OP(matrix) --- > list of lists
- Result: returns a list of the columns of the matrix, with each column as a list of elements
- See also: rows_list ,

```
columns_list(matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]])  --:
      [["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]
```

Top of the page

cone

- Possible use:
  - OP(point) --- > shape
- Result: A cone geometry which min and max angles are given by the operands.
- Comment: the centre of the cone is by default the location of the current agent in which has been called this operator.
- Special cases:
  - returns nil if the operand is nil.
- See also: around , circle , line , link , norm , point , polygon , polyline , rectangle , square , triangle ,

```
cone({0, 45}) --: returns a geometry as a cone with min angle is 0 and max angle is 45.
```

Top of the page

container

- Possible use:
  - OP(any) --- > container
- Result: casting of the operand to a container
- Special cases:
  - if the operand is a container, returns itself
  - otherwise, returns the operand casted to a list
- See also: list ,

Top of the page

contains

- Possible use:
  - container OP any --- > boolean
  - string OP string --- > bool
- Result: true, if the container contains the right operand, false otherwise
- Comment: the contains operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list or a matrix, contains returns true if the list or matrix contains the right operand
  - if it is a map, contains returns true if the operand is a key of the map
  - if it is a file, contains returns true it the operand is contained in the file content
  - if it is a population, contains returns true if the operand is an agent of the population, false otherwise
  - if it is a graph, contains returns true if the operand is a node or an edge of the graph, false otherwise
  - if both operands are strings, returns true if the right-hand operand contains the right-hand pattern;
- See also: contains_any contains_all, contains_any ,

```
[1, 2, 3] contains 2            --:   true
[{1,2}, {3,4}, {5,6}] contains {3,4}     --:   true
'abcded' contains 'bc'     --:  true
```

Top of the page

contains_all

- Possible use:
  - container OP container --- > bool
  - string OP list --- > bool
- Result: true if the left operand contains all the elements of the right operand, false otherwise
- Comment: the definition of contains depends on the container
- Special cases:
  - if the right operand is nil or empty, contains_all returns true
- See also: contains , contains_any ,

```
[1,2,3,4,5,6] contains_all [2,4]         --:     true
[1,2,3,4,5,6] contains_all [2,8]         --:     false
[1::2, 3::4, 5::6] contains_all [1,3]     --:     true
[1::2, 3::4, 5::6] contains_all [2,4]     --:     false
"abcabcabc" contains_all ["ca","xy"]     --:     false
```

Top of the page

contains_any

- Possible use:
  - string OP list --- > bool
  - container OP container --- > bool
- Result: true if the left operand contains one of the elements of the right operand, false otherwise
- Comment: the definition of contains depends on the container
- Special cases:
  - if the right operand is nil or empty, contains_any returns false
- See also: contains , contains_all ,

```
"abcabcabc" contains_any ["ca","xy"]     --:     true
[1,2,3,4,5,6] contains_any [2,4]         --:     true
[1,2,3,4,5,6] contains_any [2,8]         --:     true
[1::2, 3::4, 5::6] contains_any [1,3]     --:     true
[1::2, 3::4, 5::6] contains_any [2,4]     --:     false
```

Top of the page

contains_edge

- Possible use:
  - graph OP any --- > bool
  - graph OP pair --- > bool
- Result: returns true if the graph(left-hand operand) contains the given edge (righ-hand operand), false otherwise
- Special cases:
  - if the left-hand operand is nil, returns false
  - if the right-hand operand is a pair, returns true if it exists an edge between the two elements of the pair in the graph
- See also: contains_vertex ,

```
let graphFromMap type: graph <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
graphFromMap contains_edge link({1,5}::{12,45})  --: true
```

```
let graphEpidemio type: graph <-
generate_barabasi_albert( ["edges_specy"::edge,"vertices_specy"::node,"size"::3,"m"::5] );
graphEpidemio contains_edge (node(0)::node(3));   --:   true
```

Top of the page

contains_vertex

- Possible use:
  - graph OP any --- > bool
- Result: returns true if the graph(left-hand operand) contains the given vertex (righ-hand operand), false otherwise
- Special cases:
  - if the left-hand operand is nil, returns false
- See also: contains_edge ,

```
let graphFromMap type: graph <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
graphFromMap contains_vertex {1,5}  --: true
```

Top of the page

convex_hull

- Possible use:
  - OP(shape) --- > shape
- Result: A geometry corresponding to the convex hull of the operand.

```
convex_hull(self) --: returns the convex hull of the geometry of the agent applying the operator
```

Top of the page

copy

- Possible use:
  - OP(any) --- > any
- Result: returns a copy of the operand.

Top of the page

copy_between

- Possible use:
  - list OP point --- > list
  - string OP point --- > string
- Result: returns a copy of a sublist of the left operand between a begin index (x of the right operand point) and a end index (y of the right operand point
- Special cases:
  - if the right operand is nil or empty, copy_between returns a copy of the left operand
  - if the begin index is higher than the end index, copy_between returns a new empty list

```
[1,2,3,4,5,6,7] copy_between {0,3}     --:         [1,2,3]
"abcabcabc" copy_between {2,6}    --:     cabc
```

Top of the page

corR

- Possible use:
  - container OP container --- > any

Top of the page

cos

- Possible use:
  - OP(float) --- > float
  - OP(int) --- > float
- Result: the cosinus of the operand (in decimal degrees).
- Special cases:
  - the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- See also: sin , tan ,

```
cos (0) --: 1
```

Top of the page

count

- Possible use:
  - container OP any expression --- > int
- Result: returns an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.
- Special cases:
  - if the left-hand operand is nil, count returns 0
- See also: group_by ,

```
[1,2,3,4,5,6,7,8] count (each > 3)                  --:     5
g2 count (length(g2 out_edges_of each) = 0  )          --:     5    // Number of nodes of
graph g2 without any out edge
(list(node) count (round(node(each).location.x) > 32)  --:      2     // Number of agents node
with x > 32
[1::2, 3::4, 5::6] count (each > 4)                     --:     1
```

Top of the page

crosses

- Possible use:
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) crosses the right-geometry (or agent/point).
- Special cases:
  - if one of the operand is null, returns false.
  - if one operand is a point, returns false.

- See also: <--: , disjoint_from , intersects , overlaps , partially_overlaps , touches ,

```
polyline([{10,10},{20,20}]) crosses polyline([{10,20},{20,10}]) --: true.
polyline([{10,10},{20,20}]) crosses geometry({15,15}) --: false
polyline([{0,0},{25,25}]) crosses polygon([{10,10},{10,20},{20,20},{20,10}]) --: true
```

Top of the page

dead

- Possible use:
  - OP(agent) --- > bool
- Result: true if the agent is dead, false otherwise.

```
dead(agent_A)     --:     true or false
```

Top of the page

degree_of

- Possible use:
  - graph OP any --- > int
- Result: returns the degree (in+out) of a vertex (right-hand operand) in the graph given as left-hand operand.
- See also: in_degree_of , out_degree_of ,

```
graphEpidemio degree_of (node(3))
```

Top of the page

directed

- Possible use:
  - OP(graph) --- > graph
- Result: the operand graph becomes a directed graph.
- Comment: the operator alters the operand graph, it does not create a new one.
- See also: undirected ,

Top of the page

direction_between

- Possible use:
  - topology OP container of shapes --- > int
- Result: A direction (in degree) between a list of two geometries (geometries, agents, points) considering a topology.
- See also: towards , direction_to , distance_to , distance_between , path_between , path_to ,

```
my_topology direction_between [ag1, ag2] --: the direction between ag1 and ag2 considering the topology my_topology
```

Top of the page

direction_to

Same signification as towards operator.

Top of the page

disjoint_from

- Possible use:
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) is disjoints from the right-geometry (or agent/point).
- Special cases:
  - if one of the operand is null, returns true.
  - if one operand is a point, returns false if the point is included in the geometry.
- See also: intersects , crosses , overlaps , partially_overlaps , touches ,

```
polyline([{10,10},{20,20}]) disjoint_from polyline([{15,15},{25,25}]) --: false.
polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from polygon([{15,15},{15,25},{25,25},
{25,15}]) --: false.
polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from geometry({15,15}) --: false.
polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from geometry({25,25}) --: true.
polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from polygon([{35,35},{35,45},{45,45},
{45,35}]) --: true
```

Top of the page

distance_between

- Possible use:
  - topology OP container of shapes --- > float
- Result: A distance between a list of geometries (geometries, agents, points) considering a topology.
- See also: towards , direction_to , distance_to , direction_between , path_between , path_to ,

```
my_topology distance_between [ag1, ag2, ag3] --: the distance between ag1, ag2 and ag3
considering the topology my_topology
```

Top of the page

distance_to

- Possible use:
  - point OP point --- > float
  - shape OP shape --- > float
- Result: A distance between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.
- See also: towards , direction_to , distance_between , direction_between , path_between , path_to ,

```
ag1 distance_to ag2 --: the distance between ag1 and ag2 considering the topology of the agent
applying the operator
```

Top of the page

div

- Possible use:
  - float OP int --- > int
  - int OP float --- > int
  - int OP int --- > int
  - float OP float --- > int
- Result: an int, equal to the truncation of the division of the left-hand operand by the rigth-hand operand.
- Special cases:
  - if the right-hand operand is equal to zero, raises an exception.
- See also: mod ,

```
40 div 4.1        --:  9
40 div 3     --:  13
```

Top of the page

empty

- Possible use:
  - OP(container) --- > boolean
  - OP(string) --- > bool
- Result: true if the operand is empty, false otherwise.
- Comment: the empty operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list, empty returns true if there is no element in the list, and false otherwise
  - if it is a map, empty returns true if the map contains no key-value mappings, and false otherwise
  - if it is a file, empty returns true if the content of the file (that is also a container) is empty, and false otherwise
  - if it is a population, empty returns true if there is no agent in the population, and false otherwise
  - if it is a graph, empty returns true if it contains no vertex and no edge, and false otherwise
  - if it is a matrix of int, float or object, it will return true if all elements are respectively 0, 0.0 or null, and false otherwise
  - if it is a matrix of geometry, it will return true if the matrix contains no cell, and false otherwise
  - if it is a string, empty returns true if the string does not contain any character, and false otherwise

```
empty ([])     --:     true;
empty ('abced')     --:     false
```

Top of the page

enlarged_by

Same signification as + operator.

Top of the page

eval_gaml

- Possible use:

  - OP(string) --- > any
- Result: evaluates the given GAML string.
- See also: eval_java ,

```
eval_gaml("2+3")    --:   5
```

Top of the page

evaluate_with

- Possible use:
  - string OP any expression --- > any
- Result: evaluates the left-hand java expressions with the map of parameters (right-hand operand)
- See also: eval_gaml , eval_java ,

Top of the page

even

- Possible use:
  - OP(int) --- > bool
- Result: true if the operand is even and false if it is odd.
- Special cases:
  - if the operand is equal to 0, it returns true.

```
even (3)     --:   false
even (-12)   --:  true
```

Top of the page

every

- Possible use:
  - OP(int) --- > bool
- Result: true every operand time step, false otherwise
- Comment: the value of the every operator depends deeply on the time step. It can be used to do something not every step.

```
reflex text_every {
    if every(2) {write "the time step is even";}
        else {write "the time step is odd";}
```

Top of the page

exp

- Possible use:
  - OP(float) --- > float
  - OP(int) --- > float
- Result: returns Euler's number e raised to the power of the operand.
- Special cases:
  - the operand is casted to a float before being evaluated.

- See also: ln ,

```
exp (0)     --:     1
```

Top of the page

fact

- Possible use:
  - OP(int) --- > int
- Result: the factorial of the operand.
- Special cases:
  - if the operand is less than 0, fact returns 0.

```
fact (4)     --:     24
```

Top of the page

farthest_point_to

- Possible use:
  - shape OP point --- > point
- Result: the farthest point of the left-operand to the left-point.
- See also: any_location_in , any_point_in , closest_points_with , points_at ,

```
geom farthest_point_to(pt) --: the closest point of geom to pt
```

Top of the page

file

- Possible use:
  - OP(string) --- > file
- Result: opens a file in read only mode, creates a GAML file object, and tries to determine and store the file content in the contents attribute.
- Comment: The file should have a supported extension, see file type deifnition for supported file extensions.
- Special cases:
  - If the specified string does not refer to an existing file, an exception is risen when the variable is used.
- See also: folder , new_folder ,

```
let fileT type: file value: file("../includes/Stupid_Cell.Data");
            // fileT represents the file "../includes/Stupid_Cell.Data"
            // fileT.contents here contains a matrix storing all the data of the text file
```

Top of the page

first

- Possible use:
  - OP(string) --- > string
  - OP(container) --- > [ValueType]

- Result: the first element of the operand
- Comment: the first operator behavior depends on the nature of the operand
- Special cases:
    - if it is a string, first returns a string composed of its first character
    - if it is a list, first returns the first element of the list, or nil if the list is empty
    - if it is a map, first returns nil (the map do not keep track of the order of elements)
    - if it is a file, first returns the first element of the content of the file (that is also a container)
    - if it is a population, first returns the first agent of the population
    - if it is a graph, first returns the first element in the list of vertexes
    - if it is a matrix, first returns the element at {0,0} in the matrix
    - for a matrix of int or float, it will return 0 if the matrix is empty
    - for a matrix of object or geometry, it will return null if the matrix is empty
- See also: last ,

```
first ('abce')          --:   'a'
first ([1, 2, 3])         --:   1
first ({10,12})         --:   10.
```

Top of the page

first_with

- Possible use:
    - container OP any expression --- >   any
- Result: the first element of the left-hand operand that makes the right-hand operand evaluate to true.
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
    - if the left-hand operand is nil, first_with returns nil
- See also: group_by , last_with , where ,

```
[1,2,3,4,5,6,7,8] first_with (each > 3)                       --:     4
g2 first_with (length(g2 out_edges_of each) = 0)               --:     node9
(list(node) first_with (round(node(each).location.x) > 32)     --:     node2
[1::2, 3::4, 5::6] first_with (each.key > 4)                   --:     5::6
```

Top of the page

flip

- Possible use:
    - OP(float) --- >   bool
- Result: true or false given the probability represented by the operand
- Special cases:
    - flip 0 always returns false, flip 1 true
- See also: rnd ,

```
flip (0.66666) --: 2/3 chances to return true.
```

Top of the page

float

- Possible use:
  - OP(any) --- > float
- Result: casting of the operand to a floating point value.
- Special cases:
  - if the operand is numerical value, returns its value as a floating point value;
  - if the operand is a string, tries to convert its content to a floating point value;
  - if the operand is a boolean, returns 1.0 for true and 0.0 for false;
  - otherwise, returns 0.0
- See also: int ,

```
float(7)                  --: 7.0
float(true)              --: 1.0
float({23, 4.0}          --: 0.0
float(5::34)              --: 0.0
float(green)              --: 0.0
float([1,5,9,3])          --: 0.0
float(node1)             --: 0.0
int('4')                 --: 4.0
int('4.7')                 --: 4.7
```

Top of the page

floor

- Possible use:
  - OP(double) --- > double
- Result: maps the operand to the largest previous following integer.
- Comment: More precisely, floor(x) is the largest integer not greater than x.
- See also: ceil , round ,

```
floor(3)        --:  3.0
floor(3.5)     --:  3.0
floor(-4.7)     --:  -5.0
```

Top of the page

folder

- Possible use:
  - OP(string) --- > file
- Result: opens an existing repository
- Special cases:
  - If the specified string does not refer to an existing repository, an exception is risen.
- See also: file , new_folder ,

```
let dirT type: file value: folder("../includes/");
                // dirT represents the repository "../includes/"
                // dirT.contents here contains the list of the names of included files
```

Top of the page

frequency_of

- Possible use:
  - container OP any expression --- > map
- Result: Returns a map with keys equal to the application of the right-hand argument (like collect) and values equal to the frequency of this key (i.e. how many times it has been obtained)
- See also: as_map ,

```
[ag1, ag2, ag3, ag4] frequency_of each.size     --:   will return the different sizes as keys and
the number of agents of this size as values
```

Top of the page

gauss

- Possible use:
  - OP(point) --- > float
- Result: A value from a normally distributed random variable with expected value (mean) and variance (standardDeviation). The probability density function of such a variable is a Gaussian.
- Special cases:
  - when the operand is a point, it is read as {mean, standardDeviation}
  - when standardDeviation value is 0.0, it always returns the mean value
- See also: truncated_gauss , poisson ,

```
gauss({0,0.3})  --:  0.22354
gauss({0,0.3})  --:  -0.1357
```

Top of the page

generate_barabasi_albert

- Possible use:
  - OP(map) --- > graph
- Result: returns a random scale-free network (following Barabasi#Albert (BA) model).
- Comment: The Barabasi#Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article] The map operand should includes following elements:
- Special cases:
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
  - "size": the graph will contain (size + 1) nodes
  - "m": the number of edges added per novel node
- See also: generate_watts_strogatz ,

```
let graphEpidemio type: graph <- generate_barabasi_albert( [
        "edges_specy"::edge,
        "vertices_specy"::node,
        "size"::3,
        "m"::5] );
```

Top of the page

generate_watts_strogatz

- Possible use:
  - OP(map) --- > graph
- Result: returns a random small-world network (following Watts-Strogatz model).
- Comment: The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article] The map operand should includes following elements:
- Special cases:
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
  - "size": the graph will contain (size + 1) nodes. Size must be greater than k.
  - "p": probability to "rewire" an edge. So it must be between 0 and 1. The parameter is often called beta in the literature.
  - "k": the base degree of each node. k must be greater than 2 and even.
- See also: generate_barabasi_albert ,

```
let graphWatts type: graph <- generate_watts_strogatz( ["
            "edges_specy"::edge,
            "vertices_specy"::node,
            "size"::2,
            "p"::0.3,
            "k"::0] );
```

Top of the page

geometric_mean

- Possible use:
  - OP(list) --- > float
- Result: the geometric mean of the elements of the operand. See Geometric_mean < http://en.wikipedia.org/wiki/Geometric_mean> for more details.
- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean , median , harmonic_mean ,

```
geometric_mean ([4.5, 3.5, 5.5, 7.0]) --: 4.962326343467649
```

Top of the page

geometry

- Possible use:
  - OP(any) --- > shape
- Result: casts the operand into a geometry
- Special cases:
  - if the operand is a point, returns a corresponding geometry point
  - if the operand is a agent, returns its geometry

