GAMA 1.8 User Manual

Srirama Bhamidipati

Updated: 2018-07-15

Contents

Notice

- The content of this manual is from gama-platform.org website. I have only modified and edited very small portions of the content to give it a book format.
- This is a modified content and is not a 100% reproduction. If you do not find what you are looking for, go to the main website.
- I thank the Team of GAMA-Platform for giving me the permission to reproduce their content.

Cheers !

Srirama Bhamidipati
Delft, Netherlands
2018

GAMA 1.8
User Manual

GAMA is a modeling and simulation development environment

for building spatially explicit agent-based simulations

Original content owner:

www.gama-platform.org

June 2018

Modified and composed in a book format by:

Srirama Bhamidipati

Delft, Netherlands

Figure 1:

Chapter 1

Java version

Due to changes in the libraries used by GAMA 1.7 and 1.8, this version now **requires JDK/JVM 1.8** to run. Please note that GAMA **has not been tested with JDK 1.9 and 1.10**.

1.1 Changes between 1.6.1 and 1.7/1.8 that can influence the dynamics of models

- Initialization order between the initialization of variables and the execution of the `init` block in grids init -> vars in 1.6.1 / vars -> init in 1.7
- Initialization order of agents -> now, the init block of the agents are not executed at the end of the global init, but during it. put a sample model to explain the order of creation and its differences
- Initialization of vars to their default value map ? list ?
- Systematic casting and verification of types give examples
- header of CSV files: be careful, in GAMA 1.7, if the first line is detected as a header, it is not read when the file is casted as a matrix (so the first row of the matrix is not the header, but the first line of data) gives examples
- the step of batch experiments is now executed after all repetitions of simulations are done (not after each one). They can however be still accessed using the attributes `simulations` (see Batch.gaml in Models Library)
- signal and diffuse have been merged into a single statement
- facets do not accept a space between their identifier and the `:` anymore.
- simplification of equation/solve statements and deprecation of old facets
- in FIPA skill, `content`is replaced everywhere with `contents`
- in FIPA skill, `receivers` is replaced everywhere with `to`
- in FIPA skill, `messages` is replaced by `mailbox`
- The pseudo-attribute `user_location` has been removed (not deprecated, unfortunately) and replaced by the "unit" `#user_location`.
- The actions called by an `event` layer do not need anymore to define `point` and `list<agent>` arguments to receive the mouse location and the list of agents selected. Instead, they can now use `#user_location` and they have to compute the selected agents by themselves (using an arbitrary function).
- The random number generators now better handle seeding (larger range), but it can change the series of values previously obtained from a given seed in 1.6.1
- all models now have a starting_date and a current_date. They then dont begin at an hypothetical "zero" date, but at the epoch date defined by ISO 8601 (1970/1/1). It should not change models that dont rely on dates, except that:
- the #year (and its nicknames #y, #years) and #month (and its nickname #month) do not longer have a default value (of resp. 30 days and 360 days). Instead, they are always evaluated against the

current_date of the model. If no starting_date is defined, the values of #month and #year will then depend on the sequence of months and year since epoch day.

- `as_time`, `as_system_time`, `as_date` and `as_system_date` have been removed

Chapter 2

Enhancements in 1.7/1.8

2.1 Simulations

- simulations can now be run in parallel withing an experiment (with their outputs, displays, etc.)
- batch experiments inherit from this possibility and can now run their repetitions in parallel too.
- concurrency between agents is now possible and can be controlled on a species/grid/ask basis (from multi-threaded concurrency to complete parallelism within a species/grid or between the targets of an `ask` statement)

2.2 Language

- `gama` : a new immutable agent that can be invoked to change preferences or access to platform-only properties (like `machine-time`)
- `abort`: a new behavior (like `reflex` or `init`) that is executed once when the agent is about to die
- `try` and `catch` statements now provide a robust way to catch errors happening in the simulations.
- `super` (instead of `self`) and `invoke` (instead of `do`) can now be used to call an action defined in a parent species.
- `date` : new data type that offers the possibility to use a real calendar, to define a `starting_date` and to query a `current_date` from a simulation, to parse dates from date files or to output them in custom formats. Dates can be added, subtracted, compared. Various new operators (`minus_months`, etc.) allow for a fine manipulation of their data. Time units (`#sec`, `#s`, `#mn`, `#minute`, `#h`, `#hour`, `#day`, etc.) can be used in conjunction with them. Interval of dates (date1 to date2) can be created and used as a basis for loops, etc. Various simple operators allow for defining conditions based on the current_date (after(date1), before(date2), since(date1), etc.).
- `font` type allows to define fonts more precisely in `draw` statements
- BDI control architecture for agents
- file management, new operators, new statements, new skills(?), new built-in variables, files can now download their contents from the web by using standard http: https: addresses instead of file paths.
- The `save` can now directly manipulate files and … save them. So something like `save shape_file("bb.shp", my_agents collect each.shape);` is possible. In addition, a new facet `attributes` allows to define complex attributes to be saved.
- `assert` has a simpler syntax and can be used in any behaviour to raise an error if a condition is not met.
- `test` is a new type of experiments (`experiment aaa type: test ...`), equivalent to a `batch` with an exhaustive search method, which automatically displays the status of tests found in the model.
- new operators (`sum_of`, `product_of`, etc.)

- casting of files works
- co-modeling (importation of micro-models that can be managed within a macro-model)
- populations of agents can now be easily exported to CSV files using the `save` statement
- Simple `messaging` skill between agents

- Terminal commands can now be issued from within GAMA using the `console` operator
- New `status` statement allows to change the text of the status.
- light statement in 3D display provides the possibility to custom your lights (point lights, direction lights, spot lights)
- Displays can now inherit from other displays (facets `parent` and `virtual` to describe abstract displays)
- `on_change:` facet for attributes/parameters allows to define a sequence of statements to run whenever the value changes.
- `species` and `experiment` now support the `virtual` boolean facet (virtual species can not be instantiated, and virtual experiments do not show up).
- `experiment` now supports the `auto_run` boolean facet (to run automatically when launched)

2.3 Data importation

- draw of complex shapes through obj file
- new types fo files are taken into account: geotiff and dxf
- viewers for common files
- navigator: better overview of model files and their support files, addition of plugin models

2.4 Editor

- doc on built-in elements, templates, shortcuts to common tasks, hyperlinks to files used
- improvement in time, gathering of infos/todos
- warnings can be removed from model files

2.5 Headless

- A new option `-validate path/to/dir` allows to run a complete validation of all the models in the directory

2.6 Models library:

- New models (make a list)

2.7 Preferences

- For performances and bug fixes in displays
- For charts defaults

2.8 Simulation displays

- OpenGL displays should be up to 3 times faster in rendering
- fullscreen mode for displays (ESC key)
- CTRL+O for overlay and CTRL+L for layers side controls
- cleaner OpenGL displays (less garbage, better drawing of lines, rotation helper, sticky ROI, etc.)
- possibility to use a new OpenGl pipeline and to define keystoning parameters (for projections)
- faster java2D displays (esp. on zoom)
- better user interaction (mouse move, hover, key listener)
- a whole new set of charts
- getting values when moving the mouse on charts
- possibility to declare `permanent layout: + #splitted`, `#horizontal`, `#vertical`, `#stacked` in the `output` section to automatically layout the display view.
- Outputs can now be managed from the "Views" menu. Closed outputs can be reopened.
- Changing simulation names is reflected in their display titles (and it can be dynamic)
- OpenGL displays now handle rotations of 2D and 3D shapes, combinations of textures and colours, and keystoning

2.9 Error view

- Much faster (up to 100x !) display of errors
- Contextual menu to copy the text of errors to clipboard or open the editor on it

2.10 Validation

- Faster validation of multi-file models (x2 approx.)
- Much less memory used compared to 1.6.1 (/10 approx.)
- No more "false positive" errors

2.11 Console

- Interactive console allows to directly interact with agents (experiments, simulations and any agent) and get a direct feedback on the impact of code execution using a new interpreter integrated with the console. Available in the modeling perspective (to interact with the new `gama` agent) as well as the simulation perspective (to interact with the current `experiment` agent).
- Console now accepts colored text output

2.12 Monitor view

- monitors can have colors
- monitors now have contextual menus depending on the value displayed (save as CSV, inspect, browse…)

2.13 GAMA-wide online help on the language

- A global search engine is now available in the top-right corner of the GAMA window to look for GAML idioms

2.14 Serialization

- Serialize simulations and replay them (to come)
- Serialization and deserialization of agents between simulations (to come)

2.15 Allow TCP, UDP and MQQT communications between agents in different simulations (to come)

Chapter 3

Operators (A to A)

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

3.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

3.2

3.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

3.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

3.5 Operators

3.5.1 -

3.5.1.1 Possible use:

- - (float) —> float
- - (int) —> int
- - (point) —> point
- float - matrix —> matrix
- - (float , matrix) —> matrix
- float - int —> float
- - (float , int) —> float
- matrix - int —> matrix
- - (matrix , int) —> matrix
- int - float —> float
- - (int , float) —> float
- int - int —> int
- - (int , int) —> int
- container - container —> list
- - (container , container) —> list
- matrix - matrix —> matrix
- - (matrix , matrix) —> matrix
- date - int —> date
- - (date , int) —> date
- point - point —> point
- - (point , point) —> point
- geometry - geometry —> geometry
- - (geometry , geometry) —> geometry
- rgb - int —> rgb
- - (rgb , int) —> rgb
- geometry - float —> geometry
- - (geometry , float) —> geometry
- point - int —> point
- - (point , int) —> point
- list - unknown —> list
- - (list , unknown) —> list
- map - map —> map
- - (map , map) —> map
- date - date —> float
- - (date , date) —> float
- species - agent —> list
- - (species , agent) —> list
- rgb - rgb —> rgb
- - (rgb , rgb) —> rgb
- date - float —> date
- - (date , float) —> date
- point - float —> point
- - (point , float) —> point
- float - float —> float
- - (float , float) —> float

- `geometry - container<geometry>` —> `geometry`
- `-` (`geometry` , `container<geometry>`) —> `geometry`
- `matrix - float` —> `matrix`
- `-` (`matrix` , `float`) —> `matrix`
- `map - pair` —> `map`
- `-` (`map` , `pair`) —> `map`
- `int - matrix` —> `matrix`
- `-` (`int` , `matrix`) —> `matrix`

3.5.1.2 Result:

If it is used as an unary operator, it returns the opposite of the operand. Returns the difference of the two operands.

3.5.1.3 Comment:

The behavior of the operator depends on the type of the operands.

3.5.1.4 Special cases:

- if both operands are containers and the right operand is empty, - returns the left operand
- if the left operand is a species and the right operand is an agent of the species, - returns a list containing all the agents of the species minus this agent
- if both operands are numbers, performs a normal arithmetic difference and returns a float if one of them is a float.

```
int var10 <- 1 - 1; // var10 equals 0
```

- if both operands are containers, returns a new list in which all the elements of the right operand have been removed from the left one

```
list<int> var11 <- [1,2,3,4,5,6] - [2,4,9]; // var11 equals [1,3,5,6]
list<int> var12 <- [1,2,3,4,5,6] - [0,8]; // var12 equals [1,2,3,4,5,6]
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date minus the given number as duration (in seconds)

```
date var13 <- date('2000-01-01') - 86400; // var13 equals date('1999-12-31')
```

- if both operands are points, returns their difference (coordinates per coordinates).

```
point var14 <- {1, 2} - {4, 5}; // var14 equals {-3.0, -3.0}
```

- if both operands are a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries

```
geometry var15 <- geom1 - geom2; // var15 equals a geometry corresponding to difference between geom1 and geo
```

- if one operand is a color and the other an integer, returns a new color resulting from the subtraction of each component of the color with the right operand

```
rgb var16 <- rgb([255, 128, 32]) - 3; // var16 equals rgb([252,125,29])
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) reduced by the right-hand operand distance

```
geometry var17 <- shape - 5; // var17 equals a geometry corresponding to the geometry of the agent applying th
```

- if the left operand is a list and the right operand is an object of any type (except list), - returns a list containing the elements of the left operand minus all the occurrences of this object

```
list<int> var18 <- [1,2,3,4,5,6] - 2; // var18 equals [1,3,4,5,6]
list<int> var19 <- [1,2,3,4,5,6] - 0; // var19 equals [1,2,3,4,5,6]
```

- if both operands are dates, returns the duration in seconds between date2 and date1. To obtain a more precise duration, in milliseconds, use milliseconds_between(date1, date2)

```
float var20 <- date('2000-01-02') - date('2000-01-01'); // var20 equals 86400
```

- if both operands are colors, returns a new color resulting from the subtraction of the two operands, component by component

```
rgb var21 <- rgb([255, 128, 32]) - rgb('red'); // var21 equals rgb([0,128,32])
```

- 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.

```
point var22 <- {1, 2} - 4.5; // var22 equals {-3.5, -2.5, -4.5}
point var23 <- {1, 2} - 4; // var23 equals {-3.0,-2.0,-4.0}
```

- 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

```
geometry var24 <- rectangle(10,10) - [circle(2), square(2)]; // var24 equals rectangle(10,10) - (circle(2) +
```

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic difference of the number with each element of the matrix (results are float if the number is a float.

```
matrix var25 <- 3.5 - matrix([[2,5],[3,4]]); // var25 equals matrix([[1.5,-1.5],[0.5,-0.5]])
```

3.5.1.5 Examples:

```
float var0 <- 1.0 - 1; // var0 equals 0.0
float var1 <- 3.7 - 1.2; // var1 equals 2.5
float var2 <- 3 - 1.2; // var2 equals 1.8
int var3 <- - (-56); // var3 equals 56
map var4 <- ['a'::1,'b'::2] - ['b'::2]; // var4 equals ['a'::1]
map var5 <- ['a'::1,'b'::2] - ['b'::2,'c'::3]; // var5 equals ['a'::1]
point var6 <- -{3.0,5.0}; // var6 equals {-3.0,-5.0}
point var7 <- -{1.0,6.0,7.0}; // var7 equals {-1.0,-6.0,-7.0}
map var8 <- ['a'::1,'b'::2] - ('b'::2); // var8 equals ['a'::1]
map var9 <- ['a'::1,'b'::2] - ('c'::3); // var9 equals ['a'::1,'b'::2]
```

3.5.1.6 See also:

+, *, /, -, milliseconds_between,

3.5.2 :

3.5.2.1 Possible use:

- `unknown : unknown` —> `unknown`
- `:` (`unknown` , `unknown`) —> `unknown`

3.5.2.2 See also:

?,

3.5.3 : :

3.5.3.1 Possible use:

- `any expression :: any expression` —> `pair`
- `::` (`any expression` , `any expression`) —> `pair`

3.5.3.2 Result:

produces a new pair combining the left and the right operands

3.5.3.3 Special cases:

- nil is not acceptable as a key (although it is as a value). If such a case happens, :: will throw an appropriate error

---

3.5.4 !

3.5.4.1 Possible use:

- `!` (`bool`) —> `bool`

3.5.4.2 Result:

opposite boolean value.

3.5.4.3 Special cases:

- if the parameter is not boolean, it is casted to a boolean value.

3.5.4.4 Examples:

```
bool var0 <- ! (true); // var0 equals false
```

3.5.4.5 See also:

bool, and, or,

3.5.5 !=

3.5.5.1 Possible use:

- `unknown != unknown —> bool`
- `!= (unknown , unknown) —> bool`
- `int != float —> bool`
- `!= (int , float) —> bool`
- `date != date —> bool`
- `!= (date , date) —> bool`
- `float != int —> bool`
- `!= (float , int) —> bool`
- `float != float —> bool`
- `!= (float , float) —> bool`

3.5.5.2 Result:

true if both operands are different, false otherwise

3.5.5.3 Examples:

```
bool var0 <- [2,3] != [2,3]; // var0 equals false
bool var1 <- [2,4] != [2,3]; // var1 equals true
bool var2 <- 3 != 3.0; // var2 equals false
bool var3 <- 4 != 4.7; // var3 equals true
bool var4 <- #now != #now minus_hours 1; // var4 equals true
bool var5 <- 3.0 != 3; // var5 equals false
bool var6 <- 4.7 != 4; // var6 equals true
bool var7 <- 3.0 != 3.0; // var7 equals false
bool var8 <- 4.0 != 4.7; // var8 equals true
```

3.5.5.4 See also:

=, >, <, >=, <=,

3.5.6 ?

3.5.6.1 Possible use:

- `bool ? any expression` —> `unknown`
- `?` (`bool` , `any expression`) —> `unknown`

3.5.6.2 Result:

It is used in combination with the : operator: 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 :

3.5.6.3 Comment:

These functional tests can be combined together.

3.5.6.4 Examples:

```
list<string> var0 <- [10, 19, 43, 12, 7, 22] collect ((each > 20) ? 'above' : 'below'); // var0 equals ['below
```

3.5.6.5 See also:

:,

3.5.7 /

3.5.7.1 Possible use:

- `point / float` —> `point`
- `/` (`point` , `float`) —> `point`
- `float / float` —> `float`
- `/` (`float` , `float`) —> `float`
- `int / int` —> `float`
- `/` (`int` , `int`) —> `float`
- `rgb / float` —> `rgb`
- `/` (`rgb` , `float`) —> `rgb`
- `float / int` —> `float`
- `/` (`float` , `int`) —> `float`
- `rgb / int` —> `rgb`
- `/` (`rgb` , `int`) —> `rgb`
- `matrix / int` —> `matrix`

- `/` (`matrix` , `int`) —> `matrix`
- `point / int` —> `point`
- `/` (`point` , `int`) —> `point`
- `matrix / float` —> `matrix`
- `/` (`matrix` , `float`) —> `matrix`
- `matrix / matrix` —> `matrix`
- `/` (`matrix` , `matrix`) —> `matrix`
- `int / float` —> `float`
- `/` (`int` , `float`) —> `float`

3.5.7.2 Result:

Returns the division of the two operands.

3.5.7.3 Special cases:

- if the right-hand operand is equal to zero, raises a "Division by zero" exception
- if the left operand is a point, returns a new point with coordinates divided by the right operand

```
point var0 <- {5, 7.5} / 2.5; // var0 equals {2, 3}
point var1 <- {2,5} / 4; // var1 equals {0.5,1.25}
```

- if both operands are numbers (float or int), performs a normal arithmetic division and returns a float.

```
float var2 <- 3 / 5.0; // var2 equals 0.6
```

- if one operand is a color and the other a double, returns a new color resulting from the division of each component of the color by the right operand. The result on each component is then truncated.

```
rgb var3 <- rgb([255, 128, 32]) / 2.5; // var3 equals rgb([102,51,13])
```

- if one operand is a color and the other an integer, returns a new color resulting from the division of each component of the color by the right operand

```
rgb var4 <- rgb([255, 128, 32]) / 2; // var4 equals rgb([127,64,16])
```

3.5.7.4 See also:

*, +, -,

3.5.8 .

3.5.8.1 Possible use:

- `agent . any expression` —> `unknown`
- `.` (`agent` , `any expression`) —> `unknown`
- `matrix . matrix` —> `matrix`
- `.` (`matrix` , `matrix`) —> `matrix`

3.5.8.2 Result:

It has two different uses: it can be the dot product between 2 matrices or return an evaluation of the expression (right-hand operand) in the scope the given agent.

3.5.8.3 Special cases:

- if the agent is nil or dead, throws an exception
- if the left operand is an agent, it evaluates of the expression (right-hand operand) in the scope the given agent

```
unknown var0 <- agent1.location; // var0 equals the location of the agent agent1
```

- if both operands are matrix, returns the dot product of them

```
matrix var1 <- matrix([[1,1],[1,2]]) . matrix([[1,1],[1,2]]); // var1 equals matrix([[2,3],[3,5]])
```

3.5.9 ^

3.5.9.1 Possible use:

- `float ^ int —> float`
- `^ (float , int) —> float`
- `float ^ float —> float`
- `^ (float , float) —> float`
- `int ^ int —> float`
- `^ (int , int) —> float`
- `int ^ float —> float`
- `^ (int , float) —> float`

3.5.9.2 Result:

Returns the value (always a float) of the left operand raised to the power of the right operand.

3.5.9.3 Special cases:

- if the right-hand operand is equal to 0, returns 1
- if it is equal to 1, returns the left-hand operand.
- Various examples of power

```
float var1 <- 2 ^ 3; // var1 equals 8.0
```

3.5.9.4 Examples:

```
float var0 <- 4.84 ^ 0.5; // var0 equals 2.2
```

3.5.9.5 See also:

*, sqrt,

3.5.10 @

Same signification as at

3.5.11 *

3.5.11.1 Possible use:

- float * float —> float
- * (float , float) —> float
- int * matrix —> matrix
- * (int , matrix) —> matrix
- int * int —> int
- * (int , int) —> int
- point * float —> point
- * (point , float) —> point
- rgb * int —> rgb
- * (rgb , int) —> rgb
- int * float —> float
- * (int , float) —> float
- geometry * float —> geometry
- * (geometry , float) —> geometry
- geometry * point —> geometry
- * (geometry , point) —> geometry
- matrix * float —> matrix
- * (matrix , float) —> matrix
- matrix * matrix —> matrix
- * (matrix , matrix) —> matrix
- float * matrix —> matrix
- * (float , matrix) —> matrix
- matrix * int —> matrix
- * (matrix , int) —> matrix
- float * int —> float
- * (float , int) —> float
- point * int —> point
- * (point , int) —> point
- point * point —> float
- * (point , point) —> float

3.5.11.2 Result:

Returns the product of the two operands.

3.5.11.3 Special cases:

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic product of the number with each element of the matrix (results are float if the number is a float.

```
matrix<float> m <- (3.5 * matrix([[2,5],[3,4]]));  //m equals matrix([[7.0,17.5],[10.5,14]])
```

- if both operands are numbers (float or int), performs a normal arithmetic product and returns a float if one of them is a float.

```
int var2 <- 1 * 1; // var2 equals 1
```

- if one operand is a color and the other an integer, returns a new color resulting from the product of each component of the color with the right operand (with a maximum value at 255)

```
rgb var3 <- rgb([255, 128, 32]) * 2; // var3 equals rgb([255,255,64])
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) scaled by the right-hand operand coefficient

```
geometry var4 <- circle(10) * 2; // var4 equals circle(20)
geometry var5 <- (circle(10) * 2).location with_precision 9; // var5 equals (circle(20)).location with_preci
float var6 <- (circle(10) * 2).height with_precision 9; // var6 equals (circle(20)).height with_precision 9
```

- if the left-hand operand is a geometry and the right-hand operand a point, returns a geometry corresponding to the left-hand operand (geometry, agent, point) scaled by the right-hand operand coefficients in the 3 dimensions

```
geometry var7 <- shape * {0.5,0.5,2}; // var7 equals a geometry corresponding to the geometry of the agent ap
```

- if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number

```
point var8 <- {2,5} * 4; // var8 equals {8.0, 20.0}
point var9 <- {2, 4} * 2.5; // var9 equals {5.0, 10.0}
```

- if both operands are points, returns their scalar product

```
float var10 <- {2,5} * {4.5, 5}; // var10 equals 34.0
```

3.5.11.4 Examples:

```
float var0 <- 2.5 * 2; // var0 equals 5.0
```

3.5.11.5 See also:

/, +, -,

3.5.12 +

3.5.12.1 Possible use:

- point + point —> point
- + (point , point) —> point
- int + matrix —> matrix
- + (int , matrix) —> matrix
- date + int —> date
- + (date , int) —> date
- float + int —> float
- + (float , int) —> float
- matrix + int —> matrix
- + (matrix , int) —> matrix
- matrix + matrix —> matrix
- + (matrix , matrix) —> matrix
- float + float —> float
- + (float , float) —> float
- rgb + rgb —> rgb
- + (rgb , rgb) —> rgb
- int + int —> int
- + (int , int) —> int
- string + unknown —> string
- + (string , unknown) —> string
- map + pair —> map
- + (map , pair) —> map
- container + unknown —> list
- + (container , unknown) —> list
- point + float —> point
- + (point , float) —> point
- matrix + float —> matrix
- + (matrix , float) —> matrix
- geometry + float —> geometry
- + (geometry , float) —> geometry
- geometry + geometry —> geometry
- + (geometry , geometry) —> geometry
- string + string —> string
- + (string , string) —> string
- container + container —> container
- + (container , container) —> container
- map + map —> map
- + (map , map) —> map
- point + int —> point
- + (point , int) —> point
- date + float —> date
- + (date , float) —> date
- float + matrix —> matrix

- `+` (`float` , `matrix`) —> `matrix`
- `rgb` `+` `int` —> `rgb`
- `+` (`rgb` , `int`) —> `rgb`
- `date` `+` `string` —> `string`
- `+` (`date` , `string`) —> `string`
- `int` `+` `float` —> `float`
- `+` (`int` , `float`) —> `float`
- `+` (`geometry`, `float`, `int`) —> `geometry`
- `+` (`geometry`, `float`, `int`, `int`) —> `geometry`

3.5.12.2 Result:

Returns the sum, union or concatenation of the two operands.

3.5.12.3 Special cases:

- if one of the operands is nil, + throws an error
- if both operands are species, returns a special type of list called meta-population
- if both operands are points, returns their sum.

```
point var6 <- {1, 2} + {4, 5}; // var6 equals {5.0, 7.0}
```

- if the left-hand operand is a geometry and the right-hand operands a float and an integer, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand

```
geometry var7 <- circle(5) + (5,32); // var7 equals circle(10)
```

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic sum of the number with each element of the matrix (results are float if the number is a float.

```
matrix var8 <- 3.5 + matrix([[2,5],[3,4]]); // var8 equals matrix([[5.5,8.5],[6.5,7.5]])
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date plus the given number as duration (in seconds)

```
date var9 <- date('2000-01-01') + 86400; // var9 equals date('2000-01-02')
```

- if both operands are colors, returns a new color resulting from the sum of the two operands, component by component

```
rgb var10 <- rgb([255, 128, 32]) + rgb('red'); // var10 equals rgb([255,128,32])
```

- if both operands are numbers (float or int), performs a normal arithmetic sum and returns a float if one of them is a float.

```
int var11 <- 1 + 1; // var11 equals 2
```

- if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one beind casted into a string)

```
string var12 <- "hello " + 12; // var12 equals "hello 12"
```

- if the left-hand operand is a geometry and the right-hand operands a float, an integer and one of #round, #square or #flat, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand and a flat, square or round end cap style

```
geometry var13 <- circle(5) + (5,32,#round); // var13 equals circle(10)
```

- if the right operand is an object of any type (except a container), + returns a list of the elements of the left operand, to which this object has been added

```
list<int> var14 <- [1,2,3,4,5,6] + 2; // var14 equals [1,2,3,4,5,6,2]
list<int> var15 <- [1,2,3,4,5,6] + 0; // var15 equals [1,2,3,4,5,6,0]
```

- if the 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.

```
point var16 <- {1, 2} + 4; // var16 equals {5.0, 6.0,4.0}
point var17 <- {1, 2} + 4.5; // var17 equals {5.5, 6.5,4.5}
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the right-hand operand distance. The number of segments used by default is 8 and the end cap style is #round

```
geometry var18 <- circle(5) + 5; // var18 equals circle(10)
```

- if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the union between both geometries

```
geometry var19 <- geom1 + geom2; // var19 equals a geometry corresponding to union between geom1 and geom2
```

- if both operands are list, +returns the concatenation of both lists.

```
list<int> var20 <- [1,2,3,4,5,6] + [2,4,9]; // var20 equals [1,2,3,4,5,6,2,4,9]
list<int> var21 <- [1,2,3,4,5,6] + [0,8]; // var21 equals [1,2,3,4,5,6,0,8]
```

- if one operand is a color and the other an integer, returns a new color resulting from the sum of each component of the color with the right operand

```
rgb var22 <- rgb([255, 128, 32]) + 3; // var22 equals rgb([255,131,35])
```

3.5.12.4 Examples:

```
float var0 <- 1.0 + 1; // var0 equals 2.0
float var1 <- 1.0 + 2.5; // var1 equals 3.5
map var2 <- ['a'::1,'b'::2] + ('c'::3); // var2 equals ['a'::1,'b'::2,'c'::3]
map var3 <- ['a'::1,'b'::2] + ('c'::3); // var3 equals ['a'::1,'b'::2,'c'::3]
map var4 <- ['a'::1,'b'::2] + ['c'::3]; // var4 equals ['a'::1,'b'::2,'c'::3]
map var5 <- ['a'::1,'b'::2] + [5::3.0]; // var5 equals ['a'::1,'b'::2,5::3.0]
```

3.5.12.5 See also:

-, *, /,

3.5.13 <

3.5.13.1 Possible use:

- `string < string —> bool`
- `< (string , string) —> bool`
- `int < int —> bool`
- `< (int , int) —> bool`
- `float < float —> bool`
- `< (float , float) —> bool`
- `float < int —> bool`
- `< (float , int) —> bool`
- `date < date —> bool`
- `< (date , date) —> bool`
- `point < point —> bool`
- `< (point , point) —> bool`
- `int < float —> bool`
- `< (int , float) —> bool`

3.5.13.2 Result:

true if the left-hand operand is less than the right-hand operand, false otherwise.

3.5.13.3 Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' < 'aeb'; // var0 equals true
```

- if both operands are points, returns true if and only if the 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.

```
bool var1 <- {5,7} < {4,6}; // var1 equals false
bool var2 <- {5,7} < {4,8}; // var2 equals false
```

3.5.13.4 Examples:

```
bool var3 <- 3 < 7; // var3 equals true
bool var4 <- 3.5 < 7.6; // var4 equals true
bool var5 <- 3.5 < 7; // var5 equals true
bool var6 <- #now < #now minus_hours 1; // var6 equals false
```

```
bool var7 <- 3 < 2.5; // var7 equals false
```

3.5.13.5 See also:

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

3.5.14 <=

3.5.14.1 Possible use:

- `string` **`<=`** `string` —> `bool`
- **`<=`** (`string` , `string`) —> `bool`
- `float` **`<=`** `float` —> `bool`
- **`<=`** (`float` , `float`) —> `bool`
- `float` **`<=`** `int` —> `bool`
- **`<=`** (`float` , `int`) —> `bool`
- `int` **`<=`** `int` —> `bool`
- **`<=`** (`int` , `int`) —> `bool`
- `int` **`<=`** `float` —> `bool`
- **`<=`** (`int` , `float`) —> `bool`
- `date` **`<=`** `date` —> `bool`
- **`<=`** (`date` , `date`) —> `bool`
- `point` **`<=`** `point` —> `bool`
- **`<=`** (`point` , `point`) —> `bool`

3.5.14.2 Result:

true if the left-hand operand is less or equal than the right-hand operand, false otherwise.