- if the operand is a population, returns the union of each agent geometry
- if the operand is a pair of two agents or geometries, returns the link between the geometry of each element of the operand
- if the operans is a graph, returns the corresponding multi-points geometry
- if the operand is a container of points, if first and the last points are the same, returns the polygon built from these points
- if the operand is a container, returns the union of the geometry of each element
- otherwise, returns nil

```
geometry({23, 4.0})                     --: Point
geometry(a_graph)                        --: MultiPoint
geometry(node1)                        --: Point
geometry([{0,0},{1,4},{4,8},{0,0}])    --: Polygon
```

Top of the page

get

- Possible use:
  - shape OP string --- > any

Top of the page

graph

- Possible use:
  - OP(any) --- > graph
- Result: casting of the operand to a graph.
- Special cases:
  - if the operand is a graph, returns the graph itself
  - if the operand is a list, returns a new graph with the elements of the left-hand operand as vertices and no edge. The graph will be spatial is the right-hand operand is true;
  - if the operand is a map,
  - otherwise, returns nil

```
graph([1,5,9,3])     --: ([1: in[] + out[], 3: in[] + out[], 5: in[] + out[], 9: in[] + out[]], [])
graph(['a'::345, 'b'::13])  --:  ([b: in[] + out[b::13], a: in[] + out[a::345], 13: in[b::13] + out[], 345: in[a::345] + out[]], [a::345=(a,345), b::13=(b,13)])
graph(a_graph)    --: a_graph
graph(node1)    --: null
```

Top of the page

grid_at

- Possible use:
  - species OP point --- > agent
- Result: returns the cell of the grid (right-hand operand) at the position given by the right-hand operand
- Comment: If the left-hand operand is a point of floats, it is used as a point of ints.
- Special cases:
  - if the left-hand operand is not a grid cell species, returns nil

```
grid_cell grid_at {1,2}     --:     returns the agent grid_cell with grid_x=1 and grid_y = 2
```

Top of the page

group_by

- Possible use:
  - container OP any expression --- > map
  - map OP any expression --- > map
- Result: a map, where the keys take the possible values of the right-hand operand and the map values are the list of elements of the left-hand operand associated to the key value
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is nil, group_by returns a new empty map
- See also: first_with , last_with , where ,

```
[1,2,3,4,5,6,7,8] group_by (each > 3)     --:     [false::[1, 2, 3], true::[4, 5, 6, 7, 8]]
g2 group_by (length(g2 out_edges_of each) )     --:     [ 0::[node9, node7, node10, node8,
node11], 1::[node6], 2::[node5], 3::[node4]]
(list(node) group_by (round(node(each).location.x))    --:  [32::[node5], 21::[node1], 4::
[node0], 66::[node2], 96::[node3]]
[1::2, 3::4, 5::6] group_by (each > 4)     --:     [false::[2, 4], true::[6]]
```

Top of the page

harmonic_mean

- Possible use:
  - OP(list) --- > float
- Result: the harmonic mean of the elements of the operand. See Harmonic_mean < http://en.wikipedia.org/wiki/Harmonic_mean> for more details.
- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean , median , geometric_mean ,

```
    harmonic_mean ([4.5, 3.5, 5.5, 7.0]) --: 4.804159445407279
```

Top of the page

image

- Possible use:
  - OP(string) --- > file
- Result: opens a file that is a kind of image.
- Comment: The file should have an image extension, cf. file type deifnition for supported file extensions.
- Special cases:
  - If the specified string does not refer to an existing image file, an exception is risen.
- See also: file , shapefile , properties , text ,

```
let fileT type: file value: image("../includes/testImage.png");  // fileT represents the file
"../includes/testShape.png"
```

Top of the page

in

- Possible use:
  - string OP string --- > bool
  - any OP container --- > bool
- Result: true if the right operand contains the left operand, false otherwise
- Comment: the definition of in depends on the container
- Special cases:
  - if both operands are strings, returns true if the left-hand operand patterns is included in to the right-hand string;
  - if the right operand is nil or empty, in returns false
- See also: contains ,

```
'bc' in 'abcded'      --:  true
2 in [1,2,3,4,5,6] : true
7 in [1,2,3,4,5,6] : false
3 in [1::2, 3::4, 5::6] : true
6 in [1::2, 3::4, 5::6] : false
```

Top of the page

in_degree_of

- Possible use:
  - graph OP any --- > int
- Result: returns the in degree of a vertex (right-hand operand) in the graph given as left-hand operand.
- See also: out_degree_of , degree_of ,

```
graphEpidemio in_degree_of (node(3))   --:  2
```

Top of the page

in_edges_of

- Possible use:
  - graph OP any --- > list
- Result: returns the list of the in-edges of a vertex (right-hand operand) in the graph given as left-hand operand.
- See also: out_edges_of ,

```
graphFromMap in_edges_of node({12,45})  --:  [LineString]
```

Top of the page

index_of

- Possible use:
  - matrix OP any --- > point
  - string OP string --- > int

    - list OP any --- > int
    - map OP any --- > any
- Result: the index of the first occurence of the right operand in the left operand container
- Comment: The definition of index_of and the type of the index depend on the container
- Special cases:
    - if the left operand is a matrix, index_of returns the index as a point
    - if both operands are strings, returns the index within the left-hand string of the first occurrence of the given right-hand string
    - if the left operand is a list, index_of returns the index as an integer
    - if the left operand is a map, index_of returns the index as a pair
- See also: at , last_index_of ,

```
matrix([[1,2,3],[4,5,6]]) index_of 4      --:     {1.0;0.0}
"abcabcabc" index_of "ca"     --:     2
[1,2,3,4,5,6] index_of 4     --:     3
[4,2,3,4,5,4] index_of 4      --:     0
[1::2, 3::4, 5::6] index_of 4          --:     3::4
```

Top of the page

inside

- Possible use:
    - species OP any --- > list of agents
    - container of shapes OP any --- > list of agents
- Result: A list of agents among the left-operand list, covered by the operand (casted as a geometry).
- Special cases:
    - if the left-operand is a species, return agents of the specified species (slightly more efficient than using list(species1), for instance).
- See also: neighbours_at , neighbours_of , closest_to , overlapping , agents_overlapping , agents_inside , agent_closest_to ,

```
species1 inside(self) --: return the agents of species species1 that are covered by the shape of the agent applying the operator.
[ag1, ag2, ag3] inside(self) --: return the agents among ag1, ag2 and ag3 that are covered by the shape of the agent applying the operator.
```

Top of the page

int

- Possible use:
    - OP(any) --- > int
- Result: casting of the operand to an integer value.
- Special cases:
    - if the operand is a float, returns its value truncated (but not rounded);
    - if the operand is an agent, returns its unique index;
    - if the operand is a string, tries to convert its content to an integer value;
    - if the operand is a boolean, returns 1 for true and 0 for false;
    - if the operand is a color, returns its RGB value as an integer;
    - otherwise, returns 0
- See also: round , float ,

```
int(3.78)              --: 3
int(true)              --: 1
int({23, 4.0}          --: 0
int(5::34)          --: 0
int(green)          --: -16711936
int([1,5,9,3])      --: 0
int(node1)             --: 1
int('4')             --: 4
int('4.7')             --:  // Exception
```

Top of the page

inter

- Possible use:
  - shape OP shape --- > shape
  - container OP container --- > list
- Result: A geometry resulting from the intersection between the two geometriesthe intersection of the two operands
- Comment: both containers are transformed into sets (so without duplicated element, cf. remove_deplicates operator) before the set intersection is computed.
- Special cases:
  - returns false if the right-operand is nil
  - if an operand is a graph, it will be transformed into the set of its nodes
  - if an operand is a map, it will be transformed into the set of its values
  - if an operand is a matrix, it will be transformed into the set of the lines
- See also: union , + , - , remove_duplicates ,

```
square(5) intersects {10,10} --: false
[1,2,3,4,5,6] inter [2,4]                --:     [2,4]
[1,2,3,4,5,6] inter [0,8]                --:     []
[1::2, 3::4, 5::6] inter [2,4]         --:     [2,4]
[1::2, 3::4, 5::6] inter [1,3]         --:     []
matrix([[1,2,3],[4,5,4]]) inter [3,4]      --:     [4,3]
```

Top of the page

intersection

Same signification as inter operator.

Top of the page

intersects

- Possible use:
  - shape OP point --- > bool
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) intersects the right-geometry (or agent/point).
- Special cases:
  - if one of the operand is null, returns false.
- See also: <--: , disjoint_from , crosses , overlaps , partially_overlaps , touches ,

```
square(5) intersects {10,10} --: false.
```

Top of the page

is

- Possible use:
  - any OP any expression --- > bool
- Result: returns true is the left operand is of the right operand type, false otherwise

```
0 is int          --:     true
an_agent is node      --:     true
1 is float          --:     false
```

Top of the page

is_image

- Possible use:
  - OP(string) --- > bool
- Result: the operator tests whether the operand represents the name of a supported image file
- Comment: cf. file type definition for supported (espacially image) file extensions.
- See also: image , is_text , is_properties , is_shape ,

```
is_image("../includes/Stupid_Cell.Data")    --:  false;
is_image("../includes/test.png")            --:  true;
is_image("../includes/test.properties")     --:  false;
is_image("../includes/test.shp")            --:  false;
```

Top of the page

is_number

- Possible use:
  - OP(string) --- > bool
- Result: tests whether the operand represents a numerical value
- Comment: Note that the symbol . should be used for a float value (a string with , will not be considered as a numeric value). Symbols e and E are also accepted. A hexadecimal value should begin with #.

```
is_number("test")      --: false
is_number("123.56")      --: true
is_number("-1.2e5")      --: true
is_number("1,2")      --: false
is_number("#12FA")      --: true
```

Top of the page

is_properties

- Possible use:
  - OP(string) --- > bool
- Result: the operator tests whether the operand represents the name of a supported properties file

- Comment: cf. file type definition for supported (espacially image) file extensions.
- See also: properties , is_text , is_shape , is_image ,

```
is_properties("../includes/Stupid_Cell.Data")    --:  false;
is_properties("../includes/test.png")            --:  false;
is_properties("../includes/test.properties")     --:  true;
is_properties("../includes/test.shp")            --:  false;
```

Top of the page

is_shape

- Possible use:
  - OP(string) --- >   bool
- Result: the operator tests whether the operand represents the name of a supported shapefile
- Comment: cf. file type definition for supported (espacially image) file extensions.
- See also: image , is_text , is_properties , is_image ,

```
is_shape("../includes/Stupid_Cell.Data")    --:  false;
is_shape("../includes/test.png")            --:  false;
is_shape("../includes/test.properties")     --:  false;
is_shape("../includes/test.shp")            --:  true;
```

Top of the page

is_text

- Possible use:
  - OP(string) --- >   bool
- Result: the operator tests whether the operand represents the name of a supported text file
- Comment: cf. file type definition for supported (espacially image) file extensions.
- See also: text , is_properties , is_shape , is_image ,

```
is_text("../includes/Stupid_Cell.Data")    --:  true;
is_text("../includes/test.png")            --:  false;
is_text("../includes/test.properties")     --:  false;
is_text("../includes/test.shp")            --:  false;
```

Top of the page

last

- Possible use:
  - OP(string) --- >   string
  - OP(container) --- > [ValueType]
- Result: the last element of the operand
- Comment: the last operator behavior depends on the nature of the operand
- Special cases:
  - if it is a string, last returns a string composed of its last character, or an empty string if the operand is empty
  - if it is a list, last returns the last element of the list, or nil if the list is empty
  - if it is a map, last returns nil (the map do not keep track of the order of elements)
  - if it is a file, last returns the last element of the content of the file (that is also a container)

    - if it is a population, last returns the last agent of the population
    - if it is a graph, last returns the last element in the list of vertexes
    - if it is a matrix, last returns the element at {length-1,length-1} in the matrix
    - for a matrix of int or float, it will return 0 if the matrix is empty
    - for a matrix of object or geometry, it will return null if the matrix is empty
- See also: first ,

```
last ('abce')         --:   'e'
last ({10,12})     --:   12
last ([1, 2, 3])     --:   3.
```

Top of the page

last_index_of

- Possible use:
    - string OP string --- >   int
    - matrix OP any --- >   point
    - map OP any --- >   any
    - list OP any --- >   int
- Result: the index of the last occurence of the right operand in the left operand container
- Comment: The definition of last_index_of and the type of the index depend on the container
- Special cases:
    - if both operands are strings, returns the index within the left-hand string of the rightmost occurrence of the given right-hand string
    - if the left operand is a matrix, last_index_of returns the index as a point
    - if the left operand is a map, last_index_of returns the index as a pair
    - if the left operand is a list, last_index_of returns the index as an integer
- See also: at , last_index_of ,

```
"abcabcabc" last_index_of "ca"     --:     5
matrix([[1,2,3],[4,5,4]]) last_index_of 4      --:     {1.0;2.0}
[1::2, 3::4, 5::4] last_index_of 4      --:      5::4
[1,2,3,4,5,6] last_index_of 4      --:     3
[4,2,3,4,5,4] last_index_of 4      --:     5
```

Top of the page

last_with

- Possible use:
    - container OP any expression --- >   any
- Result: the last element of the left-hand operand that makes the right-hand operand evaluate to true.
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
    - if the left-hand operand is nil, last_with returns nil
- See also: group_by , first_with , where ,

```
[1,2,3,4,5,6,7,8] last_with (each > 3)                      --:     8
g2 last_with (length(g2 out_edges_of each) = 0 )              --:     node11
(list(node) last_with (round(node(each).location.x) > 32)   --:     node3
[1::2, 3::4, 5::6] last_with (each.key > 4)                  --:     5::6
```

Top of the page

length

- Possible use:
  - OP(string) --- > int
  - OP(container) --- > int
- Result: the number of elements contained in the operand
- Comment: the length operator behavior depends on the nature of the operand
- Special cases:
  - if it is a string, length returns the number of characters
  - if it is a list or a map, length returns the number of elements in the list or map
  - if it is a population, length returns number of agents of the population
  - if it is a graph, last returns the number of vertexes or of edges (depending on the way it was created)
  - if it is a matrix, length returns the number of cells

```
length ('I am an agent')         --:     13
length ([12,13])    --:     2
```

Top of the page

line

- Possible use:
  - OP(list of points) --- > shape
- Result: A polyline geometry from the given list of points.
- Special cases:
  - if the operand is nil, returns the point geometry {0,0}
  - if the operand is composed of a single point, returns a point geometry.
- See also: around , circle , cone , link , norm , point , polygone , rectangle , square , triangle ,

```
polyline([{0,0}, {0,10}, {10,10}, {10,0}]) --: returns a polyline geometry composed of the 4
points.
```

Top of the page

link

- Possible use:
  - OP(pair) --- > shape
- Result: A link between the 2 elements of the pair.
- Comment: The geometry of the link is the intersection of the two geometries when they intersect, and a line between their centroids when they do not.
- Special cases:
  - if the operand is nil, link returns a point {0,0}
  - if one of the elements of the pair is a list of geometries or a species, link will consider the union of the geometries or of the geometry of each agent of the species
- See also: around , circle , cone , line , norm , point , polygon , polyline , rectangle , square , triangle ,

```
link (geom1::geom2)  --: returns a link geometry between geom1 and geom2.
```

Top of the page

list

- Possible use:
  - OP(any) --- > list
  - OP(container) --- > list
- Result: transforms the operand into a list
- Comment: list always tries to cast the operand except if it is an int, a bool or a float; to create a list, instead, containing the operand (including another list), use the + operator on an empty list (like [] + 'abc').
- Special cases:
  - if the operand is a point or a pair, returns a list containing its components (two coordinates or the key and the value);
  - if the operand is a rgb color, returns a list containing its three integer components;
  - if the operand is a file, returns its contents as a list;
  - if the operand is a matrix, returns a list containing its elements;
  - if the operand is a graph, returns the list of vetices or edges (depending on the graph)
  - if the operand is a species, return a list of its agents;
  - if the operand is a string, returns a list of strings, each containing one character;
  - otherwise returns a list containing the operand.