3.5.14.3 Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var5 <- 'abc' <= 'aeb'; // var5 equals true
```

- if both operands are points, returns true if and only if the 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.

```
bool var6 <- {5,7} <= {4,6}; // var6 equals false
bool var7 <- {5,7} <= {4,8}; // var7 equals false
```

3.5.14.4 Examples:

```
bool var0 <- 3.5 <= 3.5; // var0 equals true
```

```
bool var1 <- 7.0 <= 7; // var1 equals true
bool var2 <- 3 <= 7; // var2 equals true
bool var3 <- 3 <= 2.5; // var3 equals false
bool var4 <- #now <= #now minus_hours 1; // var4 equals false
```

3.5.14.5 See also:

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

3.5.15 <>

Same signification as !=

3.5.16 =

3.5.16.1 Possible use:

- `float` = `int` —> `bool`
- = (`float` , `int`) —> `bool`
- `unknown` = `unknown` —> `bool`
- = (`unknown` , `unknown`) —> `bool`
- `int` = `int` —> `bool`
- = (`int` , `int`) —> `bool`
- `float` = `float` —> `bool`
- = (`float` , `float`) —> `bool`
- `int` = `float` —> `bool`
- = (`int` , `float`) —> `bool`
- `date` = `date` —> `bool`
- = (`date` , `date`) —> `bool`

3.5.16.2 Result:

returns true if both operands are equal, false otherwise returns true if both operands are equal, false otherwise

3.5.16.3 Special cases:

- if both operands are any kind of objects, returns true if they are identical (i.e., the same object) or equal (comparisons between nil values are permitted)

```
bool var0 <- [2,3] = [2,3]; // var0 equals true
```

3.5.16.4 Examples:

```
bool var1 <- 4.7 = 4; // var1 equals false
bool var2 <- 4 = 5; // var2 equals false
```

```
bool var3 <- 4.5 = 4.7; // var3 equals false
bool var4 <- 3 = 3.0; // var4 equals true
bool var5 <- 4 = 4.7; // var5 equals false
bool var6 <- #now = #now minus_hours 1; // var6 equals false
```

3.5.16.5 See also:

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

3.5.17 >

3.5.17.1 Possible use:

- `point` **`>`** `point` —> `bool`
- **`>`** (`point` , `point`) —> `bool`
- `float` **`>`** `int` —> `bool`
- **`>`** (`float` , `int`) —> `bool`
- `string` **`>`** `string` —> `bool`
- **`>`** (`string` , `string`) —> `bool`
- `float` **`>`** `float` —> `bool`
- **`>`** (`float` , `float`) —> `bool`
- `date` **`>`** `date` —> `bool`
- **`>`** (`date` , `date`) —> `bool`
- `int` **`>`** `float` —> `bool`
- **`>`** (`int` , `float`) —> `bool`
- `int` **`>`** `int` —> `bool`
- **`>`** (`int` , `int`) —> `bool`

3.5.17.2 Result:

true if the left-hand operand is greater than the right-hand operand, false otherwise.

3.5.17.3 Special cases:

- if one of the operands is nil, returns false
- if both operands are points, returns true if and only if the 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.

```
bool var0 <- {5,7} > {4,6}; // var0 equals true
bool var1 <- {5,7} > {4,8}; // var1 equals false
```

- if both operands are String, uses a lexicographic comparison of two strings

```
bool var2 <- 'abc' > 'aeb'; // var2 equals false
```

3.5.17.4 Examples:

```
bool var3 <- 3.5 > 7; // var3 equals false
bool var4 <- 3.5 > 7.6; // var4 equals false
bool var5 <- #now > #now minus_hours 1; // var5 equals true
bool var6 <- 3 > 2.5; // var6 equals true
bool var7 <- 3 > 7; // var7 equals false
```

3.5.17.5 See also:

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

3.5.18 >=

3.5.18.1 Possible use:

- `float >= int` —> `bool`
- `>= (float , int)` —> `bool`
- `point >= point` —> `bool`
- `>= (point , point)` —> `bool`
- `float >= float` —> `bool`
- `>= (float , float)` —> `bool`
- `int >= int` —> `bool`
- `>= (int , int)` —> `bool`
- `string >= string` —> `bool`
- `>= (string , string)` —> `bool`
- `date >= date` —> `bool`
- `>= (date , date)` —> `bool`
- `int >= float` —> `bool`
- `>= (int , float)` —> `bool`

3.5.18.2 Result:

true if the left-hand operand is greater or equal than the right-hand operand, false otherwise.

3.5.18.3 Special cases:

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

```
bool var0 <- {5,7} >= {4,6}; // var0 equals true
bool var1 <- {5,7} >= {4,8}; // var1 equals false
```

- if both operands are string, uses a lexicographic comparison of the two strings

```
bool var2 <- 'abc' >= 'aeb'; // var2 equals false
bool var3 <- 'abc' >= 'abc'; // var3 equals true
```

3.5.18.4 Examples:

```
bool var4 <- 3.5 >= 7; // var4 equals false
bool var5 <- 3.5 >= 3.5; // var5 equals true
bool var6 <- 3 >= 7; // var6 equals false
bool var7 <- #now >= #now minus_hours 1; // var7 equals true
bool var8 <- 3 >= 2.5; // var8 equals true
```

3.5.18.5 See also:

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

3.5.19 `abs`

3.5.19.1 Possible use:

- **abs** (float) —> float
- **abs** (int) —> int

3.5.19.2 Result:

Returns the absolute value of the operand (so a positive int or float depending on the type of the operand).

3.5.19.3 Examples:

```
float var0 <- abs (200 * -1 + 0.5); // var0 equals 199.5
int var1 <- abs (-10); // var1 equals 10
int var2 <- abs (10); // var2 equals 10
```

3.5.20 `accumulate`

3.5.20.1 Possible use:

- container **accumulate** any expression —> list
- **accumulate** (container , any expression) —> list

3.5.20.2 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

3.5.20.3 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 left-hand operand elements.

3.5.20.4 Examples:

```
list var0 <- [a1,a2,a3] accumulate (each neighbors_at 10); // var0 equals a flat list of all the neighbors of
list<int> var1 <- [1,2,4] accumulate ([2,4]); // var1 equals [2,4,2,4,2,4]
list<int> var2 <- [1,2,4] accumulate (each * 2); // var2 equals [2,4,8]
```

3.5.20.5 See also:

collect,

3.5.21 acos

3.5.21.1 Possible use:

- `acos` (`float`) —> `float`
- `acos` (`int`) —> `float`

3.5.21.2 Result:

Returns the value (in the interval [0,180], in decimal degrees) of the arccos of the operand (which should be in [-1,1]).

3.5.21.3 Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

3.5.21.4 Examples:

```
float var0 <- acos (0); // var0 equals 90.0
```

3.5.21.5 See also:

asin, atan, cos,

3.5.22 `action`

3.5.22.1 Possible use:

- `action` (`any`) —> `action`

3.5.22.2 Result:

Casts the operand into the type action

3.5.23 `add_days`

Same signification as plus_days

3.5.24 `add_edge`

3.5.24.1 Possible use:

- `graph add_edge pair` —> `graph`
- `add_edge` (`graph` , `pair`) —> `graph`

3.5.24.2 Result:

add an edge between a source vertex and a target vertex (resp. the left and the right element of the pair operand)

3.5.24.3 Comment:

if the edge already exists, the graph is unchanged

3.5.24.4 Examples:

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

3.5.24.5 See also:

add_node, graph,

3.5.25 `add_hours`

Same signification as plus_hours

3.5.26 add_minutes

Same signification as plus_minutes

3.5.27 add_months

Same signification as plus_months

3.5.28 add_ms

Same signification as plus_ms

3.5.29 add_node

3.5.29.1 Possible use:

- `graph` **`add_node`** `geometry` —> `graph`
- **`add_node`** (`graph` , `geometry`) —> `graph`

3.5.29.2 Result:

adds a node in a graph.

3.5.29.3 Examples:

```
graph var0 <- graph add_node node(0) ; // var0 equals the graph with node(0)
```

3.5.29.4 See also:

add_edge, graph,

3.5.30 add_point

3.5.30.1 Possible use:

- `geometry` **`add_point`** `point` —> `geometry`
- **`add_point`** (`geometry` , `point`) —> `geometry`

3.5.30.2 Result:

A new geometry resulting from the addition of the right point (coordinate) to the left-hand geometry. Note that adding a point to a line or polyline will always return a closed contour. Also note that the position at which the added point will appear in the geometry is not necessarily the last one, as points are always ordered in a clockwise fashion in geometries

3.5.30.3 Examples:

```
geometry var0 <- polygon([{10,10},{10,20},{20,20}]) add_point {20,10}; // var0 equals polygon([{10,10},{10
```

3.5.31 add_seconds

Same signification as +

3.5.32 add_weeks

Same signification as plus_weeks

3.5.33 add_years

Same signification as plus_years

3.5.34 adjacency

3.5.34.1 Possible use:

- **adjacency** (graph) —> matrix<float>

3.5.34.2 Result:

adjacency matrix of the given graph.

3.5.35 after

3.5.35.1 Possible use:

- **after** (date) —> bool
- any expression **after** date —> bool
- **after** (any expression , date) —> bool

3.5.35.2 Result:

Returns true if the current_date of the model is strictly after the date passed in argument. Synonym of 'current_date > argument'. Can be used in its composed form with 2 arguments to express the lower boundary for the computation of a frequency. Note that only dates strictly after this one will be tested against the frequency

3.5.35.3 Examples:

```
reflex when: after(starting_date) {}   // this reflex will always be run after the first step reflex when: fa
```

3.5.36 agent

3.5.36.1 Possible use:

- `agent` (`any`) —> `agent`

3.5.36.2 Result:

Casts the operand into the type agent

3.5.37 agent_closest_to

3.5.37.1 Possible use:

- `agent_closest_to` (`unknown`) —> `agent`

3.5.37.2 Result:

An agent, the closest to the operand (casted as a geometry).

3.5.37.3 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.

3.5.37.4 Examples:

```
agent var0 <- agent_closest_to(self); // var0 equals the closest agent to the agent applying the operator.
```

3.5.37.5 See also:

neighbors_at, neighbors_of, agents_inside, agents_overlapping, closest_to, inside, overlapping,

---

3.5.38 agent_farthest_to

3.5.38.1 Possible use:

- `agent_farthest_to` (`unknown`) —> `agent`

3.5.38.2 Result:

An agent, the farthest to the operand (casted as a geometry).

3.5.38.3 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.

3.5.38.4 Examples:

```
agent var0 <- agent_farthest_to(self); // var0 equals the farthest agent to the agent applying the operator.
```

3.5.38.5 See also:

neighbors_at, neighbors_of, agents_inside, agents_overlapping, closest_to, inside, overlapping, agent_closest_to, farthest_to,

---

3.5.39 agent_from_geometry

3.5.39.1 Possible use:

- `path` **`agent_from_geometry`** `geometry` —> `agent`
- **`agent_from_geometry`** (`path` , `geometry`) —> `agent`

3.5.39.2 Result:

returns the agent corresponding to given geometry (right-hand operand) in the given path (left-hand operand).

3.5.39.3 Special cases:

- if the left-hand operand is nil, returns nil

3.5.39.4 Examples:

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

3.5.39.5 See also:

path,

3.5.40 agents_at_distance

3.5.40.1 Possible use:

- `agents_at_distance` (`float`) —> `list`

3.5.40.2 Result:

A list of agents situated at a distance lower than the right argument.

3.5.40.3 Examples:

```
list var0 <- agents_at_distance(20); // var0 equals all the agents (excluding the caller) which distance to t
```

3.5.40.4 See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping, at_distance,

---

3.5.41 agents_inside

3.5.41.1 Possible use:

- `agents_inside` (`unknown`) —> `list<agent>`

3.5.41.2 Result:

A list of agents covered by the operand (casted as a geometry).

3.5.41.3 Examples:

```
list<agent> var0 <- agents_inside(self); // var0 equals the agents that are covered by the shape of the agent
```

3.5.41.4 See also:

agent_closest_to, agents_overlapping, closest_to, inside, overlapping,

---

3.5.42 agents_overlapping

3.5.42.1 Possible use:

- `agents_overlapping` (`unknown`) —> `list<agent>`

3.5.42.2 Result:

A list of agents overlapping the operand (casted as a geometry).

3.5.42.3 Examples:

```
list<agent> var0 <- agents_overlapping(self); // var0 equals the agents that overlap the shape of the agent a
```

3.5.42.4 See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping, at_distance,

---

3.5.43 all_pairs_shortest_path

3.5.43.1 Possible use:

- `all_pairs_shortest_path` (`graph`) —> `matrix<int>`

3.5.43.2 Result:

returns the successor matrix of shortest paths between all node pairs (rows: source, columns: target): a cell (i,j) will thus contains the next node in the shortest path between i and j.

3.5.43.3 Examples:

```
matrix<int> var0 <- all_pairs_shortest_paths(my_graph); // var0 equals shortest_paths_matrix will contain a
```

3.5.44 alpha_index

3.5.44.1 Possible use:

- `alpha_index` (`graph`) —> `float`

3.5.44.2 Result:

returns the alpha index of the graph (measure of connectivity which evaluates the number of cycles in a graph in comparison with the maximum number of cycles. The higher the alpha index, the more a network is connected: alpha = nb_cycles / (2*S-5) - planar graph)

3.5.44.3 Examples:

```
float var1 <- alpha_index(graphEpidemio); // var1 equals the alpha index of the graph
```

3.5.44.4 See also:

beta_index, gamma_index, nb_cycles, connectivity_index,

3.5.45 among

3.5.45.1 Possible use:

- `int` **`among`** `container` —> `list`
- **`among`** (`int` , `container`) —> `list`

3.5.45.2 Result:

Returns a list of length the value of the left-hand operand, containing random elements from the right-hand operand. As of GAMA 1.6, the order in which the elements are returned can be different than the order in which they appear in the right-hand container

3.5.45.3 Special cases:

- if the right-hand operand is empty, among returns a new empty list. If it is nil, it throws an error.
- if the left-hand operand is greater than the length of the right-hand operand, among returns the right-hand operand (converted as a list). If it is smaller or equal to zero, it returns an empty list

3.5.45.4 Examples:

```
list<int> var0 <- 3 among [1,2,4,3,5,7,6,8]; // var0 equals [1,2,8] (for example)
list var1 <- 3 among g2; // var1 equals [node6,node11,node7]
list var2 <- 3 among list(node); // var2 equals [node1,node11,node4]
list<int> var3 <- 1 among [1::2,3::4]; // var3 equals 2 or 4
```

3.5.46 and

3.5.46.1 Possible use:

- `bool` **and** `any expression` —> `bool`
- **and** (`bool` , `any expression`) —> `bool`

3.5.46.2 Result:

a bool value, equal to the logical and between the left-hand operand and the right-hand operand.

3.5.46.3 Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like (1 and 0) is accepted and returns false.

3.5.46.4 See also:

bool, or, !,

3.5.47 and

3.5.47.1 Possible use:

- `predicate` **and** `predicate` —> `predicate`
- **and** (`predicate` , `predicate`) —> `predicate`

3.5.47.2 Result:

create a new predicate from two others by including them as subintentions

3.5.47.3 Examples:

```
predicate1 and predicate2
```

3.5.48 angle_between

3.5.48.1 Possible use:

- **angle_between** (`point`, `point`, `point`) —> `float`

3.5.48.2 Result:

the angle between vectors P0P1 and P0P2 (P0, P1, P2 being the three point operands)

3.5.48.3 Examples:

```
float var0 <- angle_between({5,5},{10,5},{5,10}); // var0 equals 90
```

3.5.49 any

Same signification as one_of

3.5.50 any_location_in

3.5.50.1 Possible use:

- **any_location_in** (geometry) —> point

3.5.50.2 Result:

A point inside (or touching) the operand-geometry.

3.5.50.3 Examples:

```
point var0 <- any_location_in(square(5)); // var0 equals a point in the square, for example : {3,4.6}.
```

3.5.50.4 See also:

closest_points_with, farthest_point_to, points_at,

3.5.51 any_point_in

Same signification as any_location_in

3.5.52 append_horizontally

3.5.52.1 Possible use:

- matrix **append_horizontally** matrix —> matrix
- **append_horizontally** (matrix , matrix) —> matrix
- matrix **append_horizontally** matrix —> matrix
- **append_horizontally** (matrix , matrix) —> matrix

3.5.52.2 Result:

A matrix resulting from the concatenation of the rows of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

3.5.52.3 Examples:

```
matrix var0 <- matrix([[1.0,2.0],[3.0,4.0]]) append_horizontally matrix([[1,2],[3,4]]); // var0 equals mat
```

3.5.53 append_vertically

3.5.53.1 Possible use:

- `matrix` **`append_vertically`** `matrix` —> `matrix`
- **`append_vertically`** (`matrix` , `matrix`) —> `matrix`
- `matrix` **`append_vertically`** `matrix` —> `matrix`
- **`append_vertically`** (`matrix` , `matrix`) —> `matrix`

3.5.53.2 Result:

A matrix resulting from the concatenation of the columns of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

3.5.53.3 Examples:

```
matrix var0 <- matrix([[1,2],[3,4]]) append_vertically matrix([[1,2],[3,4]]); // var0 equals matrix([[1,2,
```

3.5.54 arc

3.5.54.1 Possible use:

- **`arc`** (`float`, `float`, `float`) —> `geometry`
- **`arc`** (`float`, `float`, `float`, `bool`) —> `geometry`

3.5.54.2 Result:

An arc, which radius is equal to the first operand, heading to the second and amplitude the third An arc, which radius is equal to the first operand, heading to the second, amplitude to the third and a boolean indicating whether to return a linestring or a polygon to the fourth

3.5.54.3 Comment:

the center of the arc is by default the location of the current agent in which has been called this operator. This operator returns a polygon by default.the center of the arc is by default the location of the current agent in which has been called this operator.

3.5.54.4 Special cases:

- returns a point if the radius operand is lower or equal to 0.
- returns a point if the radius operand is lower or equal to 0.

3.5.54.5 Examples:

```
geometry var0 <- arc(4,45,90); // var0 equals a geometry as an arc of radius 4, in a direction of 45Â° and an a
geometry var1 <- arc(4,45,90, false); // var1 equals a geometry as an arc of radius 4, in a direction of 45Â° a
```

3.5.54.6 See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

3.5.55 around

3.5.55.1 Possible use:

- `float` **`around`** `unknown` —> `geometry`
- **`around`** (`float` , `unknown`) —> `geometry`

3.5.55.2 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.

3.5.55.3 Special cases:

- returns a circle geometry of radius right-operand if the left-operand is nil

3.5.55.4 Examples:

```
geometry var0 <- 10 around circle(5); // var0 equals the ring geometry between 5 and 10.
```

3.5.55.5 See also:

circle, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

3.5.56 as

3.5.56.1 Possible use:

- `unknown as msi.gaml.types.IType` —> `unknown`
- `as` (`unknown` , `msi.gaml.types.IType`) —> `unknown`

3.5.56.2 Result:

casting of the first argument into a given type

3.5.56.3 Comment:

It is equivalent to the application of the type operator on the left operand.

3.5.56.4 Examples:

```
int var0 <- 3.5 as int; // var0 equals int(3.5)
```

3.5.57 as_4_grid

3.5.57.1 Possible use:

- `geometry as_4_grid point` —> `matrix`
- `as_4_grid` (`geometry` , `point`) —> `matrix`

3.5.57.2 Result:

A matrix of square geometries (grid with 4-neighborhood) with dimension given by the right-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

3.5.57.3 Examples:

```
matrix var0 <- self as_4_grid {10, 5}; // var0 equals the matrix of square geometries (grid with 4-neighborho
```

3.5.57.4 See also:

as_grid, as_hexagonal_grid,

3.5.58 as_distance_graph

3.5.58.1 Possible use:

- `container` **`as_distance_graph`** `map` —> `graph`
- **`as_distance_graph`** (`container` , `map`) —> `graph`
- `container` **`as_distance_graph`** `float` —> `graph`
- **`as_distance_graph`** (`container` , `float`) —> `graph`
- **`as_distance_graph`** (`container`, `float`, `species`) —> `graph`

3.5.58.2 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).

3.5.58.3 Comment:

as_distance_graph is more efficient for a list of points than as_intersection_graph.

3.5.58.4 Examples:

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

3.5.58.5 See also:

as_intersection_graph, as_edge_graph,

3.5.59 as_driving_graph

3.5.59.1 Possible use:

- `container` **`as_driving_graph`** `container` —> `graph`
- **`as_driving_graph`** (`container` , `container`) —> `graph`

3.5.59.2 Result:

creates a graph from the list/map of edges given as operand and connect the node to the edge

3.5.59.3 Examples:

```
as_driving_graph(road,node) --: build a graph while using the road agents as edges and the node agents as no
```

3.5.59.4 See also:

as_intersection_graph, as_distance_graph, as_edge_graph,

3.5.60 as_edge_graph

3.5.60.1 Possible use:

- **as_edge_graph** (map) —> graph
- **as_edge_graph** (container) —> graph
- container **as_edge_graph** float —> graph
- **as_edge_graph** (container , float) —> graph

3.5.60.2 Result:

creates a graph from the list/map of edges given as operand

3.5.60.3 Special cases:

- if the operand is a list and a tolerance (max distance in meters to consider that 2 points are the same node) is given, the graph will be built with elements of the list as edges and two edges will be connected by a node if the distance between their extremity (first or last points) are at distance lower or equal to the tolerance

```
graph var0 <- as_edge_graph([line([{1,5},{12,45}]),line([{13,45},{34,56}])],1); // var0 equals a graph wit
```

- if the operand is a map, the graph will be built by creating edges from pairs of the map

```
graph var1 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]); // var1 equals a graph with these three ver
```

- if the operand is a list, the graph will be built with elements of the list as edges

```
graph var2 <- as_edge_graph([line([{1,5},{12,45}]),line([{12,45},{34,56}])]); // var2 equals a graph with t
```

3.5.60.4 See also:

as_intersection_graph, as_distance_graph,

3.5.61 as_grid

3.5.61.1 Possible use:

- geometry **as_grid** point —> matrix
- **as_grid** (geometry , point) —> matrix

3.5.61.2 Result:

A matrix of square geometries (grid with 8-neighborhood) with dimension given by the right-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

3.5.61.3 Examples:

```
matrix var0 <- self as_grid {10, 5}; // var0 equals a matrix of square geometries (grid with 8-neighborhood)
```

3.5.61.4 See also:

as_4_grid, as_hexagonal_grid,

3.5.62 as_hexagonal_grid

3.5.62.1 Possible use:

- `geometry` **`as_hexagonal_grid`** `point` —> `list<geometry>`
- **`as_hexagonal_grid`** (`geometry` , `point`) —> `list<geometry>`

3.5.62.2 Result:

A list of geometries (hexagonal) corresponding to the hexagonal tesselation of the first operand geometry

3.5.62.3 Examples:

```
list<geometry> var0 <- self as_hexagonal_grid {10, 5}; // var0 equals list of geometries (hexagonal) corresp
```

3.5.62.4 See also:

as_4_grid, as_grid,

3.5.63 as_int

3.5.63.1 Possible use:

- `string` **`as_int`** `int` —> `int`
- **`as_int`** (`string` , `int`) —> `int`

3.5.63.2 Result:

parses the string argument as a signed integer in the radix specified by the second argument.

3.5.63.3 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

3.5.63.4 Examples:

```
int var0 <- '20' as_int 10; // var0 equals 20
int var1 <- '20' as_int 8; // var1 equals 16
int var2 <- '20' as_int 16; // var2 equals 32
int var3 <- '1F' as_int 16; // var3 equals 31
int var4 <- 'hello' as_int 32; // var4 equals 18306744
```

3.5.63.5 See also:

int,

3.5.64 as_intersection_graph

3.5.64.1 Possible use:

- `container` **`as_intersection_graph`** `float` —> `graph`
- **`as_intersection_graph`** (`container` , `float`) —> `graph`

3.5.64.2 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).

3.5.64.3 Comment:

as_intersection_graph is more efficient for a list of geometries (but less accurate) than as_distance_graph.

3.5.64.4 Examples:

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

3.5.64.5 See also:

as_distance_graph, as_edge_graph,

3.5.65 as_map

3.5.65.1 Possible use:

- container **as_map** any expression —> map
- **as_map** (container , any expression) —> map

3.5.65.2 Result:

produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand

3.5.65.3 Comment:

the right-hand operand should be a pair

3.5.65.4 Special cases:

- if the left-hand operand is nil, as_map throws an error.

3.5.65.5 Examples:

```
map<int,int> var0 <- [1,2,3,4,5,6,7,8] as_map (each::(each * 2)); // var0 equals [1::2, 2::4, 3::6, 4::8, 5:
map<int,int> var1 <- [1::2,3::4,5::6] as_map (each::(each * 2)); // var1 equals [2::4, 4::8, 6::12]
```

3.5.66 as_matrix

3.5.66.1 Possible use:

- unknown **as_matrix** point —> matrix
- **as_matrix** (unknown , point) —> matrix

3.5.66.2 Result:

casts the left operand into a matrix with right operand as preferred size

3.5.66.3 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.

3.5.66.4 Special cases:

- if the right operand is nil, as_matrix is equivalent to the matrix operator

3.5.66.5 See also:

matrix,

3.5.67 as_path

3.5.67.1 Possible use:

- `list<geometry> as_path graph —> path`
- `as_path (list<geometry> , graph) —> path`

3.5.67.2 Result:

create a graph path from the list of shape

3.5.67.3 Examples:

```
path var0 <- [road1,road2,road3] as_path my_graph; // var0 equals a path road1->road2->road3 of my_graph
```

3.5.68 asin

3.5.68.1 Possible use:

- `asin (float) —> float`
- `asin (int) —> float`

3.5.68.2 Result:

the arcsin of the operand

3.5.68.3 Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

3.5.68.4 Examples:

```
float var0 <- asin (0); // var0 equals 0.0
float var1 <- asin (90); // var1 equals #nan
```

3.5.68.5 See also:

acos, atan, sin,

3.5.69 at

3.5.69.1 Possible use:

- container<KeyType,ValueType> **at** KeyType —> ValueType
- **at** (container<KeyType,ValueType> , KeyType) —> ValueType
- string **at** int —> string
- **at** (string , int) —> string

3.5.69.2 Result:

the element at the right operand index of the container

3.5.69.3 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

3.5.69.4 Special cases:

- 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 list or a matrix, at returns the element at the index specified by the right operand

```
int var0 <- [1, 2, 3] at 2; // var0 equals 3
point var1 <- [{1,2}, {3,4}, {5,6}] at 0; // var1 equals {1.0,2.0}
```

3.5.69.5 Examples:

```
string var2 <- 'abcdef' at 0; // var2 equals 'a'
```

3.5.69.6 See also:

contains_all, contains_any,

3.5.70 at_distance

3.5.70.1 Possible use:

- container<agent> **at_distance** float —> list<geometry>
- **at_distance** (container<agent> , float) —> list<geometry>

3.5.70.2 Result:

A list of agents or geometries among the left-operand list that are located at a distance <= the right operand from the caller agent (in its topology)

3.5.70.3 Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] at_distance 20; // var0 equals the agents of the list located at a dist
```

3.5.70.4 See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping,

---

3.5.71 at_location

3.5.71.1 Possible use:

- geometry **at_location** point —> geometry
- **at_location** (geometry , point) —> geometry

3.5.71.2 Result:

A geometry resulting from the tran of a translation to the right-hand operand point of the left-hand operand (geometry, agent, point)

3.5.71.3 Examples:

```
geometry var0 <- self at_location {10, 20}; // var0 equals the geometry resulting from a translation to the l
```

3.5.72 atan

3.5.72.1 Possible use:

- **atan** (float) —> float
- **atan** (int) —> float

3.5.72.2 Result:

Returns the value (in the interval [-90,90], in decimal degrees) of the arctan of the operand (which can be any real number).

3.5.72.3 Examples:

```
float var0 <- atan (1); // var0 equals 45.0
```

3.5.72.4 See also:

acos, asin, tan,

3.5.73 atan2

3.5.73.1 Possible use:

- `float` **`atan2`** `float` —> `float`
- **`atan2`** (`float` , `float`) —> `float`

3.5.73.2 Result:

the atan2 value of the two operands.

3.5.73.3 Comment:

The function atan2 is the arctangent function with two arguments. The purpose of using two arguments instead of one is to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle, which is not possible for the single-argument arctangent function.

3.5.73.4 Examples:

```
float var0 <- atan2 (0,0); // var0 equals 0.0
```

3.5.73.5 See also:

atan, acos, asin,

3.5.74 attributes

3.5.74.1 Possible use:

- **`attributes`** (`any`) —> `attributes`

3.5.74.2 Result:

Casts the operand into the type attributes

3.5.75 `auto_correlation`

3.5.75.1 Possible use:

- `container` **`auto_correlation`** `int` —> `float`
- **`auto_correlation`** (`container` , `int`) —> `float`

3.5.75.2 Result:

Returns the auto-correlation of a data sequence

Chapter 4

Operators (B to C)

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

4.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

4.2

4.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

4.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

4.5 Operators

4.5.1 BDIPlan

4.5.1.1 Possible use:

- `BDIPlan` (`any`) —> `BDIPlan`

4.5.1.2 Result:

Casts the operand into the type BDIPlan

4.5.2 before

4.5.2.1 Possible use:

- `before` (`date`) —> `bool`
- `any expression before date` —> `bool`
- `before` (`any expression` , `date`) —> `bool`

4.5.2.2 Result:

Returns true if the current_date of the model is strictly before the date passed in argument. Synonym of 'current_date < argument'

4.5.2.3 Examples:

```
reflex when: before(starting_date) {}   // this reflex will never be run
```

4.5.3 beta

4.5.3.1 Possible use:

- `float beta float` —> `float`
- `beta` (`float` , `float`) —> `float`

4.5.3.2 Result:

Returns the beta function with arguments a, b.

4.5.4 beta_index

4.5.4.1 Possible use:

- **beta_index** (graph) —> float

4.5.4.2 Result:

returns the beta index of the graph (Measures the level of connectivity in a graph and is expressed by the relationship between the number of links (e) over the number of nodes (v) : beta = e/v.

4.5.4.3 Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- beta_index(graphEpidemio); // var1 equals the beta index of the graph
```

4.5.4.4 See also:

alpha_index, gamma_index, nb_cycles, connectivity_index,

4.5.5 between

4.5.5.1 Possible use:

- date **between** date —> bool
- **between** (date , date) —> bool
- **between** (float, float, float) —> bool
- **between** (date, date, date) —> bool
- **between** (any expression, date, date) —> bool
- **between** (int, int, int) —> bool

4.5.5.2 Result:

returns true if the first float operand is bigger than the second float operand and smaller than the third float operand

returns true the first integer operand is bigger than the second integer operand and smaller than the third integer operand

4.5.5.3 Special cases:

- returns true if the first operand is between the two dates passed in arguments (both exclusive). The version with 2 arguments compares the current_date with the 2 others

```
bool var0 <- (date('2016-01-01') between(date('2000-01-01'), date('2020-02-02'))); // var0 equals true// //
```

- returns true if the first operand is between the two dates passed in arguments (both exclusive). Can be combined with 'every' to express a frequency between two dates

```
bool var3 <- (date('2016-01-01') between(date('2000-01-01'), date('2020-02-02'))); // var3 equals true// wi
```

4.5.5.4 Examples:

```
bool var6 <- between(5.0, 1.0, 10.0); // var6 equals true
bool var7 <- between(5, 1, 10); // var7 equals true
```

4.5.6 betweenness_centrality

4.5.6.1 Possible use:

- **betweenness_centrality** (graph) —> map

4.5.6.2 Result:

returns a map containing for each vertex (key), its betweenness centrality (value): number of shortest paths passing through each vertex

4.5.6.3 Examples:

```
graph graphEpidemio <- graph([]);
map var1 <- betweenness_centrality(graphEpidemio); // var1 equals the betweenness centrality index of the gr
```

4.5.7 biggest_cliques_of

4.5.7.1 Possible use:

- **biggest_cliques_of** (graph) —> list<list>

4.5.7.2 Result:

returns the biggest cliques of a graph using the Bron-Kerbosch clique detection algorithm

4.5.7.3 Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- biggest_cliques_of (my_graph); // var1 equals the list of the biggest cliques as list
```

4.5.7.4 See also:

maximal_cliques_of,

4.5.8 binomial

4.5.8.1 Possible use:

- `int binomial float —> int`
- `binomial (int , float) —> int`

4.5.8.2 Result:

A value from a random variable following a binomial distribution. The operands represent the number of experiments n and the success probability p.

4.5.8.3 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.

4.5.8.4 Examples:

```
int var0 <- binomial(15,0.6); // var0 equals a random positive integer
```

4.5.8.5 See also:

poisson, gauss,

4.5.9 binomial_coeff

4.5.9.1 Possible use:

- `int binomial_coeff int —> float`
- `binomial_coeff (int , int) —> float`

4.5.9.2 Result:

Returns n choose k as a double. Note the integerization of the double return value.

4.5.10 binomial_complemented

4.5.10.1 Possible use:

- `binomial_complemented (int, int, float) —> float`

4.5.10.2 Result:

Returns the sum of the terms k+1 through n of the Binomial probability density, where n is the number of trials and P is the probability of success in the range 0 to 1.

4.5.11 `binomial_sum`

4.5.11.1 Possible use:

- `binomial_sum` (`int`, `int`, `float`) —> `float`

4.5.11.2 Result:

Returns the sum of the terms 0 through k of the Binomial probability density, where n is the number of trials and p is the probability of success in the range 0 to 1.

4.5.12 `blend`

4.5.12.1 Possible use:

- `rgb blend rgb` —> `rgb`
- `blend` (`rgb` , `rgb`) —> `rgb`
- `blend` (`rgb`, `rgb`, `float`) —> `rgb`

4.5.12.2 Result:

Blend two colors with an optional ratio (c1 * r + c2 * (1 - r)) between 0 and 1

4.5.12.3 Special cases:

- If the ratio is omitted, an even blend is done

```
rgb var1 <- blend(#red, #blue); // var1 equals to a color very close to the purple
```

4.5.12.4 Examples:

```
rgb var3 <- blend(#red, #blue, 0.3); // var3 equals to a color between the purple and the blue
```

4.5.12.5 See also:

rgb, hsb,

4.5.13 bool

4.5.13.1 Possible use:

- `bool` (`any`) —> `bool`

4.5.13.2 Result:

Casts the operand into the type bool

4.5.14 box

4.5.14.1 Possible use:

- `box` (`point`) —> `geometry`
- `box` (`float`, `float`, `float`) —> `geometry`

4.5.14.2 Result:

A box geometry which side sizes are given by the operands.

4.5.14.3 Comment:

the center of the box is by default the location of the current agent in which has been called this operator.the center of the box is by default the location of the current agent in which has been called this operator.

4.5.14.4 Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

4.5.14.5 Examples:

```
geometry var0 <- box(10, 5 , 5); // var0 equals a geometry as a rectangle with width = 10, height = 5 depth= 5.
geometry var1 <- box({10, 5 , 5}); // var1 equals a geometry as a rectangle with width = 10, height = 5 depth=
```

4.5.14.6 See also:

around, circle, sphere, cone, line, link, norm, point, polygon, polyline, square, cube, triangle,

---

4.5.15 brewer_colors

4.5.15.1 Possible use:

- **brewer_colors** (string) —> list<rgb>
- string **brewer_colors** int —> list<rgb>
- **brewer_colors** (string , int) —> list<rgb>

4.5.15.2 Result:

Build a list of colors of a given type (see website http://colorbrewer2.org/) with a given number of classes
Build a list of colors of a given type (see website http://colorbrewer2.org/)

4.5.15.3 Examples:

```
list<rgb> var0 <- list<rgb> colors <- brewer_colors("Pastel1", 10);; // var0 equals a list of 10 sequential c
list<rgb> var1 <- list<rgb> colors <- brewer_colors("OrRd");; // var1 equals a list of 6 blue colors
```

4.5.15.4 See also:

brewer_palettes,

4.5.16 brewer_palettes

4.5.16.1 Possible use:

- **brewer_palettes** (int) —> list<string>
- int **brewer_palettes** int —> list<string>
- **brewer_palettes** (int , int) —> list<string>

4.5.16.2 Result:

returns the list a palette with a given min number of classes and max number of classes) returns the list a palette with a given min number of classes and max number of classes)

4.5.16.3 Examples:

```
list<string> var0 <- list<rgb> colors <- brewer_palettes(5,10);; // var0 equals a list of palettes that are c
list<string> var1 <- list<rgb> colors <- brewer_palettes();; // var1 equals a list of palettes that are compo
```

4.5.16.4 See also:

brewer_colors,

4.5.17 `buffer`

Same signification as +

4.5.18 `build`

4.5.18.1 Possible use:

- `build` (`matrix<float>`) —> `regression`
- `matrix<float>` `build` `string` —> `regression`
- `build` (`matrix<float>` , `string`) —> `regression`

4.5.18.2 Result:

returns the regression build from the matrix data (a row = an instance, the last value of each line is the y value) while using the given method ("GLS" or "OLS"). Usage: build(data,method) returns the regression build from the matrix data (a row = an instance, the last value of each line is the y value) while using the given ordinary least squares method. Usage: build(data)

4.5.18.3 Examples:

```
build(matrix([[1,2,3,4],[2,3,4,2]]),"GLS") matrix([[1,2,3,4],[2,3,4,2]])
```

4.5.19 `ceil`

4.5.19.1 Possible use:

- `ceil` (`float`) —> `float`

4.5.19.2 Result:

Maps the operand to the smallest following integer, i.e. the smallest integer not less than x.

4.5.19.3 Examples:

```
float var0 <- ceil(3); // var0 equals 3.0
float var1 <- ceil(3.5); // var1 equals 4.0
float var2 <- ceil(-4.7); // var2 equals -4.0
```

4.5.19.4 See also:

floor, round,

4.5.20 centroid

4.5.20.1 Possible use:

- `centroid` (`geometry`) —> `point`

4.5.20.2 Result:

Centroid (weighted sum of the centroids of a decomposition of the area into triangles) of the operand-geometry. Can be different to the location of the geometry

4.5.20.3 Examples:

```
point var0 <- centroid(world); // var0 equals the centroid of the square, for example : {50.0,50.0}.
```

4.5.20.4 See also:

any_location_in, closest_points_with, farthest_point_to, points_at,

---

4.5.21 char

4.5.21.1 Possible use:

- `char` (`int`) —> `string`

4.5.21.2 Special cases:

- converts ACSII integer value to character

```
string var0 <- char (34); // var0 equals '"'
```

4.5.22 chi_square

4.5.22.1 Possible use:

- `float chi_square float` —> `float`
- `chi_square` (`float` , `float`) —> `float`

4.5.22.2 Result:

Returns the area under the left hand tail (from 0 to x) of the Chi square probability density function with df degrees of freedom.

4.5.23 chi_square_complemented

4.5.23.1 Possible use:

- `float chi_square_complemented float` —> `float`
- `chi_square_complemented` (`float` , `float`) —> `float`

4.5.23.2 Result:

Returns the area under the right hand tail (from x to infinity) of the Chi square probability density function with df degrees of freedom.

4.5.24 circle

4.5.24.1 Possible use:

- `circle` (`float`) —> `geometry`
- `float circle point` —> `geometry`
- `circle` (`float` , `point`) —> `geometry`

4.5.24.2 Result:

A circle geometry which radius is equal to the first operand, and the center has the location equal to the second operand. A circle geometry which radius is equal to the operand.

4.5.24.3 Comment:

the center of the circle is by default the location of the current agent in which has been called this operator.

4.5.24.4 Special cases:

- returns a point if the operand is lower or equal to 0.
- returns a point if the operand is lower or equal to 0.

4.5.24.5 Examples:

```
geometry var0 <- circle(10,{80,30}); // var0 equals a geometry as a circle of radius 10, the center will be ir
geometry var1 <- circle(10); // var1 equals a geometry as a circle of radius 10.
```

4.5.24.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

4.5.25 clean

4.5.25.1 Possible use:

- **clean** (geometry) —> geometry

4.5.25.2 Result:

A geometry corresponding to the cleaning of the operand (geometry, agent, point)

4.5.25.3 Comment:

The cleaning corresponds to a buffer with a distance of 0.0

4.5.25.4 Examples:

```
geometry var0 <- clean(self); // var0 equals returns the geometry resulting from the cleaning of the geometry
```

4.5.26 clean_network

4.5.26.1 Possible use:

- **clean_network** (list<geometry>, float, bool, bool) —> list<geometry>

4.5.26.2 Result:

A list of polylines corresponding to the cleaning of the first operand (list of polyline geometry or agents), considering the tolerance distance given by the second operand; the third operator is used to define if the operator should as well split the lines at their intersections(true to split the lines); the last operandis used to specify if the operator should as well keep only the main connected component of the network. Usage: clean_network(lines:list of geometries or agents, tolerance: float, split_lines: bool, keepMainConnectedComponent: bool)

4.5.26.3 Comment:

The cleaned set of polylines

4.5.26.4 Examples:

```
list<geometry> var0 <- clean_network(my_road_shapefile.contents, 1.0, true, false); // var0 equals returns
```

4.5.27 closest_points_with

4.5.27.1 Possible use:

- geometry **closest_points_with** geometry —> list<point>
- **closest_points_with** (geometry , geometry) —> list<point>

4.5.27.2 Result:

A list of two closest points between the two geometries.

4.5.27.3 Examples:

```
list<point> var0 <- geom1 closest_points_with(geom2); // var0 equals [pt1, pt2] with pt1 the closest point o
```

4.5.27.4 See also:

any_location_in, any_point_in, farthest_point_to, points_at,

4.5.28 closest_to

4.5.28.1 Possible use:

- container<agent> **closest_to** geometry —> geometry
- **closest_to** (container<agent> , geometry) —> geometry

4.5.28.2 Result:

An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the closest to the operand (casted as a geometry).

4.5.28.3 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.

4.5.28.4 Examples:

```
geometry var0 <- [ag1, ag2, ag3] closest_to(self); // var0 equals return the closest agent among ag1, ag2 and
```

4.5.28.5 See also:

neighbors_at, neighbors_of, inside, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

4.5.29 collect

4.5.29.1 Possible use:

- container **collect** any expression —> list
- **collect** (container , any expression) —> list

4.5.29.2 Result:

returns a new list, in which each element is the evaluation of the right-hand operand.

4.5.29.3 Comment:

collect is similar to accumulate except that accumulate always produces flat lists if the right-hand operand returns a list.In addition, collect can be applied to any container.

4.5.29.4 Special cases:

- if the left-hand operand is nil, collect throws an error

4.5.29.5 Examples:

```
list var0 <- [1,2,4] collect (each *2); // var0 equals [2,4,8]
list var1 <- [1,2,4] collect ([2,4]); // var1 equals [[2,4],[2,4],[2,4]]
list var2 <- [1::2, 3::4, 5::6] collect (each + 2); // var2 equals [4,6,8]
list var3 <- (list(node) collect (node(each).location.x * 2); // var3 equals the list of nodes with their x mu
```

4.5.29.6 See also:

accumulate,

4.5.30 column_at

4.5.30.1 Possible use:

- matrix **column_at** int —> list
- **column_at** (matrix , int) —> list

4.5.30.2 Result:

returns the column at a num_col (right-hand operand)

4.5.30.3 Examples:

```
list var0 <- matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]) column_at 2; /
```

4.5.30.4 See also:

row_at, rows_list,

4.5.31 columns_list

4.5.31.1 Possible use:

- `columns_list` (`matrix`) —> `list<list>`

4.5.31.2 Result:

returns a list of the columns of the matrix, with each column as a list of elements

4.5.31.3 Examples:

```
list<list> var0 <- columns_list(matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el3
```

4.5.31.4 See also:

rows_list,

4.5.32 command

4.5.32.1 Possible use:

- `command` (`string`) —> `string`
- `string command string` —> `string`
- `command` (`string` , `string`) —> `string`
- `command` (`string`, `string`, `msi.gama.util.GamaMap<java.lang.String,java.lang.String>`) —> `string`

4.5.32.2 Result:

command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence '&' is used at the end. In this case, the result of the operator is an empty string. The basic form with only one string in argument uses the directory of the model and does not set any environment variables. Two other forms (with a directory and a map of environment variables) are available. command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence '&' is used at the end. In this case, the result of the operator is an empty string command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence '&' is used at the end. In this case, the

result of the operator is an empty string. The basic form with only one string in argument uses the directory of the model and does not set any environment variables. Two other forms (with a directory and a map of environment variables) are available.

4.5.33 cone

4.5.33.1 Possible use:

- `cone` (`point`) —> `geometry`
- `int cone int` —> `geometry`
- `cone` (`int` , `int`) —> `geometry`

4.5.33.2 Result:

A cone geometry which min and max angles are given by the operands. A cone geometry which min and max angles are given by the operands.

4.5.33.3 Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.the center of the cone is by default the location of the current agent in which has been called this operator.

4.5.33.4 Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

4.5.33.5 Examples:

```
geometry var0 <- cone({0, 45}); // var0 equals a geometry as a cone with min angle is 0 and max angle is 45.
geometry var1 <- cone(0, 45); // var1 equals a geometry as a cone with min angle is 0 and max angle is 45.
```

4.5.33.6 See also:

around, circle, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

4.5.34 cone3D

4.5.34.1 Possible use:

- `float cone3D float` —> `geometry`
- `cone3D` (`float` , `float`) —> `geometry`

4.5.34.2 Result:

A cone geometry which base radius size is equal to the first operand, and which the height is equal to the second operand.

4.5.34.3 Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.

4.5.34.4 Special cases:

- returns a point if the operand is lower or equal to 0.