Top of the page

ln

- Possible use:
  - OP(int) --- > float
  - OP(float) --- > float
- Result: returns the natural logarithm (base e) of the operand.
- Special cases:
  - an exception is raised if the operand is less than zero.
- See also: exp ,

```
ln(1)     --:     0.0
```

Top of the page

load_graph_from_dgs

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following DGS graph file format versions 1 and 2
- Comment: similar to load_graph_from_dgs_old
- See also: load_graph_from_dgs_old ,

Top of the page

load_graph_from_dgs_old

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following DGS file format (version 3).
- Comment: DGS is a file format allowing to store graphs and dynamic graphs in a textual human readable way, yet with a small size allowing to store large graphs. Graph dynamics is defined using events like adding, deleting or changing a node or edge. With DGS, graphs will therefore be seen as stream of such events. [From GraphStream related page: http://graphstream-project.org/] The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs ,

```
let my_graph type: graph <- load_graph_from_dgs_old( [
            "filename"::"../includes/BarabasiGenerated.dgs",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_dot

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following DOT file format.
- Comment: DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. See: http://en.wikipedia.org/wiki/DOT_language for more details.The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_dot( [
            "filename"::"example_of_dot_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_edge

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following Edge file format.
- Comment: This format is a simple text file with numeric vertex ids defining the edges. The map operand should includes following elements:
- Special cases:

  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_edge( [
            "filename"::"example_of_edge_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_gexf

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following GEXF file format.
- Comment: GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications. See: http://gexf.net/format/ for more details.The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_gexf( [
            "filename"::"example_of_Gexf_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_graphml

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following GEXF file format.
- Comment: GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details.The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_graphml( [
            "filename"::"example_of_Graphml_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_lgl

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following LGL file format.
- Comment: LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details.The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_lgl( [
            "filename"::"example_of_LGL_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_ncol

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following ncol file format.
- Comment: This format is used by the Large Graph Layout progra. It is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge. See: http://bioinformatics.icmb.utexas.edu/lgl for more details.The map operand should includes following elements:
- Special cases:
  - "filename": the filename of the file containing the network
  - "edges_specy": the species of edges
  - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_ncol( [
            "filename"::"example_of_ncol_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_pajek

- Possible use:
  - OP(map) --- > graph
- Result: returns a graph loaded from a given file following Pajek file format.

- Comment: Pajek (Slovene word for Spider) is a program, for Windows, for analysis and visualization of large networks. See: http://pajek.imfm.si/doku.php?id=pajek for more details.The map operand should includes following elements:
- Special cases:
    - "filename": the filename of the file containing the network
    - "edges_specy": the species of edges
    - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_pajek( [
            "filename"::"example_of_Pajek_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

load_graph_from_tlp

- Possible use:
    - OP(map) --- >   graph
- Result: returns a graph loaded from a given file following TLP file format.
- Comment: TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details.The map operand should includes following elements:
- Special cases:
    - "filename": the filename of the file containing the network
    - "edges_specy": the species of edges
    - "vertices_specy": the species of vertices
- See also: load_graph_from_dgs_old ,

```
let my_graph type: graph <- load_graph_from_tlp( [
            "filename"::"example_of_TLP_file",
            "edges_specy"::edgeSpecy,
            "vertices_specy"::nodeSpecy] );
```

Top of the page

map

- Possible use:
    - OP(container) --- >   map
    - OP(any) --- >   map
- Result: casting of the operand to a map.
- Special cases:
    - if the operand is a color RRGGBB, returns a map with the three elements: "r"::RR , "g"::GG , "b"::BB ;
    - if the operand is a point, returns a map with two elements: "x":: x -ccordinate and "y":: y -coordinate;
    - if the operand is pair, returns a map with this only element;
    - if the operand is a species name, returns the map containing all the agents of the species as a pair nom_agent::agent ;
    - if the operand is a agent, returns a map containing all the attributes as a pair attribute_name::attribute_value ;

- if the operand is a list, returns a map containing either elements of the list if it is a list of pairs, or pairs list.get(i) ::list .get(i+1);
- if the operand is a file, returns the content casted to map;
- if the operand is a graph, returns the a map with pairs edge_source::edge_target ;
- otherwise returns a map containing only the pair operand::operand .

Top of the page

masked_by

- Possible use:
  - shape OP list of agents --- > shape
  - shape OP species --- > shape
- Result: A geometry representing the part of the right operand visible from the point of view of the agent using the operator while considering the obstacles defined by the left operand

```
perception_geom masked_by obstacle_list --: returns the geometry representing the part of
perception_geom visible from the agent position considering the list of obstacles obstacle_list.
perception_geom masked_by obstacle_species --: returns the geometry representing the part
of perception_geom visible from the agent position considering the obstacles of species
obstacle_species.
```

Top of the page

matrix

- Possible use:
  - OP(container) --- > matrix
  - OP(any) --- > matrix
  - OP(list) --- > matrix
- Result: casts the operand into a matrix
- Special cases:
  - if the operand is a file, returns its content casted as a matrix
  - if the operand is a map, returns a 2-columns matrix with keyx in the first one and value in the second one;
  - if the operand is a list, returns a 1-row matrix. Notice that each element of the list should be a single element or lists with the same length;
  - if the operand is a graph, returns nil;
  - otherwise, returns a 1x1 matrix with the operand at the (0,0) position.
- See also: as_matrix ,

Top of the page

max

- Possible use:
  - OP(container) --- > [ValueType]
- Result: the maximum element found in the operand
- Comment: the max operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list of int of float, max returns the maximum of all the elements

  - if it is a list of points: max returns the maximum of all points as a point (i.e. the point with the greatest coordinate on the x-axis, in case of equality the point with the greatest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)
  - if it is a population of a list of other type: max transforms all elements into integer and returns the maximum of them
  - if it is a map, max returns the maximum among the list of all elements value
  - if it is a file, max returns the maximum of the content of the file (that is also a container)
  - if it is a graph, max returns the maximum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
  - if it is a matrix of int, float or object, max returns the maximum of all the numerical elements (thus all elements for integer and float matrices)
  - if it is a matrix of geometry, max returns the maximum of the list of the geometries
  - if it is a matrix of another type, max returns the maximum of the elements transformed into float

- See also: min ,

```
max ([100, 23.2, 34.5])              --:     100.0
max([{1.0;3.0},{3.0;5.0},{9.0;1.0},{7.0;8.0}])     --:  {9.0;1.0}
```

Top of the page

max_of

- Possible use:
  - container OP any expression --- > any
- Result: the maximum value of the right-hand expression evaluated on each of the elements of the left-hand operand
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is nil, max_of returns the right-hand operand default value
- See also: min_of ,

```
[1,2,4,3,5,7,6,8] max_of (each * 100 )              --:     800
g2 max_of (length(g2 out_edges_of each) )              --:     3
(list(node) max_of (round(node(each).location.x))  --:     96
[1::2, 3::4, 5::6] max_of (each.value + 3)         --:     9
```

Top of the page

mean

- Possible use:
  - OP(container) --- > any
- Result: the mean of all the elements of the operand
- Comment: the elements of the operand are summed (see sum for more details about the sum of container elements) and then the sum value is divided by the number of elements.
- Special cases:
  - if the container contains points, the result will be a point
- See also: sum ,

```
mean ([4.5, 3.5, 5.5, 7.0]) --: 5.125
```

Top of the page

mean_deviation

- Possible use:
  - OP(list) --- > float
- Result: the deviation from the mean of all the elements of the operand. See Mean_deviation < http://en.wikipedia.org/wiki/Absolute_deviation> for more details.
- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean , standard_deviation ,

```
mean_deviation ([4.5, 3.5, 5.5, 7.0]) --: 1.125
```

Top of the page

meanR

- Possible use:
  - OP(container) --- > any

Top of the page

median

- Possible use:
  - OP(list) --- > float
- Result: the median of all the elements of the operand.
- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean ,

```
median ([4.5, 3.5, 5.5, 7.0]) --: 5.0
```

Top of the page

min

- Possible use:
  - OP(container) --- > [ValueType]
- Result: the minimum element found in the operand.
- Comment: the min operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list of int or float: min returns the minimum of all the elements
  - if it is a list of points: min returns the minimum of all points as a point (i.e. the point with the smallest coordinate on the x-axis, in case of equality the point with the smallest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)
  - if it is a population of a list of other types: min transforms all elements into integer and returns the minimum of them
  - if it is a map, min returns the minimum among the list of all elements value
  - if it is a file, min returns the minimum of the content of the file (that is also a container)
  - if it is a graph, min returns the minimum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)

    - if it is a matrix of int, float or object, min returns the minimum of all the numerical elements (thus all elements for integer and float matrices)
    - if it is a matrix of geometry, min returns the minimum of the list of the geometries
    - if it is a matrix of another type, min returns the minimum of the elements transformed into float
- See also: max ,

```
min ([100, 23.2, 34.5])     --:     23.2
```

Top of the page

min_of

- Possible use:
    - container OP any expression --- >   any
- Result: the minimum value of the right-hand expression evaluated on each of the elements of the left-hand operand
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
    - if the left-hand operand is nil, first_with returns nil
- See also: max_of ,

```
[1,2,4,3,5,7,6,8] min_of (each * 100 )            --:     100
g2 min_of (length(g2 out_edges_of each) )            --:     0
(list(node) min_of (round(node(each).location.x))  --:     4
[1::2, 3::4, 5::6] min_of (each.value + 3)              --:     5
```

Top of the page

mod

- Possible use:
    - int OP int --- >   int
- Result: an int, equal to the remainder of the integer division of the left-hand operand by the rigth-hand operand.
- Special cases:
    - if the right-hand operand is equal to zero, raises an exception.
- See also: div ,

```
40 mod 3          --:  1
40 mod 4         --:  0
```

Top of the page

mul

- Possible use:
    - OP(container) --- >   any
- Result: the product of all the elements of the operand
- Comment: the mul operator behavior depends on the nature of the operand
- Special cases:
    - if it is a list of int or float: mul returns the product of all the elements

- if it is a list of points: mul returns the product of all points as a point (each coordinate is the product of the corresponding coordinate of each element)
- if it is a list of other types: mul transforms all elements into integer and multiplies them
- if it is a map, mul returns the product of the value of all elements
- if it is a file, mul returns the product of the content of the file (that is also a container)
- if it is a graph, mul returns the product of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, mul returns the product of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, mul returns the product of the list of the geometries
- if it is a matrix of other types: mul transforms all elements into float and multiplies them
- See also: sum ,

```
mul ([100, 23.2, 34.5])     --:        80040.0
```

Top of the page

neighbours_at

- Possible use:
  - point OP float --- > list
  - shape OP float --- > list
- Result: a list, containing all the agents located at a distance inferior or equal to the right-hand operand to the left-hand operand (geometry, agent, point).
- Comment: The topology used to compute the neighbourhood is the one of the left-operand if this one is an agent; otherwise the one of the agent applying the operator.
- See also: neighbours_of , closest_to , overlapping , agents_overlapping , agents_inside , agent_closest_to , at_distance ,

```
(self neighbours_at (10)) --: returns all the agents located at a distance lower or equal to 10
to the agent applying the operator.
```

Top of the page

neighbours_of

- Possible use:
  - topology OP pair --- > list
  - topology OP agent --- > list
  - graph OP any --- > list
- Result: a list, containing all the agents located at a distance inferior or equal to 1 to the right-hand operand agent considering the left-hand operand topology.returns the list of neighbours of the given vertex (right-hand operand) in the given graph (left-hand operand)
- Special cases:
  - a list, containing all the agents located at a distance inferior or equal to the right member (float) of the pair (right-hand operand) to the left member (agent, geometry or point) considering the left-hand operand topology.
- See also: neighbours_at , closest_to , overlapping , agents_overlapping , agents_inside , agent_closest_to , predecessors_of , successors_of ,

```
topology(self) neighbours_of self::10--: returns all the agents located at a distance lower or
equal to 10 to the agent applying the operator considering its topology.
topology(self) neighbours_of self --: returns all the agents located at a distance lower or equal
to 1 to the agent applying the operator considering its topology.
```

```
graphEpidemio neighbours_of (node(3))         --:    [node0,node2]
graphFromMap neighbours_of node({12,45})     --: [{1.0;5.0},{34.0;56.0}]
```

Top of the page

new_folder

- Possible use:
  - OP(string) --- > file
- Result: opens an existing repository or create a new folder if it does not exist.
- Special cases:
  - If the specified string does not refer to an existing repository, the repository is created. If the string refers to an existing file, an exception is risen.
- See also: folder , file ,

```
let dirNewT type: file value: new_folder("../incl/");       // dirNewT represents the repository
"../incl/"
                                                             // eventually creates the
directory ../incl
```

Top of the page

norm

- Possible use:
  - OP(point) --- > float
- Result: the norm of the vector with the coordinnates of the point operand.

```
norm({3,4})   --:      5.0
```

Top of the page

not

Same signification as ! operator.

Top of the page

of

Same signification as . operator.

Top of the page

of_generic_species

- Possible use:
  - container OP species --- > list
- Result: a list, containing the agents of the left-hand operand whose species is that denoted by the right-hand operand and whose species extends the right-hand operand species
- See also: of_species ,

```
// species test {}
```

```
// species sous_test parent: test {}
[sous_test(0),sous_test(1),test(2),test(3)] of_generic_species test         --:
[sous_test0,sous_test1,test2,test3]
[sous_test(0),sous_test(1),test(2),test(3)] of_generic_species sous_test     --:
[sous_test0,sous_test1]
[sous_test(0),sous_test(1),test(2),test(3)] of_species test                 --: [test2,test3]
[sous_test(0),sous_test(1),test(2),test(3)] of_species sous_test             --:
[sous_test0,sous_test1]
```

Top of the page

of_species

- Possible use:
  - container OP species --- > list
- Result: a list, containing the agents of the left-hand operand whose species is that denoted by the right-hand operand.The expression agents of_species (species self) is equivalent to agents where (species each = species self); however, the advantage of using the first syntax is that the resulting list is correctly typed with the right species, whereas, in the second syntax, the parser cannot determine the species of the agents within the list (resulting in the need to cast it explicitely if it is to be used in an ask statement, for instance).
- Special cases:
  - if the right operand is nil, of_species returns the right operand
- See also: of_generic_species ,

```
(self neighbours_at 10) of_species (species (self))     --:  all the neighbouring agents of the
same species.
[test(0),test(1),node(1),node(2)] of_species test         --:  [test0,test1]
[1,2,3,4,5,6] of_species test                            --:     []
```

Top of the page

one_of

- Possible use:
  - OP(container) --- > [ValueType]
  - OP(species) --- > agent
- Result: one of the values stored in this container using GAMA.getRandom()a random element from the list
- Comment: the one_of operator behavior depends on the nature of the operand
- Special cases:
  - if the operand is empty, one_of returns nil
  - if it is a list or a matrix, one_of returns one of the elements of the list or of the matrix
  - if it is a map, one_of returns one of the values of the map
  - if it is a graph, one_of returns one of the nodes of the graph
  - if it is a file, one_of returns one of the elements of the content of the file (that is also a container)
  - if it is a population, one_of returns one of the agents of the population
  - if the list is empty, returns nil
  - If the operand is a species, the operand is casted to a list before the expression is evaluated. Therefore, if foo is the name of a species, any(foo) will return a random agent from this species (see list)

```
any ([1,2,3])         --:     1, 2, or 3
one_of ([1,2,3])     --:         1, 2, or 3
```

```
one_of ([2::3, 4::5, 6::7])     --:     3, 5 or 7
// The species bug has previously been defined
one_of (bug)         --:         bug3
let mat3 type:matrix value: matrix([["c11","c12","c13"],["c21","c22","c23"]])
one_of(mat3)         --:      "c11","c12","c13", "c21","c22" or "c23"
one_of (bug) --: bug3     // The species `bug` has previously be defined
```

Top of the page

or

- Possible use:
  - bool OP any expression --- > bool
- Result: a bool value, equal to the logical or between the left-hand operand and the rigth-hand operand.
- Comment: both operands are always casted to bool before applying the operator. Thus, an expression like 1 or 0 is accepted and returns true.
- See also: bool , and ,

Top of the page

out_degree_of

- Possible use:
  - graph OP any --- > int
- Result: returns the out degree of a vertex (right-hand operand) in the graph given as left-hand operand.
- See also: in_degree_of , degree_of ,

```
graphEpidemio out_degree_of (node(3))
```

Top of the page

out_edges_of

- Possible use:
  - graph OP any --- > list
- Result: returns the list of the out-edges of a vertex (right-hand operand) in the graph given as left-hand operand.
- See also: in_edges_of ,

```
graphEpidemio out_edges_of (node(3))
```

Top of the page

overlapping

- Possible use:
  - container of shapes OP any --- > list of agents
  - species OP any --- > list of agents
- Result: A list of agents among the left-operand list, overlapping the operand (casted as a geometry).
- Special cases:
  - if the left-operand is a species, return agents of the specified species.

- See also: neighbours_at , neighbours_of , agent_closest_to , agents_inside , closest_to , inside , agents_overlapping ,

```
[ag1, ag2, ag3] overlapping(self) --: return the agents among ag1, ag2 and ag3 that overlap the shape of the agent applying the operator.
species1 overlapping(self) --: return the agents of species species1 that overlap the shape of the agent applying the operator.
```

Top of the page

overlaps

- Possible use:
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) overlaps the right-geometry (or agent/point).
- Special cases:
  - if one of the operand is null, returns false.
  - if one operand is a point, returns true if the point is included in the geometry
- See also: <--: , disjoint_from , crosses , intersects , partially_overlaps , touches ,

```
polyline([{10,10},{20,20}]) overlaps polyline([{15,15},{25,25}]) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},{25,25},{25,15}])
--: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps geometry({25,25}) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{35,35},{35,45},{45,45},{45,35}])
--: false
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polyline([{10,10},{20,20}]) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps geometry({15,15}) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{0,0},{0,30},{30,30}, {30,0}]) --:
true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},{25,25},{25,15}])
--: true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{10,20},{20,20},{20,30},{10,30}])
--: true
```

Top of the page

pair

- Possible use:
  - OP(any) --- > pair
- Result: casting of the operand to a pair value.
- Special cases:
  - if the operand is null, returns null;
  - if the operand is a point, returns the pair x- coordinate::y -coordinate;
  - if the operand is a particular kind of geometry, a link between geometry, returns the pair formed with these two geoemtries;
  - if the operand is a map, returns the pair where the first element is the list of all the keys of the map and the second element is the list of all the values of the map;
  - if the operand is a list, returns a pair with the two first element of the list used to built the pair
  - if the operand is a link, returns a pair source_link::destination_link
  - Otherwise, returns the pair string(operand) ::operand .

```
pair(true)              --: true::true
pair({23, 4.0}              --: 23.0::4.0
```

```
pair([1,5,9,3])         --: 1::5
pair([[3,7],[2,6,9],0])     --: [3,7]::[2,6,9]
pair(['a'::345, 'b'::13, 'c'::12])      --: [b,c,a]::[13,12,345]
```

Top of the page

partially_overlaps

- Possible use:
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) partially overlaps the right-geometry (or agent/point).
- Comment: if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).
- Special cases:
  - if one of the operand is null, returns false.
- See also: <--: , disjoint_from , crosses , overlaps , intersects , touches ,

```
polyline([{10,10},{20,20}]) partially_overlaps polyline([{15,15},{25,25}]) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{15,15},{15,25},{25,25},
{25,15}]) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps geometry({25,25}) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{35,35},{35,45},{45,45},
{45,35}]) --: false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polyline([{10,10},{20,20}]) --:
false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps geometry({15,15}) --: false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{0,0},{0,30},{30,30},
{30,0}]) --: false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{15,15},{15,25},{25,25},
{25,15}]) --: true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{10,20},{20,20},{20,30},
{10,30}]) --: false
```

Top of the page

path

- Possible use:
  - OP(any) --- > path
- Result: casting of the operand to a path
- Special cases:
  - if the operand is a path, returns itself
  - if the operand is a list, casts the list into a list of point and returns the path (in the current topology) through these points.
  - otherwise, returns nil
- See also: graph ,

```
path([{2,5}, {4,7}, {2,1}])   --:  [polyline ([{2.0,5.0},{4.0,7.0}]),polyline ([{4.0,7.0},
{2.0,1.0}])]
```

Top of the page

path_between

- Possible use:
  - topology OP container of shapes --- > path
- Result: A path between a list of two geometries (geometries, agents or points) considering a topology.
- See also: towards , direction_to , distance_between , direction_between , path_to , distance_to ,

```
my_topology path_between [ag1, ag2] --: A path between ag1 and ag2
```

Top of the page

path_to

- Possible use:
  - shape OP shape --- > path
  - point OP point --- > path
- Result: A path between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.
- See also: towards , direction_to , distance_between , direction_between , path_between , distance_to ,

```
ag1 path_to ag2 --: the path between ag1 and ag2 considering the topology of the agent applying
the operator
```

Top of the page

point

- Possible use:
  - OP(any) --- > point
- Result: casting of the operand to a point value.
- Special cases:
  - if the operand is null, returns null;
  - if the operand is an agent, returns its location
  - if the operand is a geometry, returns its centroid
  - if the operand is a list with at least two elements, returns a point with the two first elements of the list (casted to float)
  - if the operand is a map, returns the point with values associated respectively with keys "x" and "y"
  - if the operand is a pair, returns a point with the two elements of the pair (casted to float)
  - otherwise, returns a point {val,val} where val is the float value of the operand

```
point(0)             --: {0.0;0.0}
point(true)          --: {1.0;1.0}
point(5::34)          --: {5.0;34.0}
point([1,5,9,3])      --: {1.0;5.0}
point([[3,7],[2,6,9],0])      --:{0.0;0.0}
point(['a'::345, 'y'::13, 'c'::12])      --:  {0.0;13.0}
point(node1)                --: {64.06165572529225;18.401233796267537}   // centroid of node1
shape
```

Top of the page

points_at

- Possible use:
  - int OP float --- > list
- Result: A list of left-operand number of points located at a the right-operand distance to the agent location.
- See also: any_location_in , any_point_in , closest_points_with , farthest_point_to ,

```
3 points_at(20.0) --: returns [pt1, pt2, pt3] with pt1, pt2 and pt3 located at a distance of 20.0
to the agent location
```

Top of the page

poisson

- Possible use:
  - OP(float) --- > int
- Result: A value from a random variable following a Poisson distribution (with the positive expected number of occurence lambda as operand).
- Comment: The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event, cf. Poisson distribution on Wikipedia.
- See also: binomial , gauss ,

```
poisson(3.5) --: a random positive integer
```

Top of the page

polygon

- Possible use:
  - OP(list of points) --- > shape
- Result: A polygon geometry from the given list of points.
- Special cases:
  - if the operand is nil, returns the point geometry {0,0}
  - if the operand is composed of a single point, returns a point geometry
  - if the operand is composed of 2 points, returns a polyline geometry.
- See also: around , circle , cone , line , link , norm , point , polyline , rectangle , square , triangle ,

```
polygon([{0,0}, {0,10}, {10,10}, {10,0}]) --: returns a polygon geometry composed of the 4
points.
```

Top of the page

polyline

Same signification as line operator.

Top of the page

predecessors_of

- Possible use:
  - graph OP any --- > list
- Result: returns the list of predecessors (i.e. sources of in edges) of the given vertex (right-hand operand) in the given graph (left-hand operand)
- See also: neighbours_of , successors_of ,

```
graphEpidemio predecessors_of (node(3))         --: [node0,node2]
graphFromMap predecessors_of node({12,45})     --:    [{1.0;5.0}]
```

Top of the page

product

Same signification as mul operator.

Top of the page

properties

- Possible use:
  - OP(string) --- > file
- Result: opens a file that is a kind of properties.
- Comment: The file should have a properties extension, cf. type file definition for supported file extensions.
- Special cases:
  - If the specified string does not refer to an existing propserites file, an exception is risen.
- See also: file , shapefile , image , text ,

```
let fileT type: file value: properties("../includes/testProperties.properties");  // fileT
represents the properties file "../includes/testProperties.properties"
```

Top of the page

R_compute

- Possible use:
  - OP(string) --- > map

Top of the page

read

- Possible use:
  - OP(file) --- > any
  - OP(string) --- > any
- Result: marks the file so that only read operations are allowed.
- Comment: A file is created by default in read-only mode. The operator write can change the mode.
- See also: file , write ,

```
read(shapefile("../images/point_eau.shp"))  --:  returns a file in read-only mode representing
"../images/point_eau.shp"
```

Top of the page

rectangle

- Possible use:
  - OP(point) --- > shape
- Result: A rectangle geometry which side sizes are given by the operands.
- Comment: the centre of the rectangle is by default the location of the current agent in which has been called this operator.
- Special cases:
  - returns nil if the operand is nil.
- See also: around , circle , cone , line , link , norm , point , polygon , polyline , square , triangle ,

```
rectangle({10, 5}) --: returns a geometry as a rectangle with width = 10 and heigh = 5.
```

Top of the page

reduced_by

Same signification as - operator.

Top of the page

remove_duplicates

- Possible use:
  - OP(container) --- > list
- Result: produces a set from the elements of the operand (i.e. a list without duplicated elements)
- Special cases:
  - if the operand is nil, remove_duplicates returns nil
  - if the operand is a graph, remove_duplicates returns the set of nodes
  - if the operand is a map, remove_duplicates returns the set of values without duplicate
  - if the operand is a matrix, remove_duplicates returns a matrix withtout duplicated row

```
remove_duplicates([3,2,5,1,2,3,5,5,5]) --: [3,2,5,1]
remove_duplicates([1::3,2::4,3::3,5::7]) --: [3,4,7]
```

Top of the page

remove_node_from

- Possible use:
  - shape OP graph --- > graph
- Result: removes a node from a graph.
- Comment: all the edges containing this node are also removed.

```
node(0) remove_node_from graphEpidemio;    --:     returns the graph without node(0)
```

Top of the page

reverse

- Possible use:
  - OP(string) --- > string
  - OP(container) --- > container
- Result: the operand elements in the reversed order in a copy of the operand.
- Comment: the reverse operator behavior depends on the nature of the operand
- Special cases:
  - if it is a string, reverse returns a new string with caracters in the reversed order
  - if it is a list, reverse returns a copy of the operand list with elements in the reversed order
  - if it is a map, reverse returns a copy of the operand map with each pair in the reversed order (i.e. all keys become values and values become keys)
  - if it is a file, reverse returns a copy of the file with a reversed content
  - if it is a population, reverse returns a copy of the population with elements in the reversed order
  - if it is a graph, reverse returns a copy of the graph (with all edges and vertexes), with all of the edges reversed
  - if it is a matrix, reverse returns a new matrix containing the transpose of the operand.

```
reverse ('abcd')         --:     'dcba';
reverse ([10,12,14])                 --:     [14, 12, 10]
reverse ([k1::44, k2::32, k3::12]) --:     [12::k3,  32::k2, 44::k1]
```

Top of the page

rewire_n

- Possible use:
  - graph OP int --- > graph
- Result: rewires the given count of edges.
- Comment: If there are too many edges, all the edges will be rewired.
- See also: rewire_p ,

```
set graphEpidemio <- graphEpidemio rewire_n 10;
```

Top of the page

rgb

- Possible use:
  - OP(any) --- > rgb
- Result: casting of the operand to a rgb color.
- Special cases:
  - if the operand is nil, returns white;
  - if the operand is a string, the allowed color names are the constants defined in the java.awt.Color class, i.e.: black, blue, cyan, darkGray, lightGray, gray, green, magenta, orange, pink, red, white, yellow. Otherwise tries to cast the string to an int and returns this color
  - if the operand is a list, the integer value associated to the three first elements of the list are used to define the three red (element 0 of the list), green (element 1 of the list) and blue (element 2 of the list) components of the color;
  - if the operand is a map, the red, green, blue components take the value associated to the keys "r", "g", "b" in the map;

- if the operand is a matrix, return the color of the matrix casted as a list;
- if the operand is a boolean, returns black for true and white for false;
- if the operand is an integer value, the decimal integer is translated into a hexadecimal value: OxRRGGBB. The red (resp. green, blue) component of the color take the value RR (resp. GG, BB) translated in decimal.

```
rgb(3.78)         --: rgb([0,0,3])
rgb(true)         --: rgb([0,0,0]) //black
rgb({23, 4.0}         --: rgb([0,0,0]) //black
rgb(5::34)         --: rgb([0,0,0]) //black
rgb(green)         --: rgb([0,255,0]) //green
rgb([1,5,9,3])             --: rgb([1,5,9])
rgb(node1)         --: rgb([0,0,1])
rgb('4')        --: rgb([0,0,4])
rgb('4.7')         --:  // Exception
```

Top of the page

rnd

- Possible use:
  - OP(int) --- > int
  - OP(point) --- > point
  - OP(float) --- > int
- Result: a random integer in the interval [0, operand]
- Comment: to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision
- Special cases:
  - if the operand is a point, returns a point with two random integers in the interval [0, operand]
  - if the operand is a float, it is casted to an int before being evaluated
- See also: flip ,

```
rnd (2) --: 0, 1 or 2
rnd (1000) / 1000 --: a float between 0 and 1 with a precision of 0.001
rnd ({2.5,3}) --: {x,y} with x in [0,2] and y in [0,3]
rnd (2.5) --: 0, 1 or 2
```

Top of the page

rotated_by

- Possible use:
  - shape OP int --- > shape
  - shape OP float --- > shape
- Result: A geometry resulting from the application of a rotation by the right-hand operand angle (degree) to the left-hand operand (geometry, agent, point)
- Comment: the right-hand operand can be a float or a int
- See also: transformed_by , translated_by ,

```
self rotated_by 45 --: returns the geometry resulting from a 45 degres rotation to the geometry
of the agent applying the operator.
```

Top of the page

round

- Possible use:
  - OP(float) --- > int
  - OP(int) --- > int
- Result: the rounded value of the operand.
- Special cases:
  - if the operand is an int, round returns it
- See also: int , with_precision ,

```
round (0.51)     --:     1
round (100.2)     --:     100
```

Top of the page

row_at

- Possible use:
  - matrix OP int --- > list
- Result: returns the row at a num_line (rigth-hand operand)
- See also: column_at , columns_list ,

```
matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]) row_at 2  --:
["el13","el23","el33"]
```

Top of the page

rows_list

- Possible use:
  - OP(matrix) --- > list of lists
- Result: returns a list of the rows of the matrix, with each row as a list of elements
- See also: columns_list ,

```
rows_list(matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]])   --:
        [["el11","el21","el31"],["el12","el22","el32"],["el13","el23","el33"]]
```

Top of the page

scaled_by

Same signification as * operator.

Top of the page

select

Same signification as where operator.

Top of the page

set_verbose

- Possible use:
  - graph OP bool --- > graph
- Result: sets the verbose attributes of the graph (left-hand operand) to the given boolean value (right-hand operand).
- Comment: When verbose of a graph is true, it will display the shortest path computation level with static optimizer. This operator is useful to monitor the computation of
- See also: with_optimizer_type ,

```
set graphEpidemio <- graphEpidemio set_verbose false;
```

Top of the page

shapefile

- Possible use:
  - OP(string) --- > file
- Result: opens a file that a is a kind of shapefile.
- Comment: The file should have a shapefile extension, cf. file type definition for supported file extensions.
- Special cases:
  - If the specified string does not refer to an existing shapefile file, an exception is risen.
- See also: file , properties , image , text ,

```
let fileT type: file value: shapefile("../includes/testProperties.shp");
            // fileT represents the shapefile file "../includes/testProperties.shp"
```

Top of the page

shuffle

- Possible use:
  - OP(matrix) --- > matrix
  - OP(list) --- > list
  - OP(species) --- > list
  - OP(string) --- > string
- Result: The elements of the operand in random order.
- Special cases:
  - if the operand is empty, returns an empty list (or string, matrix)
- See also: reverse ,

```
shuffle ([["c11","c12","c13"],["c21","c22","c23"]]) --: [["c12","c21","c11"],["c13","c22","c23"]]
shuffle ([12, 13, 14]) --: [14,12,13];
shuffle (bug) --:  shuffle the list of all agents of the `bug` species
shuffle ('abc') --: 'bac'
```

Top of the page

simple_clustering_by_distance

- Possible use:

- list of agents OP float --- > list
- Result: A list of agent groups clustered by distance considering a distance min between two groups.
- Comment: use of hierarchical clustering with Minimum for linkage criterion between two groups of agents.
- See also: simple_clustering_by_envelope_distance ,

```
[ag1, ag2, ag3, ag4, ag5] simpleClusteringByDistance 20.0 --: for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

Top of the page

simple_clustering_by_envelope_distance

- Possible use:
  - list of agents OP float --- > list
- Result: A list of agent groups clustered by distance (considering the agent envelop) considering a distance min between two groups.
- Comment: use of hierarchical clustering with Minimum for linkage criterion between two groups of agents.
- See also: simple_clustering_by_distance ,

```
[ag1, ag2, ag3, ag4, ag5] simpleClusteringByDistance 20.0 --: for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

Top of the page

simplification

- Possible use:
  - shape OP float --- > shape
- Result: A geometry corresponding to the simplification of the operand (geometry, agent, point) considering a tolerance distance.
- Comment: The algorithm used for the simplification is Douglas-Peucker

```
self simplification 0.1 --: returns the geometry resulting from the application of the Douglas-Peuker algorithm on the geometry of the agent applying the operator with a tolerance distance of 0.1.
```

Top of the page

sin

- Possible use:
  - OP(float) --- > float
  - OP(int) --- > float
- Result: the sinus of the operand (in decimal degrees).
- Special cases:
  - the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- See also: cos , tan ,

```
cos (0) --: 0
```

Top of the page

skeletonize

- Possible use:
  - OP(shape) --- > list of shapes
- Result: A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent)

```
skeletonize(self) --: returns the list of geometries corresponding to the skeleton of the
geometry of the agent applying the operator.
```

Top of the page

solid

Same signification as without_holes operator.

Top of the page

sort

Same signification as sort_by operator.

Top of the page

sort_by

- Possible use:
  - container OP any expression --- > list
  - map OP any expression --- > map
- Result: a list, containing the elements of the left-hand operand sorted in ascending order by the value of the right-hand operand when it is evaluated on them.
- Comment: the left-hand operand is casted to a list before applying the operator. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.
- Special cases:
  - if the left-hand operand is nil, sort_by returns nil
- See also: group_by ,

```
[1,2,4,3,5,7,6,8] sort_by (each)                   --:     [1,2,3,4,5,6,7,8]
g2 sort_by (length(g2 out_edges_of each) )         --:     [node9, node7, node10, node8, node11,
node6, node5, node4]
(list(node) sort_by (round(node(each).location.x)) --:      [node5, node1, node0, node2, node3]
[1::2, 3::4, 5::6] sort_by (each)                  --:
```

Top of the page

source_of

- Possible use:
  - graph OP any --- > any
- Result: returns the source of the edge (right-hand operand) contained in the graph given in left-hand operand.
- Special cases:
  - if the lef-hand operand (the graph) is nil, throws an Exception

- See also: target_of ,

```
let graphEpidemio type: graph <-
generate_barabasi_albert( ["edges_specy"::edge,"vertices_specy"::node,"size"::3,"m"::5] );
graphEpidemio source_of(edge(3))                    --:  node1
let graphFromMap type: graph <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
graphFromMap source_of(link({1,5}::{12,45}))      --: {1.0;5.0}
```

Top of the page

species

- Possible use:
  - OP(any) --- >   species
- Result: casting of the operand to a species.
- Special cases:
  - if the operand is nil, returns nil;
  - if the operand is an agent, returns its species;
  - if the operand is a string, returns the species with this name (nil if not found);
  - otherwise, returns nil

```
species(self)            --: species of the current agent
species('node')        --: node
species([1,5,9,3])     --: null
species(node1)            --: node
```

Top of the page

species_of

Same signification as species operator.

Top of the page

split_at

- Possible use:
  - shape OP point --- >   list of shapes
- Result: The two part of the left-operand lines split at the given right-operand point
- Special cases:
  - if the left-operand is a point or a polygon, returns an empty list

```
polyline([{1,2},{4,6}]) split_at {7,6}  --: [polyline([{1.0;2.0},{7.0;6.0}]),
polyline([{7.0;6.0},{4.0;6.0}])].
```

Top of the page

split_lines

- Possible use:
  - OP(list) --- >   list of shapes
- Result: A list of geometries resulting after cutting the lines at their intersections.

```
split_lines([line([{0,10}, {20,10}], line([{0,10}, {20,10}]]) --: returns a list of four
polylines: line([{0,10}, {10,10}]), line([{10,10}, {20,10}]), line([{10,0}, {10,10}]) and
line([{10,10}, {10,20}]).
```

Top of the page

split_with

- Possible use:
  - string OP string --- > list
- Result: a list, containing the sub-strings (tokens) of the left-hand operand delimited by each of the characters of the right-hand operand.
- Comment: delimiters themselves are excluded from the resulting list

```
'to be or not to be,that is the question' split_with ' ,'  --:
[to,be,or,not,to,be,that,is,the,question]
```

Top of the page

sqrt

- Possible use:
  - OP(int) --- > float
  - OP(float) --- > float
- Result: returns the square root of the operand.
- Special cases:
  - if the operand is negative, an exception is raised

```
sqrt(4)     --:     2.0
```

Top of the page

square

- Possible use:
  - OP(float) --- > shape
- Result: A square geometry which side size is equal to the operand.
- Comment: the centre of the square is by default the location of the current agent in which has been called this operator.
- Special cases:
  - returns nil if the operand is nil.
- See also: around , circle , cone , line , link , norm , point , polygon , polyline , rectangle , triangle ,

```
square(10) --: returns a geometry as a square of side size 10.
```

Top of the page

standard_deviation

- Possible use:
  - OP(list) --- > float
- Result: the standard deviation on the elements of the operand. See Standard_deviation < http://en.wikipedia.org/wiki/Standard_deviation> for more details.

- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean , mean_deviation ,

```
standard_deviation ([4.5, 3.5, 5.5, 7.0]) --: 1.2930100540985752
```

Top of the page

string

- Possible use:
  - OP(any) --- > string
- Result: casting of the operand to a string.
- Special cases:
  - if the operand is nil, returns 'nil';
  - if the operand is an agent, returns its name;
  - if the operand is a string, returns the operand;
  - if the operand is an int or a float, returns their string representation (as in Java);
  - if the operand is a boolean, returns 'true' or 'false';
  - if the operand is a species, returns its name;
  - if the operand is a color, returns its litteral value if it has been created with one (i.e. 'black', 'green', etc.) or the string representation of its hexadecimal value.
  - if the operand is a container, returns its string representation.