4.5.34.5 Examples:

```
geometry var0 <- cone3D(10.0,5.0); // var0 equals a geometry as a cone with a base circle of radius 10 and a h
```

4.5.34.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

4.5.35 connected_components_of

4.5.35.1 Possible use:

- `connected_components_of` (`graph`) —> `list<list>`
- `graph connected_components_of bool` —> `list<list>`
- `connected_components_of` (`graph , bool`) —> `list<list>`

4.5.35.2 Result:

returns the connected components of a graph, i.e. the list of all vertices that are in the maximally connected component together with the specified vertex. returns the connected components of a graph, i.e. the list of all edges (if the boolean is true) or vertices (if the boolean is false) that are in the connected components.

4.5.35.3 Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- connected_components_of (my_graph); // var1 equals the list of all the components as list
list<list> var3 <- connected_components_of (my_graph2, true); // var3 equals the list of all the components
```

4.5.35.4 See also:

alpha_index, connectivity_index, nb_cycles,

4.5.36 connectivity_index

4.5.36.1 Possible use:

- **connectivity_index** (graph) —> float

4.5.36.2 Result:

returns a simple connectivity index. This number is estimated through the number of nodes (v) and of sub-graphs (p) : IC = (v - p) /(v - 1).

4.5.36.3 Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- connectivity_index(graphEpidemio); // var1 equals the connectivity index of the graph
```

4.5.36.4 See also:

alpha_index, beta_index, gamma_index, nb_cycles,

4.5.37 container

4.5.37.1 Possible use:

- **container** (any) —> container

4.5.37.2 Result:

Casts the operand into the type container

4.5.38 contains

4.5.38.1 Possible use:

- container<KeyType,ValueType> **contains** unknown —> bool
- **contains** (container<KeyType,ValueType> , unknown) —> bool
- string **contains** string —> bool
- **contains** (string , string) —> bool

4.5.38.2 Result:

true, if the container contains the right operand, false otherwise

4.5.38.3 Comment:

the contains operator behavior depends on the nature of the operand

4.5.38.4 Special cases:

- 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;
- if it is a list or a matrix, contains returns true if the list or matrix contains the right operand

```
bool var0 <- [1, 2, 3] contains 2; // var0 equals true
bool var1 <- [{1,2}, {3,4}, {5,6}] contains {3,4}; // var1 equals true
```

4.5.38.5 Examples:

```
bool var2 <- 'abcded' contains 'bc'; // var2 equals true
```

4.5.38.6 See also:

contains_all, contains_any,

4.5.39 `contains_all`

4.5.39.1 Possible use:

- `string` **`contains_all`** `list` —> `bool`
- **`contains_all`** (`string` , `list`) —> `bool`
- `container` **`contains_all`** `container` —> `bool`
- **`contains_all`** (`container` , `container`) —> `bool`

4.5.39.2 Result:

true if the left operand contains all the elements of the right operand, false otherwise

4.5.39.3 Comment:

the definition of contains depends on the container

4.5.39.4 Special cases:

- if the right operand is nil or empty, contains_all returns true
- if the left-operand is a string, test whether the string contains all the element of the list;

```
bool var0 <- "abcabcabc" contains_all ["ca","xy"]; // var0 equals false
```

4.5.39.5 Examples:

```
bool var1 <- [1,2,3,4,5,6] contains_all [2,4]; // var1 equals true
bool var2 <- [1,2,3,4,5,6] contains_all [2,8]; // var2 equals false
bool var3 <- [1::2, 3::4, 5::6] contains_all [1,3]; // var3 equals false
bool var4 <- [1::2, 3::4, 5::6] contains_all [2,4]; // var4 equals true
```

4.5.39.6 See also:

contains, contains_any,

4.5.40 `contains_any`

4.5.40.1 Possible use:

- `string` **`contains_any`** `list` —> `bool`
- **`contains_any`** (`string` , `list`) —> `bool`
- `container` **`contains_any`** `container` —> `bool`
- **`contains_any`** (`container` , `container`) —> `bool`

4.5.40.2 Result:

true if the left operand contains one of the elements of the right operand, false otherwise

4.5.40.3 Comment:

the definition of contains depends on the container

4.5.40.4 Special cases:

- if the right operand is nil or empty, contains_any returns false

4.5.40.5 Examples:

```
bool var0 <- "abcabcabc" contains_any ["ca","xy"]; // var0 equals true
bool var1 <- [1,2,3,4,5,6] contains_any [2,4]; // var1 equals true
bool var2 <- [1,2,3,4,5,6] contains_any [2,8]; // var2 equals true
bool var3 <- [1::2, 3::4, 5::6] contains_any [1,3]; // var3 equals false
bool var4 <- [1::2, 3::4, 5::6] contains_any [2,4]; // var4 equals true
```

4.5.40.6 See also:

contains, contains_all,

4.5.41 `contains_edge`

4.5.41.1 Possible use:

- `graph` **`contains_edge`** `pair` —> `bool`
- **`contains_edge`** (`graph` , `pair`) —> `bool`
- `graph` **`contains_edge`** `unknown` —> `bool`
- **`contains_edge`** (`graph` , `unknown`) —> `bool`

4.5.41.2 Result:

returns true if the graph(left-hand operand) contains the given edge (righ-hand operand), false otherwise

4.5.41.3 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

```
bool var0 <- graphEpidemio contains_edge (node(0)::node(3)); // var0 equals true
```

4.5.41.4 Examples:

```
graph graphFromMap <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
bool var2 <- graphFromMap contains_edge link({1,5},{12,45}); // var2 equals true
```

4.5.41.5 See also:

contains_vertex,

4.5.42 contains_vertex

4.5.42.1 Possible use:

- graph **contains_vertex** unknown —> bool
- **contains_vertex** (graph , unknown) —> bool

4.5.42.2 Result:

returns true if the graph(left-hand operand) contains the given vertex (righ-hand operand), false otherwise

4.5.42.3 Special cases:

- if the left-hand operand is nil, returns false

4.5.42.4 Examples:

```
graph graphFromMap<-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
bool var1 <- graphFromMap contains_vertex {1,5}; // var1 equals true
```

4.5.42.5 See also:

contains_edge,

4.5.43 conversation

4.5.43.1 Possible use:

- **conversation** (unknown) —> conversation

4.5.44 convex_hull

4.5.44.1 Possible use:

- **convex_hull** (geometry) —> geometry

4.5.44.2 Result:

A geometry corresponding to the convex hull of the operand.

4.5.44.3 Examples:

```
geometry var0 <- convex_hull(self); // var0 equals the convex hull of the geometry of the agent applying the
```

4.5.45 copy

4.5.45.1 Possible use:

- `copy` (`unknown`) —> `unknown`

4.5.45.2 Result:

returns a copy of the operand.

4.5.46 copy_between

4.5.46.1 Possible use:

- `copy_between` (`list`, `int`, `int`) —> `list`
- `copy_between` (`string`, `int`, `int`) —> `string`

4.5.46.2 Result:

Returns a copy of the first operand between the indexes determined by the second (inclusive) and third operands (exclusive)

4.5.46.3 Special cases:

- If the first operand is empty, returns an empty object of the same type
- If the second operand is greater than or equal to the third operand, return an empty object of the same type
- If the first operand is nil, raises an error

4.5.46.4 Examples:

```
list var0 <-  copy_between ([4, 1, 6, 9 ,7], 1, 3); // var0 equals [1, 6]
string var1 <- copy_between("abcabcabc", 2,6); // var1 equals "cabc"
```

4.5.47 corR

4.5.47.1 Possible use:

- `container` `corR` `container` —> `unknown`
- `corR` (`container` , `container`) —> `unknown`

4.5.47.2 Result:

returns the Pearson correlation coefficient of two given vectors (right-hand operands) in given variable (left-hand operand).

4.5.47.3 Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

4.5.47.4 Examples:

```
list X <- [1, 2, 3]; list Y <- [1, 2, 4];
unknown var2 <- corR(X, Y); // var2 equals 0.981980506061966
```

4.5.48 `correlation`

4.5.48.1 Possible use:

- `container` **`correlation`** `container` —> `float`
- **`correlation`** (`container` , `container`) —> `float`

4.5.48.2 Result:

Returns the correlation of two data sequences

4.5.49 `cos`

4.5.49.1 Possible use:

- **`cos`** (`float`) —> `float`
- **`cos`** (`int`) —> `float`

4.5.49.2 Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

4.5.49.3 Special cases:

- Operand values out of the range [0-359] are normalized.

4.5.49.4 Examples:

```
float var0 <- cos (0); // var0 equals 1.0
float var1 <- cos(360); // var1 equals 1.0
float var2 <- cos(-720); // var2 equals 1.0
```

4.5.49.5 See also:

sin, tan,

4.5.50 `cos_rad`

4.5.50.1 Possible use:

- `cos_rad` (`float`) —> `float`

4.5.50.2 Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in radians).

4.5.50.3 Special cases:

- Operand values out of the range [0-359] are normalized.

4.5.50.4 See also:

sin, tan,

4.5.51 `count`

4.5.51.1 Possible use:

- `container count any expression` —> `int`
- `count` (`container` , `any expression`) —> `int`

4.5.51.2 Result:

returns an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.

4.5.51.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

4.5.51.4 Special cases:

- if the left-hand operand is nil, count throws an error

4.5.51.5 Examples:

```
int var0 <- [1,2,3,4,5,6,7,8] count (each > 3); // var0 equals 5// Number of nodes of graph g2 without any out
int var3 <- g2 count (length(g2 out_edges_of each) = 0  ) ; // var3 equals the total number of out edges// Numb
int var6 <- [1::2, 3::4, 5::6] count (each > 4); // var6 equals 1
```

4.5.51.6 See also:

group_by,

4.5.52 covariance

4.5.52.1 Possible use:

- container **covariance** container —> float
- **covariance** (container , container) —> float

4.5.52.2 Result:

Returns the covariance of two data sequences

4.5.53 covers

4.5.53.1 Possible use:

- geometry **covers** geometry —> bool
- **covers** (geometry , geometry) —> bool

4.5.53.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) covers the right-geometry (or agent/point).

4.5.53.3 Special cases:

- if one of the operand is null, returns false.

4.5.53.4 Examples:

```
bool var0 <- square(5) covers square(2); // var0 equals true
```

4.5.53.5 See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

4.5.54 `create_map`

4.5.54.1 Possible use:

- `list` **`create_map`** `list` —> `map`
- **`create_map`** (`list` , `list`) —> `map`

4.5.54.2 Result:

returns a new map using the left operand as keys for the right operand

4.5.54.3 Special cases:

- if the left operand contains duplicates, create_map throws an error.
- if both operands have different lengths, choose the minimum length between the two operandsfor the size of the map

4.5.54.4 Examples:

```
map<int,string> var0 <- create_map([0,1,2],['a','b','c']); // var0 equals [0::'a',1::'b',2::'c']
map<int,float> var1 <- create_map([0,1],[0.1,0.2,0.3]); // var1 equals [0::0.1,1::0.2]
map<string,float> var2 <- create_map(['a','b','c','d'],[1.0,2.0,3.0]); // var2 equals ['a'::1.0,'b'::2.0,'
```

4.5.55 `cross`

4.5.55.1 Possible use:

- **`cross`** (`float`) —> `geometry`
- `float` **`cross`** `float` —> `geometry`
- **`cross`** (`float` , `float`) —> `geometry`

4.5.55.2 Result:

A cross, which radius is equal to the first operand and the width of the lines for the second A cross, which radius is equal to the first operand

4.5.55.3 Examples:

```
geometry var0 <- cross(10,2); // var0 equals a geometry as a cross of radius 10, and with a width of 2 for the
geometry var1 <- cross(10); // var1 equals a geometry as a cross of radius 10
```

4.5.55.4 See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

4.5.56 crosses

4.5.56.1 Possible use:

- `geometry` **`crosses`** `geometry` —> `bool`
- **`crosses`** (`geometry` , `geometry`) —> `bool`

4.5.56.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) crosses the right-geometry (or agent/point).

4.5.56.3 Special cases:

- if one of the operand is null, returns false.
- if one operand is a point, returns false.

4.5.56.4 Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) crosses polyline([{10,20},{20,10}]); // var0 equals true
bool var1 <- polyline([{10,10},{20,20}]) crosses {15,15}; // var1 equals true
bool var2 <- polyline([{0,0},{25,25}]) crosses polygon([{10,10},{10,20},{20,20},{20,10}]); // var2 equals
```

4.5.56.5 See also:

disjoint_from, intersects, overlaps, partially_overlaps, touches,

---

4.5.57 crs

4.5.57.1 Possible use:

- **`crs`** (`file`) —> `string`

4.5.57.2 Result:

the Coordinate Reference System (CRS) of the GIS file

4.5.57.3 Examples:

```
string var0 <- crs(my_shapefile); // var0 equals the crs of the shapefile
```

4.5.58 CRS_transform

4.5.58.1 Possible use:

- **CRS_transform** (geometry) —> geometry
- geometry **CRS_transform** string —> geometry
- **CRS_transform** (geometry , string) —> geometry

4.5.58.2 Special cases:

- returns the geometry corresponding to the transformation of the given geometry by the current CRS (Coordinate Reference System), the one corresponding to the world's agent one

```
geometry var0 <- CRS_transform(shape); // var0 equals a geometry corresponding to the agent geometry transfo
```

- returns the geometry corresponding to the transformation of the given geometry by the left operand CRS (Coordinate Reference System)

```
geometry var1 <- shape CRS_transform("EPSG:4326"); // var1 equals a geometry corresponding to the agent geom
```

4.5.59 csv_file

4.5.59.1 Possible use:

- **csv_file** (string) —> file

4.5.59.2 Result:

Constructs a file of type csv. Allowed extensions are limited to csv, tsv

---

4.5.60 cube

4.5.60.1 Possible use:

- **cube** (float) —> geometry

4.5.60.2 Result:

A cube geometry which side size is equal to the operand.

4.5.60.3 Comment:

the center of the cube is by default the location of the current agent in which has been called this operator.

4.5.60.4 Special cases:

- returns nil if the operand is nil.

4.5.60.5 Examples:

```
geometry var0 <- cube(10); // var0 equals a geometry as a square of side size 10.
```

4.5.60.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

4.5.61 curve

4.5.61.1 Possible use:

- `curve` (`point`, `point`, `float`) —> `geometry`
- `curve` (`point`, `point`, `point`) —> `geometry`
- `curve` (`point`, `point`, `point`, `int`) —> `geometry`
- `curve` (`point`, `point`, `float`, `bool`) —> `geometry`
- `curve` (`point`, `point`, `float`, `float`) —> `geometry`
- `curve` (`point`, `point`, `point`, `point`) —> `geometry`
- `curve` (`point`, `point`, `float`, `int`, `float`) —> `geometry`
- `curve` (`point`, `point`, `point`, `point`, `int`) —> `geometry`
- `curve` (`point`, `point`, `float`, `bool`, `int`) —> `geometry`
- `curve` (`point`, `point`, `float`, `bool`, `int`, `float`) —> `geometry`
- `curve` (`point`, `point`, `float`, `int`, `float`, `float`) —> `geometry`

4.5.61.2 Result:

A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points, considering the given rotation angle (90 = along the z axis). A cubic Bezier curve geometry built from the four given points composed of a given number of points. A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of 10 points. A quadratic Bezier curve geometry built from the three given points composed of a given numnber of points. A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of 10 points - the last boolean is used to specified if it is the right side. A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius considering the given rotation angle (90 = along the z axis). A quadratic Bezier curve geometry built from

the three given points composed of 10 points. A cubic Bezier curve geometry built from the four given points composed of 10 points. A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points - the boolean is used to specified if it is the right side and the last value to indicate where is the inflection point (between 0.0 and 1.0 - default 0.5). A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points, considering the given inflection point (between 0.0 and 1.0 - default 0.5), and the given rotation angle (90 = along the z axis). A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points - the boolean is used to specified if it is the right side.

4.5.61.3 Special cases:

- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil

4.5.61.4 Examples:

```
geometry var0 <- curve({0,0},{10,10}, 0.5, 100, 90); // var0 equals a cubic Bezier curve geometry composed of
geometry var1 <- curve({0,0}, {0,10}, {10,10}); // var1 equals a cubic Bezier curve geometry composed of 10 p
geometry var2 <- curve({0,0},{10,10}, 0.5); // var2 equals a cubic Bezier curve geometry composed of 10 point
geometry var3 <- curve({0,0}, {0,10}, {10,10}, 20); // var3 equals a quadratic Bezier curve geometry composed
geometry var4 <- curve({0,0},{10,10}, 0.5, false); // var4 equals a cubic Bezier curve geometry composed of 1
geometry var5 <- curve({0,0},{10,10}, 0.5, 90); // var5 equals a cubic Bezier curve geometry composed of 100
geometry var6 <- curve({0,0}, {0,10}, {10,10}); // var6 equals a quadratic Bezier curve geometry composed of
geometry var7 <- curve({0,0}, {0,10}, {10,10}); // var7 equals a cubic Bezier curve geometry composed of 10 p
geometry var8 <- curve({0,0},{10,10}, 0.5, false, 100, 0.8); // var8 equals a cubic Bezier curve geometry com
geometry var9 <- curve({0,0},{10,10}, 0.5, 100, 0.8, 90); // var9 equals a cubic Bezier curve geometry compos
geometry var10 <- curve({0,0},{10,10}, 0.5, false, 100); // var10 equals a cubic Bezier curve geometry compos
```

4.5.61.5 See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle, line,

---

4.5.62 `cylinder`

4.5.62.1 Possible use:

- `float cylinder float` —> `geometry`
- `cylinder` (`float` , `float`) —> `geometry`

4.5.62.2 Result:

A cylinder geometry which radius is equal to the operand.

4.5.62.3 Comment:

the center of the cylinder is by default the location of the current agent in which has been called this operator.

4.5.62.4 Special cases:

- returns a point if the operand is lower or equal to 0.

4.5.62.5 Examples:

```
geometry var0 <- cylinder(10,10); // var0 equals a geometry as a circle of radius 10.
```

4.5.62.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

Chapter 5

Operators (D to H)

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

5.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

5.2

5.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

5.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

5.5 Operators

5.5.1 `date`

5.5.1.1 Possible use:

- `string date string` —> `date`
- `date` (`string` , `string`) —> `date`
- `date` (`string`, `string`, `string`) —> `date`

5.5.1.2 Result:

converts a string to a date following a custom pattern and a specific locale (e.g. 'fr', 'en'...). The pattern can use "%Y %M %N %D %E %h %m %s %z" for parsing years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will parse the date using one of the ISO date & time formats (similar to date('...') in that case). The pattern can also follow the pattern definition found here, which gives much more control over what will be parsed: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns. Different patterns are available by default as constant: #iso_local, #iso_simple, #iso_offset, #iso_zoned and #custom, which can be changed in the preferences converts a string to a date following a custom pattern. The pattern can use "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will parse the date using one of the ISO date & time formats (similar to date('...') in that case). The pattern can also follow the pattern definition found here, which gives much more control over what will be parsed: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns. Different patterns are available by default as constant: #iso_local, #iso_simple, #iso_offset, #iso_zoned and #custom, which can be changed in the preferences

5.5.1.3 Examples:

```
date d <- date("1999-january-30", 'yyyy-MMMM-dd', 'en'); date den <- date("1999-12-30", 'yyyy-MM-dd');
```

5.5.2 `dbscan`

5.5.2.1 Possible use:

- `dbscan` (`list`, `float`, `int`) —> `list<list>`

5.5.2.2 Result:

returns the list of clusters (list of instance indices) computed with the dbscan (density-based spatial clustering of applications with noise) algorithm from the first operand data according to the maximum radius of the neighborhood to be considered (eps) and the minimum number of points needed for a cluster (minPts). Usage: dbscan(data,eps,minPoints)

5.5.2.3 Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

5.5.2.4 Examples:

```
list<list> var0 <- dbscan ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],10,2); // var0 equals []
```

5.5.3 dead

5.5.3.1 Possible use:

- `dead` (`agent`) —> `bool`

5.5.3.2 Result:

true if the agent is dead (or null), false otherwise.

5.5.3.3 Examples:

```
bool var0 <- dead(agent_A); // var0 equals true or false
```

5.5.4 degree_of

5.5.4.1 Possible use:

- `graph` `degree_of` `unknown` —> `int`
- `degree_of` (`graph` , `unknown`) —> `int`

5.5.4.2 Result:

returns the degree (in+out) of a vertex (right-hand operand) in the graph given as left-hand operand.

5.5.4.3 Examples:

```
int var1 <- graphFromMap degree_of (node(3)); // var1 equals 3
```

5.5.4.4 See also:

in_degree_of, out_degree_of,

5.5.5 dem

5.5.5.1 Possible use:

- **dem** (file) —> geometry
- file **dem** file —> geometry
- **dem** (file , file) —> geometry
- file **dem** float —> geometry
- **dem** (file , float) —> geometry
- **dem** (file, file, float) —> geometry

5.5.5.2 Result:

A polygon that is equivalent to the surface of the texture

5.5.5.3 Examples:

```
geometry var0 <- dem(dem,texture); // var0 equals a geometry as a rectangle of weight and height equal to the
geometry var1 <- dem(dem); // var1 equals returns a geometry as a rectangle of width and height equal to the t
geometry var2 <- dem(dem,texture,z_factor); // var2 equals a geometry as a rectangle of width and height equa
geometry var3 <- dem(dem,z_factor); // var3 equals a geometry as a rectangle of weight and height equal to th
```

5.5.6 det

Same signification as determinant

5.5.7 determinant

5.5.7.1 Possible use:

- **determinant** (matrix) —> float

5.5.7.2 Result:

The determinant of the given matrix

5.5.7.3 Examples:

```
float var0 <- determinant(matrix([[1,2],[3,4]])); // var0 equals -2
```

5.5.8 `diff`

5.5.8.1 Possible use:

- `float diff float` —> `float`
- `diff` (`float` , `float`) —> `float`

5.5.8.2 Result:

A placeholder function for expressing equations

5.5.9 `diff2`

5.5.9.1 Possible use:

- `float diff2 float` —> `float`
- `diff2` (`float` , `float`) —> `float`

5.5.9.2 Result:

A placeholder function for expressing equations

5.5.10 `directed`

5.5.10.1 Possible use:

- `directed` (`graph`) —> `graph`

5.5.10.2 Result:

the operand graph becomes a directed graph.

5.5.10.3 Comment:

the operator alters the operand graph, it does not create a new one.

5.5.10.4 See also:

undirected,

5.5.11 direction_between

5.5.11.1 Possible use:

- topology **direction_between** container<geometry> —> float
- **direction_between** (topology , container<geometry>) —> float

5.5.11.2 Result:

A direction (in degree) between a list of two geometries (geometries, agents, points) considering a topology.

5.5.11.3 Examples:

```
float var0 <- my_topology direction_between [ag1, ag2]; // var0 equals the direction between ag1 and ag2 cons
```

5.5.11.4 See also:

towards, direction_to, distance_to, distance_between, path_between, path_to,

---

5.5.12 direction_to

Same signification as towards

5.5.13 disjoint_from

5.5.13.1 Possible use:

- geometry **disjoint_from** geometry —> bool
- **disjoint_from** (geometry , geometry) —> bool

5.5.13.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) is disjoints from the right-geometry (or agent/point).

5.5.13.3 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.

5.5.13.4 Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) disjoint_from polyline([{15,15},{25,25}]); // var0 equals false
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from polygon([{15,15},{15,25},{25,25},{2
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from {15,15}; // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from {25,25}; // var3 equals true
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from polygon([{35,35},{35,45},{45,45},{4
```

5.5.13.5 See also:

intersects, crosses, overlaps, partially_overlaps, touches,

5.5.14 distance_between

5.5.14.1 Possible use:

- topology **distance_between** container<geometry> —> float
- **distance_between** (topology , container<geometry>) —> float

5.5.14.2 Result:

A distance between a list of geometries (geometries, agents, points) considering a topology.

5.5.14.3 Examples:

```
float var0 <- my_topology distance_between [ag1, ag2, ag3]; // var0 equals the distance between ag1, ag2 and
```

5.5.14.4 See also:

towards, direction_to, distance_to, direction_between, path_between, path_to,

---

5.5.15 distance_to

5.5.15.1 Possible use:

- geometry **distance_to** geometry —> float
- **distance_to** (geometry , geometry) —> float
- point **distance_to** point —> float
- **distance_to** (point , point) —> float

5.5.15.2 Result:

A distance between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

5.5.15.3 Examples:

```
float var0 <- ag1 distance_to ag2; // var0 equals the distance between ag1 and ag2 considering the topology o
```

5.5.15.4 See also:

towards, direction_to, distance_between, direction_between, path_between, path_to,

---

5.5.16 distinct

5.5.16.1 Possible use:

- **distinct** (container) —> list

5.5.16.2 Result:

produces a set from the elements of the operand (i.e. a list without duplicated elements)

5.5.16.3 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 matrix, remove_duplicates returns a matrix without duplicated row
- if the operand is a map, remove_duplicates returns the set of values without duplicate

```
list var1 <- remove_duplicates([1::3,2::4,3::3,5::7]); // var1 equals [3,4,7]
```

5.5.16.4 Examples:

```
list var0 <- remove_duplicates([3,2,5,1,2,3,5,5,5]); // var0 equals [3,2,5,1]
```

5.5.17 distribution_of

5.5.17.1 Possible use:

- **distribution_of** (container) —> map
- container **distribution_of** int —> map
- **distribution_of** (container , int) —> map
- **distribution_of** (container, int, float, float) —> map

5.5.17.2 Result:

Discretize a list of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nbbins) or (list,nbbins,valmin,valmax)

5.5.17.3 Examples:

```
map var0 <- distribution_of([1,1,2,12.5]); // var0 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'::[
map var1 <- distribution_of([1,1,2,12.5]); // var1 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'::[
map var2 <- distribution_of([1,1,2,12.5],10); // var2 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'
```

5.5.17.4 See also:

as_map,

5.5.18 distribution2d_of

5.5.18.1 Possible use:

- `container` **`distribution2d_of`** `container` —> `map`
- **`distribution2d_of`** (`container` , `container`) —> `map`
- **`distribution2d_of`** (`container`, `container`, `int`, `int`) —> `map`
- **`distribution2d_of`** (`container`, `container`, `int`, `float`, `float`, `int`, `float`, `float`) —> `map`

5.5.18.2 Result:

Discretize two lists of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nbbins) or (list,nbbins,valmin,valmax)

5.5.18.3 Examples:

```
map var0 <- distribution_of([1,1,2,12.5],10); // var0 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'
map var1 <- distribution2d_of([1,1,2,12.5]); // var1 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend':
map var2 <- distribution_of([1,1,2,12.5],10); // var2 equals map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'
```

5.5.18.4 See also:

as_map,

5.5.19 `div`

5.5.19.1 Possible use:

- `int div float` —> `int`
- `div (int , float)` —> `int`
- `float div float` —> `int`
- `div (float , float)` —> `int`
- `float div int` —> `int`
- `div (float , int)` —> `int`
- `int div int` —> `int`
- `div (int , int)` —> `int`

5.5.19.2 Result:

Returns the truncation of the division of the left-hand operand by the right-hand operand.

5.5.19.3 Special cases:

- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.

5.5.19.4 Examples:

```
int var0 <- 40 div 4.1; // var0 equals 9
int var1 <- 40.1 div 4.5; // var1 equals 8
int var2 <- 40.5 div 3; // var2 equals 13
int var3 <- 40 div 3; // var3 equals 13
```

5.5.19.5 See also:

mod,

5.5.20 `dnorm`

Same signification as normal_density

5.5.21 `dtw`

5.5.21.1 Possible use:

- `list dtw list` —> `float`

- **dtw** (list , list) —> float
- **dtw** (list, list, int) —> float

5.5.21.2 Result:

returns the dynamic time warping between the two series of value with Sakoe-Chiba band (radius: the window width of Sakoe-Chiba band) returns the dynamic time warping between the two series of value

5.5.21.3 Examples:

```
float var0 <- dtw([10.0,5.0,1.0, 3.0],[1.0,10.0,5.0,1.0], 2); // var0 equals 2.0
float var1 <- dtw([10.0,5.0,1.0, 3.0],[1.0,10.0,5.0,1.0]); // var1 equals 2
```

5.5.22 durbin_watson

5.5.22.1 Possible use:

- **durbin_watson** (container) —> float

5.5.22.2 Result:

Durbin-Watson computation

5.5.23 dxf_file

5.5.23.1 Possible use:

- **dxf_file** (string) —> file

5.5.23.2 Result:

Constructs a file of type dxf. Allowed extensions are limited to dxf

---

5.5.24 edge

5.5.24.1 Possible use:

- **edge** (unknown) —> unknown
- **edge** (pair) —> unknown
- pair **edge** float —> unknown
- **edge** (pair , float) —> unknown
- unknown **edge** unknown —> unknown
- **edge** (unknown , unknown) —> unknown

- `unknown edge float —> unknown`
- `edge (unknown , float) —> unknown`
- `edge (unknown, unknown, unknown) —> unknown`
- `edge (pair, unknown, float) —> unknown`
- `edge (unknown, unknown, float) —> unknown`
- `edge (unknown, unknown, unknown, float) —> unknown`

5.5.25 edge_between

5.5.25.1 Possible use:

- `graph edge_between pair —> unknown`
- `edge_between (graph , pair) —> unknown`

5.5.25.2 Result:

returns the edge linking two nodes

5.5.25.3 Examples:

```
unknown var0 <- graphFromMap edge_between node1::node2; // var0 equals edge1
```

5.5.25.4 See also:

out_edges_of, in_edges_of,

5.5.26 edge_betweenness

5.5.26.1 Possible use:

- `edge_betweenness (graph) —> map`

5.5.26.2 Result:

returns a map containing for each edge (key), its betweenness centrality (value): number of shortest paths passing through each edge

5.5.26.3 Examples:

```
graph graphEpidemio <- graph([]);
map var1 <- edge_betweenness(graphEpidemio); // var1 equals the edge betweenness index of the graph
```

5.5.27 edges

5.5.27.1 Possible use:

- `edges` (`container`) —> `container`

5.5.28 eigenvalues

5.5.28.1 Possible use:

- `eigenvalues` (`matrix`) —> `list<float>`

5.5.28.2 Result:

The eigen values (matrix) of the given matrix

5.5.28.3 Examples:

```
list<float> var0 <- eigenvalues(matrix([[5,-3],[6,-4]])); // var0 equals [2.0000000000000004,-0.999999999
```

5.5.29 electre_DM

5.5.29.1 Possible use:

- `electre_DM` (`msi.gama.util.IList<java.util.List>`, `msi.gama.util.IList<java.util.Map<java.lang.String,j` `float`) —> `int`

5.5.29.2 Result:

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 characterizes the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterizes 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 [http://www.springerlink.com/content/g367r44322876223/ 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)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: 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; the last operand is the fuzzy cut.