```
string(0)              --: 0
string({23, 4.0}      --: {23.0;4.0}
string(5::34)          --: 5::34
string(['a'::345, 'b'::13, 'c'::12])  --:  b,13; c,12; a,345;
```

Top of the page

successors_of

- Possible use:
  - graph OP any --- > list
- Result: returns the list of successors (i.e. targets of out edges) of the given vertex (right-hand operand) in the given graph (left-hand operand)
- See also: predecessors_of , neighbours_of ,

```
graphEpidemio successors_of (node(3))         --: []
graphFromMap successors_of node({12,45})     --: [{34.0;56.0}]
```

Top of the page

sum

- Possible use:
  - OP(container) --- > any
- Result: the sum of all the elements of the operand
- Comment: the sum operator behavior depends on the nature of the operand
- Special cases:
  - if it is a list of int or float: sum returns the sum of all the elements

    - if it is a list of points: sum returns the sum of all points as a point (each coordinate is the sum of the corresponding coordinate of each element)
    - if it is a population or a list of other types: sum transforms all elements into integer and sums them
    - if it is a map, sum returns the sum of the value of all elements
    - if it is a file, sum returns the sum of the content of the file (that is also a container)
    - if it is a graph, sum returns the sum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
    - if it is a matrix of int, float or object, sum returns the sum of all the numerical elements (i.e. all elements for integer and float matrices)
    - if it is a matrix of geometry, sum returns the sum of the list of the geometries
    - if it is a matrix of other types: sum transforms all elements into float and sums them
- See also: mul ,

```
sum ([12,10, 3])     --:     25.0
sum([{1.0;3.0},{3.0;5.0},{9.0;1.0},{7.0;8.0}])       --: {20.0;17.0}
```

Top of the page

tan

- Possible use:
    - OP(float) --- >   float
    - OP(int) --- >   float
- Result: the trigonometic tangent of the operand (in decimal degrees).
- Special cases:
    - the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- See also: cos , sin ,

```
cos (180) --: 0
```

Top of the page

tanh

- Possible use:
    - OP(int) --- >   float
    - OP(float) --- >   float
- Result: the hyperbolic tangent of the operand (which has to be expressed in decimal degrees).

```
tanh(0)      --: 0.0
tanh(1)      --: 0.7615941559557649
tanh(10)     --: 0.9999999958776927
```

Top of the page

target_of

- Possible use:
    - graph OP any --- >   any
- Result: returns the target of the edge (right-hand operand) contained in the graph given in left-hand operand.

- Special cases:
  - if the lef-hand operand (the graph) is nil, returns nil
- See also: source_of ,

```
let graphEpidemio type: graph <-
generate_barabasi_albert( ["edges_specy"::edge,"vertices_specy"::node,"size"::3,"m"::5] );
graphEpidemio source_of(edge(3))                  --:  node1
let graphFromMap type: graph <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
graphFromMap source_of(link({1,5}::{12,45}))      --: {1.0;5.0}
```

Top of the page

text

- Possible use:
  - OP(string) --- > file
- Result: opens a file that a is a kind of text.
- Comment: The file should have a text extension, cf. file type definition for supported file extensions.
- Special cases:
  - If the specified string does not refer to an existing text file, an exception is risen.
- See also: file , properties , image , shapefile ,

```
let fileT type: file value: text("../includes/Stupid_Cell.Data");
                // fileT represents the text file "../includes/Stupid_Cell.Data"
```

Top of the page

TGauss

Same signification as truncated_gauss operator.

Top of the page

to_gaml

- Possible use:
  - OP(any) --- > string
- Result: represents the gaml way to write an expression in gaml, depending on its type
- See also: to_java ,

```
to_gaml(0)              --: 0
to_gaml(3.78)              --: 3.78
to_gaml(true)              --: true
to_gaml({23, 4.0})          --: {23.0,4.0}
to_gaml(5::34)              --: (5)::(34)
to_gaml(green)              --: rgb (-16711936)
to_gaml('hello')         --: 'hello'
to_gaml([1,5,9,3])          --: [1,5,9,3]
to_gaml(['a'::345, 'b'::13, 'c'::12])          --:  ([('b')::(13),('c')::(12),('a')::(345)] as
map )
to_gaml([[3,5,7,9],[2,4,6,8]])            --: [3,2,5,4,7,6,9,8] as matrix
to_gaml(a_graph)            --: ([((1 as node)::(3 as node))::(5 as edge),((0 as node)::(3 as
node))::(3 as edge),((1 as node)::(2 as node))::(1 as edge),((0 as node)::(2 as node))::(2 as
edge),((0 as node)::(1 as node))::(0 as edge),((2 as node)::(3 as node))::(4 as edge)] as map )
as graph
```

```
to_gaml(node1)                --: 1 as node
```

Top of the page

to_java

- Possible use:
  - OP(any) --- > string
- Result: represents the java way to write an expression in java, depending on its type
- Comment: NOT YET IMPLEMENTED
- See also: to_gaml ,

Top of the page

tokenize

Same signification as split_with operator.

Top of the page

topology

- Possible use:
  - OP(any) --- > topology
- Result: casting of the operand to a topology.
- Special cases:
  - if the operand is a topology, returns the topology itself;
  - if the operand is a spatial graph, returns the graph topology associated;
  - if the operand is a population, returns the topology of the population;
  - if the operand is a shape or a geometry, returns the continuous topology bounded by the geometry;
  - if the operand is a matrix, returns the grid topology associated
  - if the operand is another kind of container, returns the multiple topology associated to the container
  - otherwise, casts the operand to a geometry and build a topology from it.
- See also: geometry ,

```
topology(0)         --: null
topology(a_graph)    --: Multiple topology in POLYGON ((24.712119771887785 7.867357373616512,
24.712119771887785 61.283226839310565, 82.4013676510046  7.867357373616512)) at
location[53.556743711446195;34.57529210646354]
```

Top of the page

touches

- Possible use:
  - shape OP shape --- > bool
- Result: A boolean, equal to true if the left-geometry (or agent/point) touches the right-geometry (or agent/point).
- Comment: returns true when the left-operand only touches the right-operand. When one geometry covers partially (or fully) the other one, it returns false.

- Special cases:
  - if one of the operand is null, returns false.
- See also: <--: , disjoint_from , crosses , overlaps , partially_overlaps , intersects ,

```
polyline([{10,10},{20,20}]) touches geometry({15,15}) --: false
polyline([{10,10},{20,20}]) touches geometry({10,10}) --: true
geometry({15,15}) touches geometry({15,15}) --: false
polyline([{10,10},{20,20}]) touches polyline([{10,10},{5,5}]) --: true
polyline([{10,10},{20,20}]) touches polyline([{5,5},{15,15}]) --: false
polyline([{10,10},{20,20}]) touches polyline([{15,15},{25,25}]) --: false
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{15,15},{15,25},{25,25},{25,15}]) --:
false
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,20},{20,20},{20,30},{10,30}]) --:
true
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,10},{0,10},{0,0},{10,0}]) --:
true
polygon([{10,10},{10,20},{20,20},{20,10}]) touches geometry({15,15}) --: false
polygon([{10,10},{10,20},{20,20},{20,10}]) touches geometry({10,15}) --: true
```

Top of the page

towards

- Possible use:
  - shape OP shape --- > int
- Result: The direction (in degree) between the two geometries (geometries, agents, points) considering the topology of the agent applying the operator.
- See also: distance_between , distance_to , direction_between , path_between , path_to ,

```
ag1 towards ag2 --: the direction between ag1 and ag2 and ag3 considering the topology of the
agent applying the operator
```

Top of the page

transformed_by

- Possible use:
  - shape OP point --- > shape
- Result: A geometry resulting from the application of a rotation and a translation (rigth-operand : point {angle(degree), distance} of the left-hand operand (geometry, agent, point)
- See also: rotated_by , translated_by ,

```
self transformed_by {45, 20} --: returns the geometry resulting from 45# rotation and 10m
translation of the geometry of the agent applying the operator.
```

Top of the page

translated_by

- Possible use:
  - shape OP point --- > shape
- Result: A geometry resulting from the application of a translation by the right-hand operand distance to the left-hand operand (geometry, agent, point)
- See also: rotated_by , transformed_by ,

```
self translated_by 45 --: returns the geometry resulting from a 10m translation to the geometry
of the agent applying the operator.
```

Top of the page

translated_to

Same signification as at_location operator.

Top of the page

triangle

- Possible use:
  - OP(float) --- > shape
- Result: A triangle geometry which side size is given by the operand.
- Comment: the centre of the triangle is by default the location of the current agent in which has been called this operator.
- Special cases:
  - returns nil if the operand is nil.
- See also: around , circle , cone , line , link , norm , point , polygon , polyline , rectangle , square ,

```
triangle(5) --: returns a geometry as a triangle with side_size = 5.
```

Top of the page

triangulate

- Possible use:
  - OP(shape) --- > list of shapes
  - OP(list of shapes) --- > list of shapes
- Result: A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point)A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand list of geometries

```
triangulate(self) --: returns the list of geometries (triangles) corresponding to the Delaunay
triangulation of the geometry of the agent applying the operator.
triangulate(self) --: returns the list of geometries (triangles) corresponding to the Delaunay
triangulation of the geometry of the agent applying the operator.
```

Top of the page

truncated_gauss

- Possible use:
  - OP(point) --- > float
  - OP(list) --- > float
- Result: A random value from a normally distributed random variable in the interval]mean - standardDeviation; mean + standardDeviation[.
- Special cases:
  - when the operand is a point, it is read as {mean, standardDeviation}
  - if the operand is a list, only the two first elements are taken into account as [mean, standardDeviation]

  - when truncated_gauss is called with a list of only one element mean, it will always return 0.0
- See also: gauss ,

```
truncated_gauss ({0, 0.3})  --:  an float between -0.3 and 0.3
truncated_gauss ([0.5, 0.0])  --:  0.5 (always)
```

Top of the page

undirected

- Possible use:
  - OP(graph) --- > graph
- Result: the operand graph becomes an undirected graph.
- Comment: the operator alters the operand graph, it does not create a new one.
- See also: directed ,

Top of the page

union

Same signification as + operator.

- Possible use:
  - OP(list) --- > shape
  - OP(species) --- > shape
  - list OP list --- > list
- Result: The geometry resulting from the union of all geometries of agents of the operand-speciesreturns a new list containing all the elements of both operands without duplicated elements. Elements of this new list are sorted.
- Comment: union is only defined with a list as left operand
- Special cases:
  - if the right-operand is a list of points, geometries or agents, returns the geometry resulting from the union all the geometries
  - if the right operand is nil, union returns a copy of the left operand
- See also: inter , + ,

```
union([geom1, geom2, geom3]) --: a geometry corresponding to union between geom1, geom2 and geom3
union(species1)  --: returns the geometry resulting from the union of all of the geometries of
agents of species species1.
[1,2,3,4,5,6] union [2,4,9]         --:     [1,2,3,4,5,6,9]
[1,2,3,4,5,6] union [0,8]             --:     [0,1,2,3,4,5,6,8]
[1,3,2,4,5,6,8,5,6] union [0,8]     --:     [0,1,2,3,4,5,6,8]
```

Top of the page

unknown

- Possible use:
  - OP(any) --- > any
- Result: returns the operand itself

Top of the page

user_input

- Possible use:
  - OP(map) --- > map
- Result: asks the user for some values (not defined as parameters)
- Comment: This operator takes a map [string::value] as argument, displays a dialog asking the user for these values, and returns the same map with the modified values (if any). The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section to force the user to input new values instead of relying on the initial values of parameters :

```
init {
    let values <- user_input(["Number" :: 100, "Location" :: {10, 10}]);
    create node number : int(values at "Number") with: [location:: (point(values at
"Location"))];
}
```

Top of the page

variance

- Possible use:
  - OP(list) --- > float
- Result: the variance of the elements of the operand. See Variance < http://en.wikipedia.org/wiki/Variance> for more details.
- Comment: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- See also: mean , median ,

```
variance ([4.5, 3.5, 5.5, 7.0]) --: 1.671875
```

Top of the page

weight_of

- Possible use:
  - graph OP any --- > float
- Result: returns the weight of the given edge (right-hand operand) contained in the graph given in right-hand operand.
- Comment: In a localized graph, an edge has a weight by default (the distance between both vertices).
- Special cases:
  - if the left-operand (the graph) is nil, returns nil
  - if the right-hand operand is not an edge of the given graph, weight_of checks whether it is a node of the graph and tries to return its weight
  - if the right-hand operand is neither a node, nor an edge, returns 1.

```
let graphFromMap type: graph <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
graphFromMap source_of(link({1,5}::{12,45}))  --: 41.48493702538308
```

Top of the page

where

- Possible use:
  - map OP any expression --- > map
  - container OP any expression --- > list
- Result: a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is a list nil, where returns a new empty list
- See also: first_with , last_with , where ,

```
[1,2,3,4,5,6,7,8] where (each > 3)                          --:     [4, 5, 6, 7, 8]
g2 where (length(g2 out_edges_of each) = 0 )                  --:     [node9, node7, node10,
node8, node11]
(list(node) where (round(node(each).location.x) > 32)        --:      [node2, node3]
[1::2, 3::4, 5::6] where (each.value > 4)                          --:
```

Top of the page

with_max_of

- Possible use:
  - container OP any expression --- > any
- Result: one of elements of the left-hand operand that maximizes the value of the right-hand operand
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand
- See also: where , with_min_of ,

```
[1,2,3,4,5,6,7,8] with_max_of (each )                          --:     8
g2 with_max_of (length(g2 out_edges_of each)  )              --:     node4
(list(node) with_max_of (round(node(each).location.x))        --:      node3
[1::2, 3::4, 5::6] with_max_of (each)                          --:     6
```

Top of the page

with_min_of

- Possible use:
  - container OP any expression --- > any
- Result: one of elements of the left-hand operand that minimizes the value of the right-hand operand
- Comment: in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.
- Special cases:
  - if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand
- See also: where , with_max_of ,

```
[1,2,3,4,5,6,7,8] with_min_of (each )                          --:     1
g2 with_min_of (length(g2 out_edges_of each)  )              --:     node11
(list(node) with_min_of (round(node(each).location.x))        --:      node0
```

```
[1::2, 3::4, 5::6] with_min_of (each)                           --:      2
```

Top of the page

with_optimizer_type

- Possible use:
  - graph OP string --- > graph
- Result: changes the shortest path computation method of the griven graph
- Comment: the right-hand operand can be "Djikstra", "Bellmann", "Astar" to use the associated algorithm. Note that these methods are dynamic: the path is computed when needed. In contrarily, if the operand is another string, a static method will be used, i.e. all the shortest are previously computed.
- See also: set_verbose ,

```
set graphEpidemio <- graphEpidemio with_optimizer_type "static";
```

Top of the page

with_precision

- Possible use:
  - float OP int --- > float
- Result: round off the value of left-hand operand to the precision given by the value of right-hand operand
- See also: round ,

```
12345.78943 with_precision 2      --:      12345.79
123 with_precision 2      --:      123.00
```

Top of the page

with_weights

- Possible use:
  - graph OP list --- > graph
  - graph OP map --- > graph
- Result: returns the graph (left-hand operand) with weight given in the map (right-hand operand).
- Comment: this operand re-initializes the path finder
- Special cases:
  - if the right-hand operand is a list, affects the n elements of the list to the n first edges. Note that the ordering of edges may change overtime, which can create some problems...
  - if the left-hand operand is a map, the map should contains pairs such as: vertex/ edge::double

```
graph_from_edges (list(ant) as_map each::one_of (list(ant))) with_weights (list(ant) as_map
each::each.food)
```

Top of the page

without_holes

- Possible use:
  - OP(shape) --- > shape

- Result: A geometry corresponding to the operand geometry (geometry, agent, point) without its holes

```
solid(self) --: returns the geometry corresponding to the geometry of the agent applying the operator without its holes.
```

Top of the page

write

- Possible use:
  - OP(file) --- > any
- Result: marks the file so that read and write operations are allowed.
- Comment: A file is created by default in read-only mode.
- See also: file , read ,

```
write(shapefile("../images/point_eau.shp"))   --: returns a file in read-write mode representing "../images/point_eau.shp"
```

Top of the page

Keywords

constants

nil

Represents the null value (or undefined value). It is the default value of variables of type agent, point or species when they are not initialized.

true, false

Represent the two possible values of boolean variables or expressions. (see bool).

global built-in variables

Global built-in variables can be accessed (and sometimes modified) by the world agent and every other agents in the model.

time

- float, read-only, represents the current simulated time in seconds (the default unit). Begins at zero.

```
global {
...
    int nb_minutes function: { int(time / 60)};
...
}
```

step

- float, represents the time step between two executions of the set of agents, in seconds. Its default value is 1. Each turn, the value of time is incremented by the value of step. The definition of step must be coherent with that of the agents' variables like speed.

```
global {
...
    float step <- 10.0;
...
}
```

seed

- float, represents the seed used in the computation of random numbers. Keeping the same seed between two runs of the same model ensures that the sequence of events will remain the same,

which can be useful when debugging a model. Declaring it as a parameter allows the user or an external process (batch, for instance) to modify it.

```
global {
...
    float seed <- 354 parameter: true;
...
}
```

agents

- list, read-only, returns a list of all the agents of the model that are considered as "active" (i.e. all the agents with behaviors, excluding the places). Note that obtaining this list can be quite time consuming, as the world has to go through all the species and get their agents before assembling the result. For instance, instead of writing something like:

```
ask agents of_species my_species {
...
}
```

one would prefer to write (which is much faster):

```
ask list (my_species) {
...
}
```

pseudo-variables

Pseudo-variables are special variables whose value cannot be changed by agents (but can change depending on the context of execution).

self

self is a pseudo-variable (can be read, but not written) that always holds a reference to the executing agent.

- Example (sets the friend field of another random agent to self and reversely) :

```
let potential_friend <- one_of (list (species(self)) - self);
if potential_friend != nil {
    set potential_friend.friend <- self;
    set friend <- potential_friend;
}
```

myself

myself is the same as self, except that it needs to be used instead of self in the definition of remotely-executed code (ask and |create commands). myself represents the sender agent when the code is executed by the target agent.

- Example (asks the first agent of my species to set its color to my color) :

```
ask first (list (species (self))){
    set color <- myself.color;
}
```

- Example (create 10 new agents of my species, share my energy between them, turn them towards me, and make them move 4 times to get closer to me) :

```
create species (self) number: 10 {
  set energy <- myself.energy / 10.0;
  loop times: 4 {
      set heading <- towards (myself);
      do move;
  }
}
```

each

each is a pseudo-variable only used in filter expressions following operators such as where or first_with. It represents, in turn, each of the elements of the target datatype (a list, string, point or matrix, usually).

units

Units can be used to qualify the values of numeric variables. By default, unqualified values are considered as:

- meters for distances, lengths...
- seconds for durations
- cubic meters for volumes
- kilograms for masses

So, an expression like

```
let foo type: float <- 1;
```

will be considered as 1 meter if foo is a distance, or 1 second if it is a duration, or 1 meter/second if it is a speed. If one wants to specify the unit, it can be done very simply by adding the unit name after the numeric value, like:

```
let foo type: float <- 1 centimeter;
```

In that case, the numeric value of foo will be automatically translated to 0.01 (meter). It is recommended to always use float as the type of the variables that might be qualified by units (otherwise, for example in the previous case, they might be truncated to 0). Several units names are allowed as qualifiers of numeric variables. As they can be used in expressions directly, they are considered as reserved keywords (and therefore cannot be used for naming variables and species). Their complete list is:

length

meter (default), meters, m, centimeters, centimeter, cm, millimeter, millimeters, mm, decimeter, decimeter, dm, kilometer, kilometers, km, mile, miles, yard, yards, inch, inches, foot, feet, ft.

time

second (default), seconds, sec, s, minute, minutes, mn, hour, hours, h, day, days, d, month, months, year, years, y, millisecond, milliseconds, msec.

mass

kilogram (default),kilogram, kilo, kg, ton, tons, t, gram, grams, g, ounce, ounces, oz, pound, pounds, lb, lbm.

surface

square_meter (default), m2, square_meters, square_mile, square_miles, sqmi, square_foot, square_feet, sqft.

volume

m3 (default), liter, liters, l, centiliter, centiliters, cl, deciliter, deciliters, dl, hectoliter, hectoliters, hl. These represent the basic metric and US units. Composed and derived units (like velocity, acceleration, special volumes or surfaces) can be obtained by combining these units using the **and / operators. For instance:**

```
float one_kmh <- 1 km / h const: true;
float one_microsecond <-1 sec / 1000;
float one_cubic_inch <- 1 sqin * 1 inch;
... etc ...
```

—

Built-in Agents

Introduction

It is possible to use in the models a set of built-in agents. These agents allow to directly use some advance features like clustering, multi-criteria analysis, etc. The creation of these agents are similar as for other kinds of agents:

```
create species: my_built_in_agent return: the_agent;
```

The list of available built-in agents in GAMA is:

- cluster_builder: allows to use clustering techniques on a set of agents.
- multicriteria_analyzer: allows to use multi-criteria analysis methods.

So, for instance, to be able to use clustering techniques in the model:

```
create cluster_builder return: clusterer;
```

cluster_builder

The **cluster_builder** agent allows to divide a set of agents into different clusters according to the values of some of their attributes. This agents is built from the weka library : most of the clustering algorithms are directlty based on their weka implementation.

actions

simple_clustering_by_distance

Returns groups of agents using hierarchical clustering. The distance between agents are directly computed from the agent locations (euclidean distance). The distance between two groups of agents corresponds to the min of distances between the agents of each group.

- → agents: list, the list of Entities to be divided into groups.
- → dist_min: float, minimal distance between two groups. By default, num_clusters = -1.
- → distance_geom: bool, optional, if true, uses the distance between the agent geometry; otherwise uses the distance between the agent centroid (more optimize). By default, distance_geom = false.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let groups type: list <- self simple_clustering_by_distance [agents::ags, dist_min::10.0];
```

clustering_cobweb

Returns groups of agents using the Cobweb and Classit clustering algorithms (Weka implementation). For more information see: D. Fisher (1987). Knowledge acquisition via incremental conceptual clustering.

Machine Learning. 2(2):139-172; J. H. Gennari, P. Langley, D. Fisher (1990). Models of incremental concept formation. Artificial Intelligence. 40:11-61.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → acuity: float, optional, minimum standard deviation for numeric attributes. By default, acuity = 1.0.
- → cutt_off: float, optional, set the category utility threshold by which to prune nodes. By default, cutoff = 0.0028209479177387815.
- → seed, int, optional, The random number seed to be used. By default, seed = 42.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_cobweb [agents::ags, attributes::attribs];
```

clustering_DBScan

Returns groups of agents using the DBScan algorithm (Weka implementation). For more information see: Martin Ester, Hans-Peter Kriegel, Joerg Sander, Xiaowei Xu: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Second International Conference on Knowledge Discovery and Data Mining, 226-231, 1996.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → distance_f: string, optional, The distance function to use : 2 possible distance functions: (by default) euclidean; and 'manhattan'.
- → epsilon: float, optional, radius of the epsilon-range-queries. By default, epsilon = 0.9.
- → min_points: int, optional, minimun number of [DataObjects] required in an epsilon-range-query. By default, min_points = 6.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_DBScan [agents::ags, attributes::attribs];
```

clustering_em

Returns groups of agents using the EM (expectation maximisation) algorithm (Weka implementation).

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 100.
- → num_clusters: int, optional, set number of clusters. if num_clusters equals -1, EM decides how many clusters to create by cross validation. By default, num_clusters = -1.
- → min_std_dev: float, optional, set minimum allowable standard deviation. By default, min_std_dev = 1.0E-6.
- → seed, int, optional, The random number seed to be used. By default, seed = 100.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_em [agents::ags, attributes::attribs];
```

clustering_farthestFirst

Returns groups of agents using the farthestFirst algorithm (Weka implementation). For more information see: Hochbaum, Shmoys (1985). A best possible heuristic for the k-center problem. Mathematics of Operations Research. 10(2):180-184.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → num_clusters: int, optional, set number of clusters. By default, num_clusters = 2.
- → seed, int, optional, The random number seed to be used. By default, seed = 1.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_farthestFirst [agents::ags, attributes::attribs];
```

clustering_simple_kmeans

Returns groups of agents using the K-Means algorithm (Weka implementation).

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → distance_f: string, optional, The distance function to use : 4 possible distance functions: (by default) euclidean; otherwise, 'chebyshev', 'manhattan' and 'levenshtein'.
- → dont_replace_missing_values: bool, optional, Replace missing values globally with mean/mode. By default, dont_replace_missing_values = false.
- → max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 500.
- → num_clusters: int, optional, set number of clusters. By default, num_clusters = 2.
- → preserve_instances_order: bool, optional, Preserve order of instances. By default, preserve_instances_order = false.
- → seed, int, optional, The random number seed to be used. By default, seed = 10.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_simple_kmeans [agents::ags, attributes::attribs];
```

clustering_xmeans

Returns groups of agents using the X-Means algorithm (Weka implementation). "X-Means is K-Means extended by an Improve-Structure part. In this part of the algorithm the centers are attempted to be split in its region. The decision between the children of each center and itself is done comparing the BIC-values of the two structures. For more information see: Dan Pelleg, Andrew W. Moore: X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: Seventeenth International Conference on Machine Learning, 727-734, 2000." (Weka documentation).

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → bin_ < - float, optional, Set the value that represents true in the new attributes. By default, bin_value = 1.0.
- → cut_off_factor: float, optional, the cut-off factor to use. By default, cut_off_factor = 0.5.
- → distance_f: string, optional, The distance function to use : 4 possible distance functions: (by default) euclidean; otherwise, 'chebyshev', 'manhattan' and 'levenshtein'
- → max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 1.
- → max_kmeans: int, optional, the maximum number of iterations to perform in KMeans. By default, max_iterations = 1000.
- → max_kmeans_for_children: int, optional, the maximum number of iterations KMeans that is performed on the child centers. By default, max_kmeans_for_children = 1000.
- → max_num_clusters: int, optional, set maximum number of clusters. By default, max_num_clusters = 4.
- → min_num_clusters: int, optional, set minimum number of clusters. By default, min_num_clusters = 2.
- → seed, int, optional, The random number seed to be used. By default, seed = 10.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent <- [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string <- ['area','food_quantity','proximity_to_roads'];
let groups type: list <- self clustering_xmeans [agents::ags, attributes::attribs];
```

multicriteria_analyzer

The **multicriteria_analyzer** agent allows to make a decision from a set of candidate solutions according to a set of criteria.

actions

weighted_means_DM

Returns the index of the candidate with the highest utility. The utility of each candidate is computed by a weighted means.

- → criteria: list, the list of criteria. A criterion is a map that contains two elements: a name, and a weight.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list <- [[name::'proximity', weight::1], [name::'quality', weight::3],
[name::'usefulness', weight::2]];
let cands type: list <- [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],[0.9, 0.2, 0.4],[0.6,
0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int <- self weighted_means_DM [criteria::crits, candidates::cands];
```

promethee_DM

Returns the index of the *best* candidate according to the Promethee II method. This method is based on a comparison per pair of possible candidates along each criterion: all candidates are compared to each other by pair and ranked. More information about this method can be found in Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research(2009)

- → criteria: list, the list of criteria. A criterion is a map that contains fours elements: a name, a weight, a preference value (p) and an indifference value (q). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list <- [[name::'proximity', weight::1, p::0.9, q::0.1], [name::'quality',
weight::3, p::1.0, q::0.0],[name::'usefulness', weight::2, p::0.8, q::0.2]];
let cands type: list <- [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],[0.9, 0.2, 0.4],[0.6,
0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int <- self promethee_DM [criteria::crits, candidates::cands];
```

electre_DM

Returns the index of the *best* candidate according to a method based on the ELECTRE methods. The principle of the ELECTRE methods is to compare the possible candidates by pair. These methods analyses the possible outranking relation existing between two candidates. An candidate outranks another if this one is at least as good as the other one. The ELECTRE methods are based on two concepts: the concordance and the discordance. The concordance characterises the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterises the fact that, for an outranking relation to be validated, none of the criteria in the minority should oppose too strongly this assertion. These two conditions must be true for validating the outranking assertion. More information about the ELECTRE methods can be found in Figueira, J., Mousseau, V., Roy, B.: ELECTRE Methods. In: Figueira, J., Greco, S., and Ehrgott, M., (Eds.), Multiple Criteria Decision Analysis: State of the Art Surveys, Springer, New York, 133--162 (2005)

- → criteria: list, the list of criteria. A criterion is a map that contains fives elements: a name, a weight, a preference value (p), an indifference value (q) and a veto value (v). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant. The veto value represents the threshold from which the difference between two criterion values disqualifies the candidate that obtained the smaller value.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list <- [[name::'proximity', weight::1, p::0.9, q::0.1, v::0.95],
[name::'quality', weight::3, p::0.8, q::0.0, v:1.0],
    [name::'usefulness', weight::2, p::0.8, q::0.2, v::0.9]];
let cands type: list <- [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],[0.9, 0.2, 0.4],[0.6,
0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int <- self electre_DM [criteria::crits, candidates::cands];
```

evidence_theory_DM

Returns the index of the *best* candidate according to a method based on the Evidence theory. This theory, which was proposed by Shafer (Shafer G (1976) A mathematical theory of evidence, Princeton University Press), is based on the work of Dempster (Dempster A (1967) Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, vol. 38, pp. 325--339) on lower and upper probability distributions.

- → criteria: list, the list of criteria. A criterion is a map that contains seven elements: a name, a first threshold s1, a second threshold s2, a value for the assertion "this candidate is the best" at threshold s1 (v1p), a value for the assertion "this candidate is the best" at threshold s2 (v2p), a value for the assertion "this candidate is not the best" at threshold s1 (v1c), a value for the assertion "this candidate is not the best" at threshold s2 (v2c). v1p, v2p, v1c and v2c have to been defined in order that: v1p + v1c < = 1.0; v2p + v2c < = 1.0.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list <- [[name::'proximity', s1::0.1, s2::0.8, v1p::0, v2p::0.3, v1c::0.5,
v2c::0], [name::'quality',s1::0.0, s2::0.8, v1p::0.05, v2p::0.5, v1c::0.6, v2c::0],
[name::'usefulness', s1::0.0, s2::0.8, v1p::0, v2p::0.2, v1c::0.9, v2c::0]];
let cands type: list <- [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],[0.9, 0.2, 0.4],[0.6,
0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int <- self evidence_theory_DM [criteria::crits, candidates::cands];
```

Built-in Actions

Built-in actions

Two built-in actions are provided to each agent. These actions can be directly used by all species of agents (amongst them the world agent).

debug

makes the agent output an arbitrary message in the console. The message is automatically prefixed with the cycle of the simulation and followed by a carriage return and postfixed with information concerning the agent that called this action.

- → **message** : string, mandatory, the message to display.

```
do debug message: 'This is a message from ' + self;
```

tell

makes the agent output a dialog (if the simulation contains a user interface). The simulation goes on, but its interface is not accessible until the dialog is closed (use with caution, as it may prevent the user from accessing the interface).

- → **message** : string, mandatory, the message to display.

```
do tell message: 'This is a message dialog raised by ' + self;
```

Global built-in actions

In addition to the built-in actions, there are two actions provided to the world agent.

halt

- stops the simulation.

```
global {
    ...
    reflex halting when: empty (agents) {
            do halt;
    }
}
```

pause

- pauses the simulation, which can then be continued by the user.

```
global {
```

```
    ...
    reflex toto when: time = 100 {
            do pause;
    }
}
```

—

Additional features

Gama 3D

Introduction

Gama default display is based on the Java2D API that can show some limits when dealing with a huge amount of data or when once wants to display the model in a more realistic way. Since 1.5 version , OpenGL 3D display is integrated in GAMA.

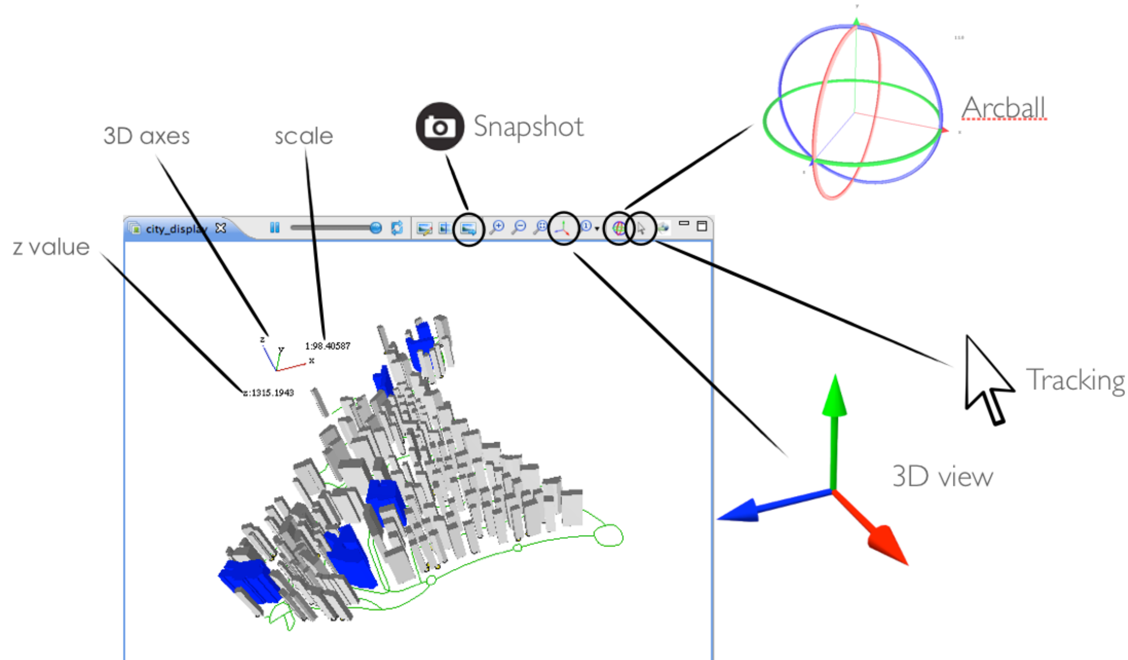
How to use OpenGL display in Gama

To use the openGL display just define the atrribute type of the display with " type:opengl " in the output of your model:

```
experiment my_3D_experiment type: gui {
  output {
    display myDisplay type:opengl {
      species mySpecies;
    }
}
```

This will create an opengl view of your model. With this view you can now navigate throught the model in 2D as with the default display but also in 3D.

Gama 3D Simulation perspective



Mouse interaction

- click + mouve = make the model moving on the plan
- ctrl (or cmd for mac) + mouve: make the camera orbitting around the model. (you can use the arcball button to activate this function without using ctrl (or cmd for mac).
- scroll out/in: zoom out/in

Toogle button

- Arcball: Enables to orbit around the model. When this button is clicked simply drag the mouse to orbit around the model. And press shift to pan the model. If this button is not activated press the ctrl (or cmd for mac) + drag the mouse to use arcball.
- Tracking: This function is only available in 2D mode. When activing this function the display switch automatically to 2D mode. You can then select one agent. This agent will be displayed in red during the rest of the simulation.
- 3D view: Positions the model in 3D view when activated and positions the model in 2D view when desactivated.
- Snapshot: Creates a picture of the current display when clicking on it. The picture will be stored in a folder snapshots in the same folder where your model is. You can take different picture of your model the name of the file will be linked with the current iteration (this function also exists with the default display in Java 2D).

Information

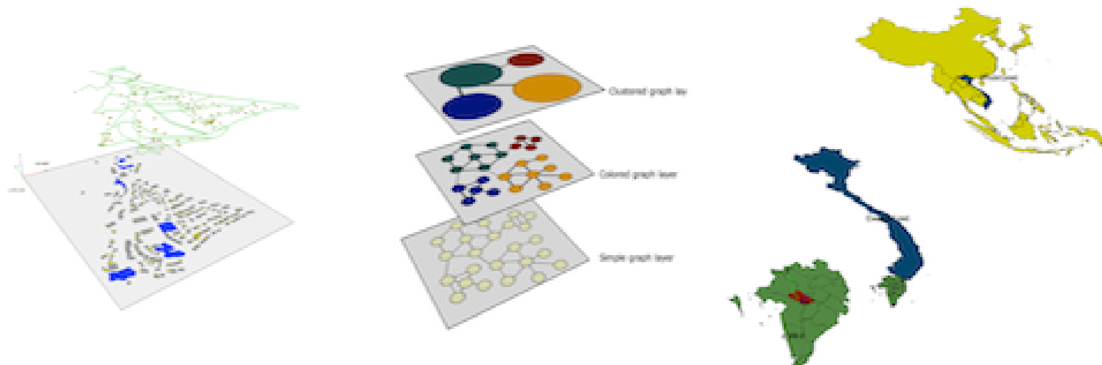
- 3D axes: Gives an indication of the position of the camera so that you always know where you are positioned in 3D.
- z value: Displays the z value of the camera.
- scale: Indicates the scale of the model. Shows what is the value of 1 unit of each axes.

Why using 3D in your model?

The third dimension not only enables to move and manipulate the world (the model) in an interactive and intuitive way but also provides new way to visualize models. Many examples can be found in the models/3D repository. Here is an overview of the feature provided by Gama 3D.

Multi layer

During a simulation, a user may want to observe the same agent evolving in different environment, such a thing can easily be provided by using the multi layer display. On each different layer the user describes the agent he wants to see. Species can be placed on different z value for each layer using the opengl display. z:0 means the layer will be placed on the ground and z=1 means it will be placed at an height equal to the maximun size of the environment.



```
display myDisplay type:opengl {
species building aspect: base z:0;
species road aspect: base z:0.5;
species people aspect: base z:1;
}
```

See example in:

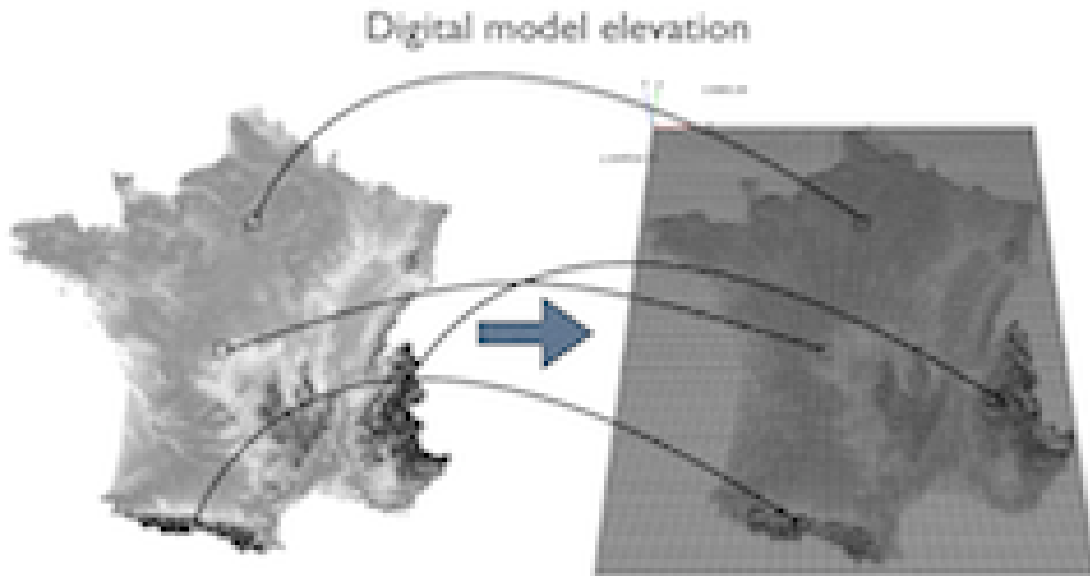
```
models/3D/road_traffic/models/model_building_elevation_multi_layer.gaml
```

or

```
models/3D/multi_layer_rendering/multi_layer_picture.gaml
models/3D/multi_layer_rendering/vietnam_multi_layer.gaml
models/3D/multi_layer_rendering/vietnam_multi_layer_with_3D_Agent.gaml
```

Digital Model Elevation

A digital elevation model is a digital model or 3D representation of a terrain’s surface. From a 2D picture the value of the pixel is converted to an altitude value and is then rendered in 3D.

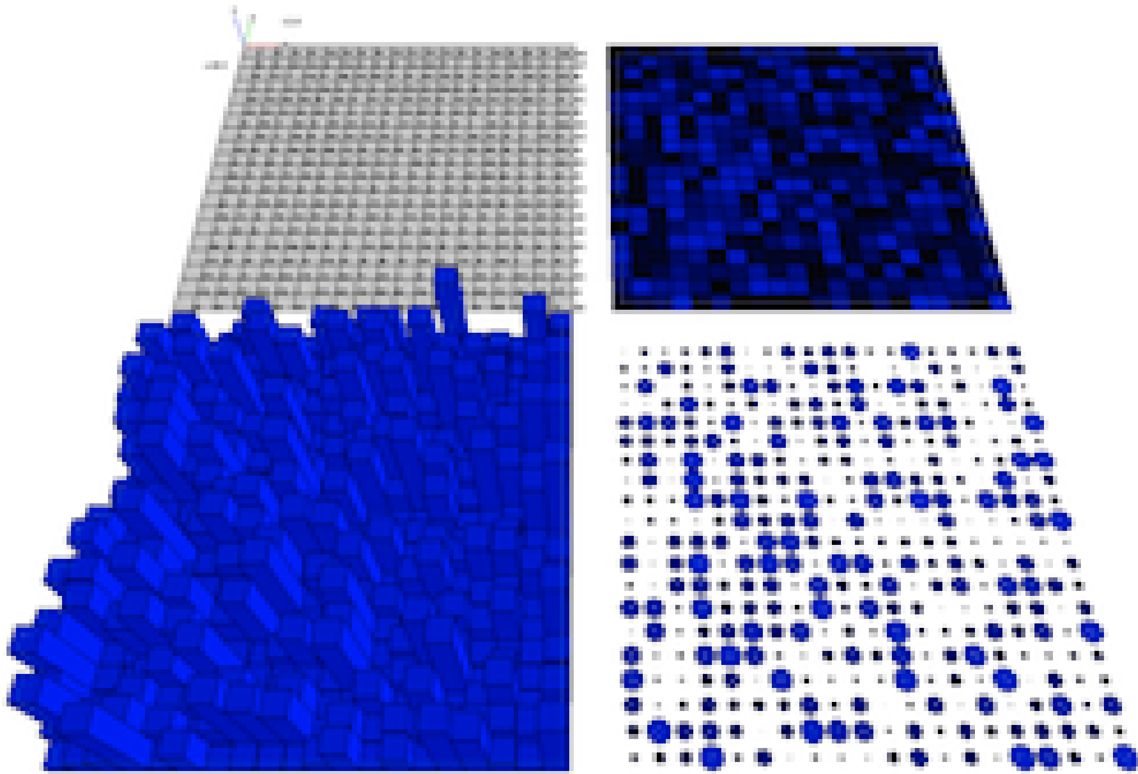


See example in:

```
models/3D/DEM/models/model_mekong_mnt.gaml
models/3D/DEM/models/model_france_mnt.gaml
```

Data into form

There is many way to visualize and distinguish data elements: size, value, texture, color, orientation and shape. For each agent the value of one of its parameters can represented in different ways such as a text, a color (e.g light for low value dark for high value), a cercle where the radius of the circle is proportional to the value and a 3D view where the height of the agent is proportional to the value. In many cases the shape of the agent can be used to represent one or several of its attributes.



See example in:

```
models/3D/PrimitiveShape/models/grid_data_to_form.gaml
```

User Control

Introduction

GAMA provides some tools to give more flexibility to the user in controlling the agents, creating agents, killing agents, running specific actions, etc.

user_command

Anywhere in the global block, in a species or in an (GUI) experiment, user_command statements can be implemented. They can either call directly an existing action (with or without arguments) or be followed by a block that describes what to do when this command is run. Their syntax can be (depending of the modeler needs) either:

```
user_command cmd_name action: action_without_arg_name;
```

or

```
user_command cmd_name action: action_name with: [arg1::val1, arg2::val2, ...];
```

or

```
user_command cmd_name {
  [statements]
}
```

For instance :

```
user_command kill_myself action: die;
```

or

```
user_command kill_myself action: some_action with: [arg1::val1, arg2::val2, ...];
```

or

```
user_command kill_myself {
    do action: die;
}
```

These commands (which belong to the "top-level" statements like actions, reflexes, etc.) are not executed when an agent runs. Instead, they are collected and used as follows:

- When defined in a GUI experiment, they appear as buttons above the parameters of the simulation;
- When defined in the global block or in any species,
  - when the agent is inspected, they appear as buttons above the agents' attributes
  - when the agent is selected by a right-click in a display, these command appear under the usual "Inspect", "Focus" and "Highlight" commands in the pop-up menu.

Remark: The execution of a command obeys the following rules:

- when the command is called from right-click pop-menu, it is executed immediately,
- when the command is called from panels, its execution is postponed until the end of the current step and then executed at that time.

user_location

In the special case when the user_command is called from the pop-up menu (from a right-click on an agent in a display), the location chosen by the user (translated into the model coordinates) is passed to the execution scope under the name **user_location** .

Example :

```
global {
  user_command "Create agents here" {
      create my_species number: 10 with: [location::user_location];
  }
}
```

This will allow the user to click on a display, choose the world (always present now), and select the menu item "Create agents here". Note that if the world is inspected (this user_command appears thus as a button) and the user chooses to push the button, the agent will be created at a random location.

`user_input` operator

As it is also, sometimes, necessary to ask the user for some values (not defined as parameters), the user_input unary operator has been introduced. This operator takes a map [string::value] as argument, displays a dialog asking the user for these values, and returns the same map with the modified values (if any). The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section like the following one to force the user to input new values instead of relying on the initial values of parameters :

```
global {
  init {
      let values type: map <- user_input(["Number" :: 100, "Location" :: {10, 10}]);
      create my_species number : int(values at "Number") with: [location:: (point(values at
"Location"))];
  }
}
```

user control architecture

user_only, user_first, user_last

A new type of control architecture has been introduced to allow users to take control over an agent during the course of the simulation. It can be invoked using three different keywords: user_only , user_first , user_last .

```
species user control: user_only {
  ...
}
```

If the control chosen is user_first , it means that the user controled panel is opened first, and then the agent has a chance to run its "own" behaviors (reflexes, essentially, or "init" in the case of a "user_init" panel). If the control chosen is user_last , it is the contrary.

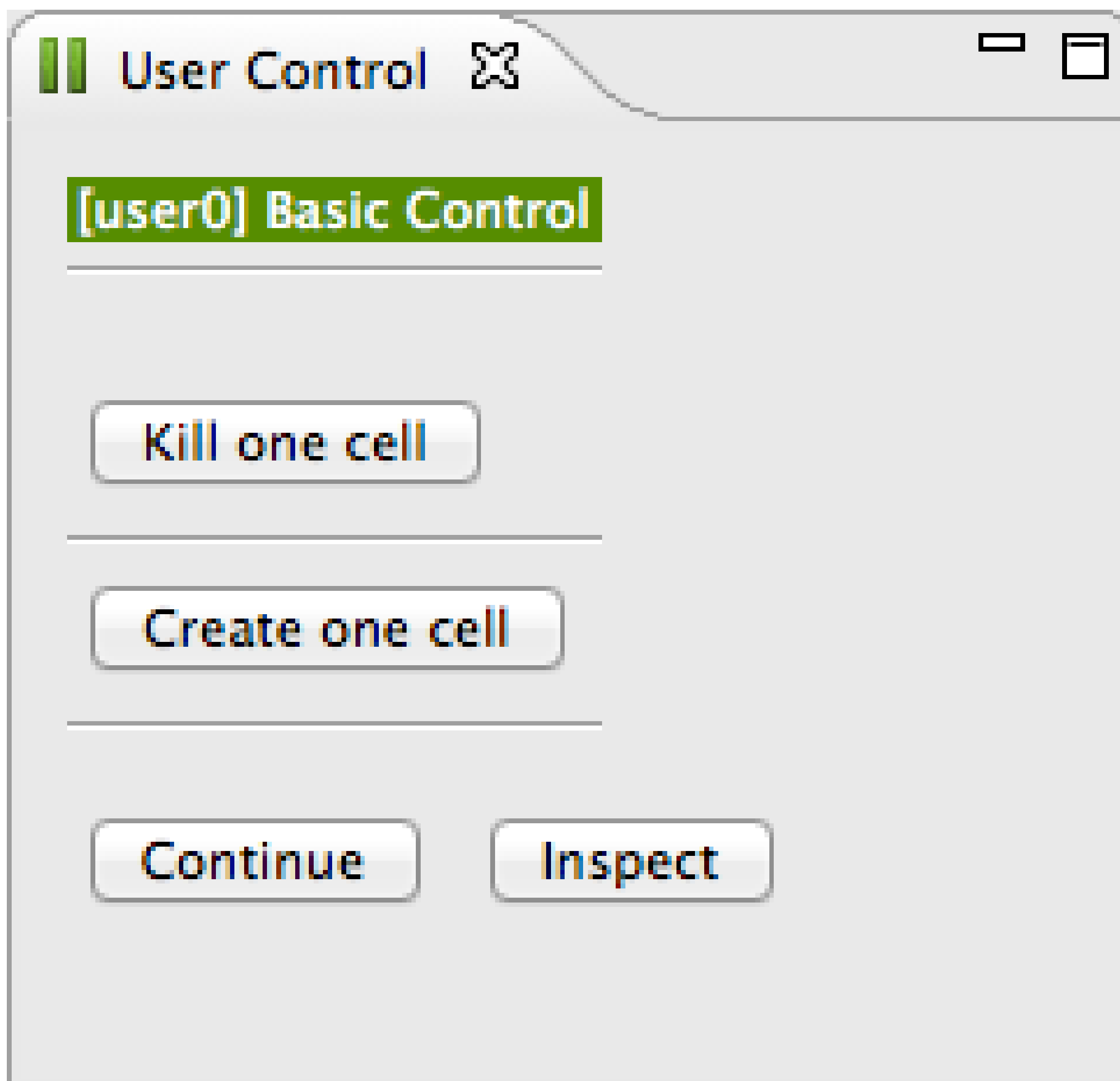
user_panel

This control architecture is a specialization of the Finite State Machine Architecture where the "behaviors" of agents can be defined by using new constructs called user_panel (and one user_init), mixed with "states" or "reflexes". This user_panel translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each user_panel , like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of user_command s which describe the different action buttons present in the panel. Example, in the circle.gaml model :

```
species user control:user_only {

  user_panel "Basic Control"{
      user_command "Kill one cell" {
    ask (one_of(cells)){
        do die;
    }
      }
      user_command "Create one cell" {
    create cells number: 1;
      }
  }
}
```

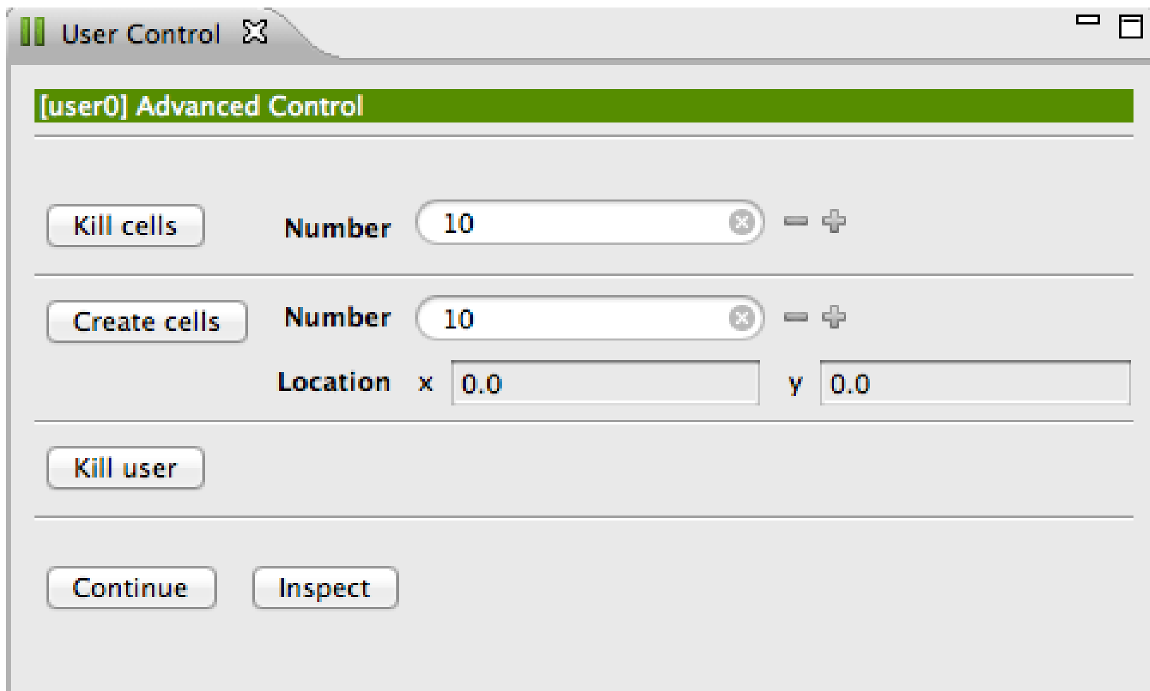
this will create a simple panel like the one on the following image.



user_commands can also accept inputs, in order to create more interesting commands for the user. This uses the user_input statement (and not operator), which is basically the same as a temporary variable declaration whose value is asked to the user. Example:

```
user_command "Kill cells" {
  user_input "Number" returns: number type: int <- 10;
  ask (number among list(cells)){
    do die;
  }
}
```

It translates into a button + an input (as depicted on the following image).



Several inputs are of course allowed (see « Create Cells » on the same panel).

```
  user_command "Create cells" {
      user_input "Number" returns: number type: int <- 10;
      user_input "Location" returns: loc type: point <- {0,0};
      create cells number: number with: [location::loc];
  }
```

As user_panel# is a specialization of state , the modeler has the possibility to describe several panels and choose the one to open depending on some condition, using the same syntax than for finite state machines :

- either adding transitions to the user_panels,
- or setting the state attribute to a new value, from inside or from another agent.

This ensures a great flexibility for the design of the user interface proposed to the user, as it can be adapted to the different stages of the simulation, etc.. Follows a simple example, where, every 10 steps, and depending on the value of an attribute called « advanced », either the basic or the advanced panel is proposed.

```
species user control:user_only{
  bool advanced <- false;
  user_panel default initial: true {
      transition to: "Basic Control" when: every (10) and !advanced;
      transition to: "Advanced Control" when: every(10) and advanced;
  }

  user_panel "Advanced Control" {
      user_command "Kill cells" {
    user_input "Number" returns: number type: int <- 10;
    ask (number among list(cells)){
      do die;
    }
      }
```

```
      user_command "Create cells" {
    user_input "Number" returns: number type: int <- 10;
    user_input "Location" returns: loc type: point <- {0,0};
    create cells number: number with: [location::loc];
      }
      user_command "Kill user" {
    do die;
      }
      transition to: default when: true;
  }

  user_panel "Basic Control"{
      user_command "Kill one cell" {
    ask (one_of(cells)){
      do die;
    }
      }
      user_command "Create one cell" {
    create cells number: 1;
      }
      transition to: default when: true;
  }
}
```

The panel marked with the « initial: true » facet will be the one run first when the agent is supposed to run. If none is marked, the first panel (in their definition order) is chosen. A special panel called user_init will be invoked only once when initializing the agent if it is defined. If no panel is described or if all panels are empty (ie. no user_commands), the control view is never invoked. If the control is said to be "user_only", the agent will then not run any of its behaviors.

user_controlled

Finally, each agent provided with this architecture inherits a boolean attribute called user_controlled .
If this attribute becomes false, no panels will be displayed and the agent will run "normally" unless its species is defined with a user_only control.

Database access

Description

- plug-in: irit.maelia.gaml.additions
- author: TRUONG Minh Thai, Frederic AMBLARD, Benoit GAUDOU, Christophe SIBERTIN-BLANC

Introduction

SQL features of GAMA provide a set of actions on [DataBase] Management Systems (DBMS) for agents in GAMA. With these features, an agent can execute SQL queries (create, Insert, select, update, drop, delete) to various kinds of DBMS. These features are implemented in the irit.maelia.gaml.additions plug-in with two components:

- the skill SQLSKILL
- the built-in species AgentDB .

SQLSKILL and AgentDB provide almost same features (a same set of actions on DBMS) but with a little difference:

- an agent of species AgentDB will maintain a unique connection to the database during the whole simulation. The connection is thus initialized when the agent is created.
- in contrarily, an agent of a species with the SQLSKILL skill will open a connection each time he wants to execute a query. This means that each action will composed of three running steps:
  - Make a database connection.
  - Execute SQL statement.
  - Close database connection.

An agent with the SQLSKILL spends lot of time to create/close the connection each time it needs to send a query, it saves database connection (DBMS often limit the number of simultaneous connections). In contrast, an AgentDB agent only needs to establish one database connection and it can be used for any actions. Because it does not need to create and close database connection for each action, therefore actions of AgentDB agents are executed faster than actions of SQLSKILL ones but we must pay a connection for each agent. With SQL features, we can create species, define environment or store simulation results into DBMS. It helps us to have more flexibility in management of simulation models and analysis of simulation results.

SQLSkill

Define a species that uses the SQLSKILL skill

Example of declaration:

```
entities {
```

```
  species toto skills: [SQLSKILL]
  {
    //insert your descriptions here
  }
  ...
}
```

Agents with such a skill can use additional actions (defined in the skill).

Map of connection parameters

In the actions defined in the SQLSkill, a parameter containing the connection parameters is requiered. It is a map with following key::value pairs:

Key	**Description**
dbtype	DBMS type value. Its value is a string. we must use "sqlserver" when we want to connect to a sqlerver. That is the same for "sqlite" or "mysql" (ignore case sensitive)
host	Host name or IP address of data server. it is absent when we work with SQlite.
port	Port of connection. It is absent when we work with SQLite.
database	Name of database. It is file name include path when we work with SQLite.
user	Username. It is absent when we work with SQLite.
passwd	Password. It is absent when we work with SQLite.

Action: testConnection [params:: connection_parameter]

The action tests the connection to a given database.

- **Return value** : boolean
  - true: the agent can connect to the DBMS (to the given Database with given name and password)
  - false: the agent cannot connect
- **Arguments**
  - params: (type = map) map of connection parameters

Example: Definitions of connection parameter

```
// SQLSERVER connection parameter
map SQLSERVER <- [
      'host'::'localhost',
      'dbtype'::'sqlserver',
      'database'::'BPH',
      'port'::'1433',
      'user'::'sa',
      'passwd'::'abc'];
```

```
// MySQL connection parameter
map MySQL <- [
      'host'::'localhost',
      'dbtype'::'MySQL',
      'database'::'', // it may be a null string
      'port'::'3306',
      'user'::'root',
      'passwd'::'abc'];
//SQLite
map SQLITE <- [
      'dbtype'::'sqlite',
      'database'::'../includes/meteo.db'];
```

Use example: Check a connection to MySQL

```
if (self testConnection[ params::MySQL]){
      write "Connection is OK" ;
}else{
      write "Connection is false" ;
}
```

Action: select [params:: connection_parameter, select::select_string]

The action creates a connection to a DBMS and executes the select statement. If the connection fails then it throws a [GamaRuntimeException].

- **Return value** : if the connection succeed, it returns a list with three elements:
  - The first element is a list of column name.
  - The second element is a list of column type.
  - The third element is a data set.
- **Arguments**
  - params: (type = map) map containing the connection parameters
  - select: (type = string) the SQL request

Examples:

```
map PARAMS <- ['dbtype'::'sqlite','database'::'../includes/meteo.db'];
list t <- [];
...
reflex test {
  set t <- list(self select[params::PARAMS, select::"SELECT * FROM points ;"]);
}
```

Action: executeUpdate [params:: connection_parameter, updateComm:: update_string]

The action creates a connection to the DBMS and executes an update command (create/insert/delete/drop). If the connection is fail then it throws a [GamaRuntimeException].

- **Return value** : if the connection succeed, it returns an integer that is the number of lines in the table affected by the update

- **Arguments**
  - params: (type = map) map containing the connection parameters
  - select: (type = string) the SQL update command

Example: Table creation

```
map PARAMS <- ['dbtype'::'sqlite','database'::'../includes/meteo.db'];
do executeUpdate params: PARAMS
                updateComm: "CREATE TABLE registration " +
                      "(id INTEGER PRIMARY KEY, " +
                      " first TEXT NOT NULL, " +
                      " last TEXT NOT NULL, " +
                      " age INTEGER);";
}
```

Example: Insert data into table

```
do executeUpdate params: PARAMS
                updateComm: "INSERT INTO registration " +
                    "VALUES (101, 'Mahnaz', 'Fatma', 25);";
}
```

Example: Update data

```
do executeUpdate params: PARAMS
                updateComm: "UPDATE Registration " +
                    "SET age = 30 WHERE id in (100, 101)";
}
```

Example: Delete record

```
do executeUpdate params: PARAMS
                updateComm: "DELETE FROM registration WHERE id=100 ";
}
```

Example: Drop table

```
do executeUpdate params: PARAMS
                updateComm: "DROP TABLE registration";
}
```

AgentDB

AgentBD is a built-in species, it supports behaviors that look like actions in SQLSKILL but it is a bit different from SQLSKILL: an AgentDB agent uses only one connection for several actions. It means that AgentDB creates a connection to a DBMS and keeps that connection open for its later operations with the DBMS.

Map of connection parameters

In the actions defined in the AgentDB, a parameter containing the connection parameters is requiered. It is a map with following key::value pairs:

Key	Description
dbtype	DBMS type value. Its value is a string. we must use "sqlserver" when we want to connect

	to a sqlerver. That is the same for "sqlite" or "mysql" (ignore case sensitive)
host	Host name or IP address of data server. it is absent when we work with SQlite.
port	Port of connection. It is absent when we work with SQLite.
database	Name of database. It is file name include path when we work with SQLite.
user	Username. It is absent when we work with SQLite.
passwd	Password. It is absent when we work with SQLite.

Action: testConnection [params:: connection_parameter]

This action tests whether the connection is possible to the DBMS.

- **Return value** : it returns a boolean value:
  - true: the agent can connect to DBMS.
  - false: the agent cannot connect to DBMS.
- **Arguments**
  - params: (type = map) map containing the connection parameters

Example: test of a possible connection to a DB

```
// MySQL connection parameter
let MySQL type:map <- [
  'host'::'localhost',
  'dbtype'::'MySQL',
  'database'::'',
  'port'::'3306',
  'user'::'root',
  'passwd'::'abc'];
if (self testConnection[ params::MySQL]){
  write "The connection is possible" ;
}else{
  write "The connection is not possible" ;
}
```

Action: connect [params:: connection_parameter]

The action connect creates a connection to a DBMS. If the connection can be established then it will assign the connection object into a internal attribute of the AgentDB species (conn) otherwise it throws a [GamaRuntimeException]. Note: The current current connection must be closed before creating a new one.

- **Arguments**
  - params: (type = map) map containing the connection parameters

Example: Establish a connection to SQLServer

```
// SQLSERVER connection parameter
let SQLSERVER type:map <- [
  'host'::'localhost',
  'dbtype'::'sqlserver',
  'database'::'BPH',
  'port'::'1433',
  'user'::'sa',
  'passwd'::'abc'];
// make connection
do connect params::SQLSERVER;
```

Action: isConnected [ ]

The action isConnected tests whether the agent connection is open.

- **Return value** : it returns a boolean value:
  - true: the agent has already a connection to a DBMS
  - false: the agent has no connection to a DBMS.
- **Arguments**
  - params: (type = map) map containing the connection parameters

Example

```
if (self isConnected []){
  write "It already has a connection";
}else {
  do connect params: SQLITE;
}
```

Action: close [ ]

The action close closes the current database connection of the agent. If the agent has no database connection then it throws a [GamaRuntimeException]. **Example:**

```
if (self isConnected []){
  do close;
}
```

Action: getParameter [ ]

The action getParameter will return a map object containing parameters of the current database connection if the agent has a database connection. If it do not have an open connection, it throws a [GamaRuntimeException].

- **Return value** : a map containing connection parameters

Example

```
if (self isConnected []){
  write self getParameter[];
}
```

Action: select [select:: select_string]

The action select executes the select statement by using the current database connection of the agent. If the database connection does not exist or the select statement fails then it throws a [GamaRuntimeException], otherwise it returns a list with three elements.

- **Return value** : a list of three elements:
  - The first element is a list of column name.
  - The second element is a list of column type.
  - The third element is a data set.
- **Arguments**
  - select (type = string): the select query

Example

```
if (self isConnected[]){
  let t value: self select[select::"SELECT id_point, temp_min FROM points ;"];
  write t;
}else {
  write "Error: You must establish a connections before select!";
}
```

Action: executeUpdate [updateComm:: update_string]

The action executeUpdate executes an update command (create/insert/delete/drop) by using the current database connection of the agent. If the database connection does not exist or the update command fails then it throws a [GamaRuntimeException] otherwise it returns an integer value.

- **Return value** : an integer that is the number of lines in the table affected by the update
- **Arguments**
  - updateComm (type = string): update command string

Example: Table creation

```
let PARAMS type:map <-
  ['dbtype'::'sqlite','database'::'../includes/meteo.db'];
do connect params: PARAMS;
do executeUpdate updateComm: "CREATE TABLE registration " +
          "(id INTEGER PRIMARY KEY, " +
          " first TEXT NOT NULL, " +
          " last TEXT NOT NULL, " +
          " age INTEGER);";
}
```

Example: Insertion of data into a table

```
do executeUpdate updateComm: "INSERT INTO registration " +
                "VALUES (101, 'Mahnaz', 'Fatma', 25);";
}
```

Example: Update of data

```
do executeUpdate updateComm: "UPDATE Registration " +
                    "SET age = 30 WHERE id in (100, 101)";
}
```

Example: Deletion of a record

```
do executeUpdate updateComm: "DELETE FROM registration WHERE id=100 ";
}
```

Example: Drop table

```
do executeUpdate updateComm: "DROP TABLE registration";
}
```

Using SQL features to define environment or create species

In Gama, we can use results of select action of SQLSKILL or AgentDB to create species or define boundary of environment in the same way we are doing it with shape files.

Define the boundary of the environment from database

We can select geometry data from MySQL, SQLite or SQLServer [DataBase] Management Systems and define the environment boundary by using the query result. We have to follow the two following steps to define the boundary of an environment.

- **Step 1:** specify select query by declaration a map object with keys as below:

Key	Description
dbtype	DBMS type value. Its value is a string. we must use "sqlserver" when we want to connect to a sqlerver. That is the same for "sqlite" or "mysql" (ignore case sensitive)
host	Host name or IP address of data server. it is absent when we work with SQlite.
port	Port of connection. It is absent when we work with SQLite.
database	Name of database. It is file name include path when we work with SQLite.
user	Username. It is absent when we work with SQLite.
passwd	Password. It is absent when we work with SQLite.
select	Selection string

Example:

```
global {
  map BOUNDS <-
    ['dbtype'::'sqlite','database'::'../includes/bph.sqlite',
      "select"::"select geometry from bph where id_2=38253 or id_2=38254;"];
}
```

- **Step 2:** define boundary of environment by using the map object in first step.

```
environnent bounds: BOUNDS ;
```

Note: We can do similarly if we work with MySQL server or SQLServer. **But with SQLServer** , we must convert Geometry format in SQLServer to binary format. For example, we want to use [MidiPyrenees] region (ID_1=1004) from table FRA_ADM2 as a boundary of simulation environment:

```
global {
  map BOUNDS <-
    ['host'::'localhost',
    'dbtype'::'SQLSERVER',
    'port'::'1433',
    'database'::'BPH',`
    'user'::'test',
    'passwd'::'abc',
    'select'::'select geom.STAsBinary() as geo from FRA_ADM2 WHERE ID_1=1004'];
}
environment bounds: BOUNDS ;
```

Create agents from the result of a `select` action

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1** : Define a species with SQLSKILL or AgentDB

```
entities {
  species agt_with_DB skills: SQLSKILL {
  {
    //insert your descriptions here
  }
  ...
}
```

- **Step 2** : Define a connection and select paramaters

```
global {
  map PARAMS <- ['dbtype'::'sqlite','database'::'../includes/bph.sqlite'];
  string LOCATIONS <-
    'select ID_4, Name_4, geometry from bph where id_2=38253 or id_2=38254;';
...
}
```

- **Step 3** : Create agents by using select result

```
global {
  ...
  init {
    let resSelect type: list <- [];
    create agt_with_DB number: 1 {
      set resSelect <- list(self select [params:: PARAMS, select:: LOCATIONS]);
    }
    create locations from: resSelect
            with:[ id:: "id_4", custom_name:: "name_4", geo::"geometry"]  {
      set shape value: geo;
    }
  }
...
}
```

Driving Skill

Description

- plug-in: simtools.gaml.extensions.traffic
- author: Patrick Taillandier, Javier Gil Quijano and Philippe Caillou

Introduction

This extensions of GAMA is a new skill that extends the moving skill and that provides new moving actions that take into account the traffic jam and the number of lanes of roads. This skill is implemented in the simtools.gaml.extensions.traffic plug-in.

driving skill

Define a species that uses the driving skill

Example of declaration:

```
entities {
  species car skills: [driving]
  {
    //insert your descriptions here
  }
  ...
}
```

Agents with such a skill will automatically be provided with the variables of the moving skill ("speed, heading, destination (r/o)") and the following variables : "living_space, lanes_attribute, tolerance, obstacle_species". They will also be provided with the action of the moving skill ("move, goto, wander, follow") and the following action: "moveTraffic".

variables

living_space

- float, the min distance between the agent and an obstacle (in meter).

lanes_attribute

- string, the name of the attribut of the road agent that determine the number of road lanes.

tolerance

- float, the tolerance distance used for the computation (in meter).

obstacle_species

- list of species, the list of species that are considered as obstacles.

action

goto

moves the agent towards the target passed in the arguments. When moving on a road section, an agent cannot pass through **n** obstacles excepts if the number of lanes for this road section is equal or higher than **n+1** .

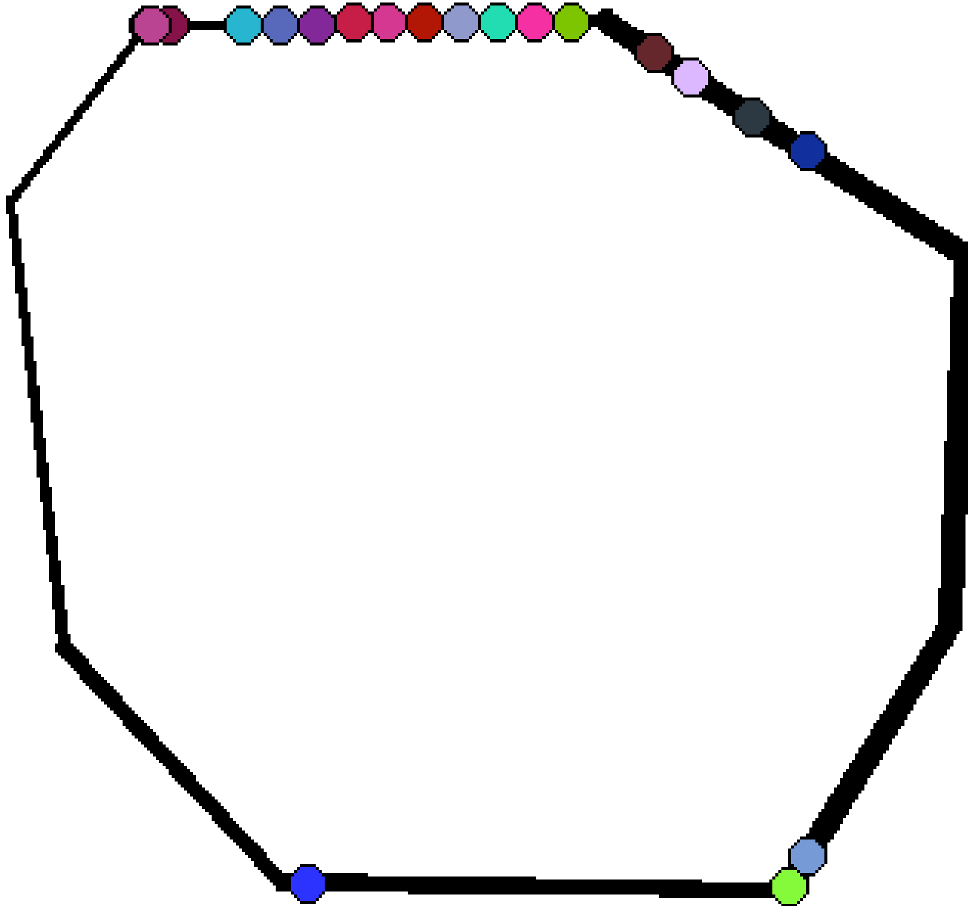
- → **target** : point or agent, mandatory, the location or entity towards which to move.
- → **speed** : float, optional, the speed to use for this move (replaces the current value of speed).
- → **on** : list, agent, graph, geometry, optional, that restrains this move (the agent moves inside this geometry).
- → **return_path** : bool, optional, if true, the action returns the path followed.
- → **living_space** : float, optional, min distance between the agent and an obstacle (replaces the current value of living_space).
- → **tolerance** : float, optional, tolerance distance used for the computation (replaces the current value of tolerance).
- → **lanes_attribute** : string, optional, the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)
- → **return_path** : bool, optional, if true, the action returns the path followed
- ← return: null or the path followed if **return_path** is set to *true*

```
do gotoTraffic target: one_of (list (species (self))) speed: speed * 2 on: road_network
living_space: 2.0;
```

Model examples

Three models based on this plug-in are available in GAMA 1.5.1: RoadTrafficSimple , RoadTrafficComplex and RoadTrafficCity . These models are defined in the driving_traffic project.

- RoadTrafficSimple



- RoadTrafficCity