5.5.29.3 Special cases:

- returns -1 is the list of candidates is nil or empty

5.5.29.4 Examples:

```
int var0 <- electre_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"utility", "weight" :: 2.0,"p"::0.5, "c
```

5.5.29.5 See also:

weighted_means_DM, promethee_DM, evidence_theory_DM,

5.5.30 ellipse

5.5.30.1 Possible use:

- `float ellipse float` —> `geometry`
- `ellipse` (`float` , `float`) —> `geometry`

5.5.30.2 Result:

An ellipse geometry which x-radius is equal to the first operand and y-radius is equal to the second operand

5.5.30.3 Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

5.5.30.4 Special cases:

- returns a point if both operands are lower or equal to 0, a line if only one is.

5.5.30.5 Examples:

```
geometry var0 <- ellipse(10, 10); // var0 equals a geometry as an ellipse of width 10 and height 10.
```

5.5.30.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, circle, squircle, triangle,

---

5.5.31 emotion

5.5.31.1 Possible use:

- `emotion` (`any`) —> `emotion`

5.5.31.2 Result:

Casts the operand into the type emotion

5.5.32 empty

5.5.32.1 Possible use:

- `empty` (`string`) —> `bool`
- `empty` (`container<KeyType,ValueType>`) —> `bool`

5.5.32.2 Result:

true if the operand is empty, false otherwise.

5.5.32.3 Comment:

the empty operator behavior depends on the nature of the operand

5.5.32.4 Special cases:

- 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

```
bool var0 <- empty ('abced'); // var0 equals false
```

- if it is a list, empty returns true if there is no element in the list, and false otherwise

```
bool var1 <- empty([]); // var1 equals true
```

5.5.33 enlarged_by

Same signification as +

5.5.34 envelope

5.5.34.1 Possible use:

- `envelope` (`unknown`) —> `geometry`

5.5.34.2 Result:

A 3D geometry that represents the box that surrounds the geometries or the surface described by the arguments. More general than geometry(arguments).envelope, as it allows to pass int, double, point, image files, shape files, asc files, or any list combining these arguments, in which case the envelope will be correctly expanded. If an envelope cannot be determined from the arguments, a default one of dimensions (0,100, 0, 100, 0, 100) is returned

---

5.5.35 eval_gaml

5.5.35.1 Possible use:

- `eval_gaml` (`string`) —> `unknown`

5.5.35.2 Result:

evaluates the given GAML string.

5.5.35.3 Examples:

```
unknown var0 <- eval_gaml("2+3"); // var0 equals 5
```

5.5.36 eval_when

5.5.36.1 Possible use:

- `eval_when` (`BDIPlan`) —> `bool`

5.5.36.2 Result:

evaluate the facet when of a given plan

5.5.36.3 Examples:

```
eval_when(plan1)
```

5.5.37 evaluate_sub_model

5.5.37.1 Possible use:

- msi.gama.kernel.experiment.IExperimentAgent **evaluate_sub_model** string —> unknown
- **evaluate_sub_model** (msi.gama.kernel.experiment.IExperimentAgent , string) —> unknown

5.5.37.2 Result:

Load a submodel

5.5.37.3 Comment:

loaded submodel

5.5.38 even

5.5.38.1 Possible use:

- **even** (int) —> bool

5.5.38.2 Result:

Returns true if the operand is even and false if it is odd.

5.5.38.3 Special cases:

- if the operand is equal to 0, it returns true.
- if the operand is a float, it is truncated before

5.5.38.4 Examples:

```
bool var0 <- even (3); // var0 equals false
bool var1 <- even(-12); // var1 equals true
```

5.5.39 every

5.5.39.1 Possible use:

- **every** (int) —> bool
- **every** (any expression) —> bool
- list **every** int —> list
- **every** (list , int) —> list
- msi.gama.util.GamaDateInterval **every** any expression —> msi.gama.util.IList<msi.gama.util.GamaDate>
- **every** (msi.gama.util.GamaDateInterval , any expression) —> msi.gama.util.IList<msi.gama.util.GamaDate

5.5.39.2 Result:

true every operand * cycle, false otherwise Retrieves elements from the first argument every **step** (second argument) elements. Raises an error if the step is negative or equal to zero expects a frequency (expressed in seconds of simulated time) as argument. Will return true every time the current_date matches with this frequency applies a step to an interval of dates defined by 'date1 to date2'

5.5.39.3 Comment:

the value of the every operator depends on the cycle. It can be used to do something every x cycle.Used to do something at regular intervals of time. Can be used in conjunction with 'since', 'after', 'before', 'until' or 'between', so that this computation only takes place in the temporal segment defined by these operators. In all cases, the starting_date of the model is used as a reference starting point

5.5.39.4 Examples:

```
if every(2#cycle) {write "the cycle number is even";}      else {write "the cycle number is odd";} reflex whe
```

5.5.39.5 See also:

since, after, to,

5.5.40 every_cycle

Same signification as every

5.5.41 evidence_theory_DM

5.5.41.1 Possible use:

- msi.gama.util.IList<java.util.List> **evidence_theory_DM** msi.gama.util.IList<java.util.Map<java.lang.S —> int
- **evidence_theory_DM** (msi.gama.util.IList<java.util.List> , msi.gama.util.IList<java.util.Map<java.lang —> int

- **evidence_theory_DM** (msi.gama.util.IList<java.util.List>, msi.gama.util.IList<java.util.Map<java.lang bool) —> int

5.5.41.2 Result:

The index of the best candidate according to a method based on the Evidence theory. This theory, which was proposed by Shafer ([http://www.glennshafer.com/books/amte.html Shafer G (1976) A mathematical theory of evidence, Princeton University Press]), is based on the work of Dempster ([http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177698950 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. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: 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.; the last operand allows to use a simple version of this multi-criteria decision making method (simple if true)

5.5.41.3 Special cases:

- if the operator is used with only 2 operands (the candidates and the criteria), the last parameter (use simple method) is set to true
- returns -1 is the list of candidates is nil or empty

5.5.41.4 Examples:

```
int var0 <- evidence_theory_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"utility", "s1" :: 0.0,"s2"::1
```

5.5.41.5 See also:

weighted_means_DM, electre_DM,

5.5.42 exp

5.5.42.1 Possible use:

- **exp** (float) —> float
- **exp** (int) —> float

5.5.42.2 Result:

Returns Euler's number e raised to the power of the operand.

5.5.42.3 Special cases:

- the operand is casted to a float before being evaluated.
- the operand is casted to a float before being evaluated.

5.5.42.4 Examples:

```
float var0 <- exp (0); // var0 equals 1.0
```

5.5.42.5 See also:

ln,

5.5.43 `fact`

5.5.43.1 Possible use:

- `fact` (`int`) —> `float`

5.5.43.2 Result:

Returns the factorial of the operand.

5.5.43.3 Special cases:

- if the operand is less than 0, fact returns 0.

5.5.43.4 Examples:

```
float var0 <- fact(4); // var0 equals 24
```

5.5.44 `farthest_point_to`

5.5.44.1 Possible use:

- `geometry` **`farthest_point_to`** `point` —> `point`
- **`farthest_point_to`** (`geometry` , `point`) —> `point`

5.5.44.2 Result:

the farthest point of the left-operand to the left-point.

5.5.44.3 Examples:

```
point var0 <- geom farthest_point_to(pt); // var0 equals the farthest point of geom to pt
```

5.5.44.4 See also:

any_location_in, any_point_in, closest_points_with, points_at,

---

5.5.45 farthest_to

5.5.45.1 Possible use:

- container<agent> **farthest_to** geometry —> geometry
- **farthest_to** (container<agent> , geometry) —> geometry

5.5.45.2 Result:

An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the farthest to the operand (casted as a geometry).

5.5.45.3 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.

5.5.45.4 Examples:

```
geometry var0 <- [ag1, ag2, ag3] closest_to(self); // var0 equals return the farthest agent among ag1, ag2 an
```

5.5.45.5 See also:

neighbors_at, neighbors_of, inside, overlapping, agents_overlapping, agents_inside, agent_closest_to, closest_to, agent_farthest_to,

---

5.5.46 file

5.5.46.1 Possible use:

- **file** (string) —> file
- string **file** container —> file
- **file** (string , container) —> file

5.5.46.2 Result:

Creates a file in read/write mode, setting its contents to the container passed in parameter 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.

5.5.46.3 Comment:

The type of container to pass will depend on the type of file (see the management of files in the documentation). Can be used to copy files since files are considered as containers. For example: save file('image_copy.png', file('image.png')); will copy image.png to image_copy.pngThe file should have a supported extension, see file type definition for supported file extensions.

5.5.46.4 Special cases:

- If the specified string does not refer to an existing file, an exception is risen when the variable is used.

5.5.46.5 Examples:

```
let fileT type: file value: file("../includes/Stupid_Cell.Data");       // fileT represents the file "../i
```

5.5.46.6 See also:

folder, new_folder,

5.5.47 `file`

5.5.47.1 Possible use:

- `file` (`any`) —> `file`

5.5.47.2 Result:

Casts the operand into the type file

5.5.48 `file_exists`

5.5.48.1 Possible use:

- `file_exists` (`string`) —> `bool`

5.5.48.2 Result:

Test whether the parameter is the path to an existing file.

5.5.49 first

5.5.49.1 Possible use:

- first (container<KeyType,ValueType>) —> ValueType
- first (string) —> string
- int first container —> list
- first (int , container) —> list

5.5.49.2 Result:

the first value of the operand

5.5.49.3 Comment:

the first operator behavior depends on the nature of the operand

5.5.49.4 Special cases:

- if it is a map, first returns the first value of the first pair (in insertion order)
- 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 edge (in creation order)
- 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 nil if the matrix is empty
- if it is a list, first returns the first element of the list, or nil if the list is empty

```
int var0 <- first ([1, 2, 3]); // var0 equals 1
```

- if it is a string, first returns a string composed of its first character

```
string var1 <- first ('abce'); // var1 equals 'a'
```

5.5.49.5 See also:

last,

5.5.50 first_of

Same signification as first

5.5.51 first_with

5.5.51.1 Possible use:

- container **first_with** any expression —> unknown
- **first_with** (container , any expression) —> unknown

5.5.51.2 Result:

the first element of the left-hand operand that makes the right-hand operand evaluate to true.

5.5.51.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

5.5.51.4 Special cases:

- if the left-hand operand is nil, first_with throws an error. If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] first_with (each >= 4); // var4 equals 4
unknown var5 <- [1::2, 3::4, 5::6].pairs first_with (each.value >= 4); // var5 equals (3::4)
```

5.5.51.5 Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] first_with (each > 3); // var0 equals 4
unknown var2 <- g2 first_with (length(g2 out_edges_of each) = 0); // var2 equals node9
unknown var3 <- (list(node) first_with (round(node(each).location.x) > 32); // var3 equals node2
```

5.5.51.6 See also:

group_by, last_with, where,

5.5.52 flip

5.5.52.1 Possible use:

- `flip` (`float`) —> `bool`

5.5.52.2 Result:

true or false given the probability represented by the operand

5.5.52.3 Special cases:

- flip 0 always returns false, flip 1 true

5.5.52.4 Examples:

```
bool var0 <- flip (0.66666); // var0 equals 2/3 chances to return true.
```

5.5.52.5 See also:

rnd,

5.5.53 float

5.5.53.1 Possible use:

- `float` (`any`) —> `float`

5.5.53.2 Result:

Casts the operand into the type float

5.5.54 floor

5.5.54.1 Possible use:

- `floor` (`float`) —> `float`

5.5.54.2 Result:

Maps the operand to the largest previous following integer, i.e. the largest integer not greater than x.

5.5.54.3 Examples:

```
float var0 <- floor(3); // var0 equals 3.0
float var1 <- floor(3.5); // var1 equals 3.0
float var2 <- floor(-4.7); // var2 equals -5.0
```

5.5.54.4 See also:

ceil, round,

5.5.55 `folder`

5.5.55.1 Possible use:

- `folder` (`string`) —> `file`

5.5.55.2 Result:

opens an existing repository

5.5.55.3 Special cases:

- If the specified string does not refer to an existing repository, an exception is risen.

5.5.55.4 Examples:

```
file dirT <- folder("../includes/");           // dirT represents the repository "../includes/"           /
```

5.5.55.5 See also:

file, new_folder,

5.5.56 `font`

5.5.56.1 Possible use:

- `font` (`string`, `int`, `int`) —> `font`

5.5.56.2 Result:

Creates a new font, by specifying its name (either a font face name like ‘Lucida Grande Bold’ or ‘Helvetica’, or a logical name like ‘Dialog’, ‘SansSerif’, ‘Serif’, etc.), a size in points and a style, either #bold, #italic or #plain or a combination (addition) of them.

5.5.56.3 Examples:

```
font var0 <- font ('Helvetica Neue',12, #bold + #italic); // var0 equals a bold and italic face of the Helveti
```

5.5.57 frequency_of

5.5.57.1 Possible use:

- container **frequency_of** any expression —> map
- **frequency_of** (container , any expression) —> map

5.5.57.2 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)

5.5.57.3 Examples:

```
map var0 <- [ag1, ag2, ag3, ag4] frequency_of each.size; // var0 equals the different sizes as keys and the nu
```

5.5.57.4 See also:

as_map,

5.5.58 from

Same signification as since

5.5.59 fuzzy_choquet_DM

5.5.59.1 Possible use:

- **fuzzy_choquet_DM** (msi.gama.util.IList<java.util.List>, list<string>, map) —> int

5.5.59.2 Result:

The index of the candidate that maximizes the Fuzzy Choquet Integral value. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion (list of string); the third operand the weights of each sub-set of criteria (map with list for key and float for value)

5.5.59.3 Special cases:

- returns -1 is the list of candidates is nil or empty

5.5.59.4 Examples:

```
int var0 <- fuzzy_choquet_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], ["utility", "price", "size"],[["utility"]:
```

5.5.59.5 See also:

promethee_DM, electre_DM, evidence_theory_DM,

5.5.60 fuzzy_kappa

5.5.60.1 Possible use:

- **fuzzy_kappa** (list<agent>, list, list, list<float>, list, matrix<float>, float) —> float
- **fuzzy_kappa** (list<agent>, list, list, list<float>, list, matrix<float>, float, list) —> float

5.5.60.2 Result:

fuzzy kappa indicator for 2 map comparisons: fuzzy_kappa(agents_list,list_vals1,list_vals2, output_similarity_per_agents,categories,fuzzy_categories_matrix, fuzzy_distance). Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21 fuzzy kappa indicator for 2 map comparisons: fuzzy_kappa(agents_list,list_vals1,list_vals2, output_similarity_per_agents,categories,fuzzy_categories_matrix, fuzzy_distance, weights). Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21

5.5.60.3 Examples:

```
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2], similarity_p
```

5.5.61 fuzzy_kappa_sim

5.5.61.1 Possible use:

- **fuzzy_kappa_sim** (list<agent>, list, list, list, list<float>, list, matrix<float>, float) —> float
- **fuzzy_kappa_sim** (list<agent>, list, list, list, list<float>, list, matrix<float>, float, list) —> float

5.5.61.2 Result:

fuzzy kappa simulation indicator for 2 map comparisons: fuzzy_kappa_sim(agents_list,list_vals1,list_vals2, output_similarity_per_agents,fuzzy_transitions_matrix, fuzzy_distance). Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800, fuzzy kappa simulation indicator for 2 map comparisons: fuzzy_kappa_sim(agents_list,list_vals1,list_vals2, output_similarity_per_agents,fuzzy_transitions_matrix, fuzzy_distance, weights). Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800,

5.5.61.3 Examples:

```
fuzzy_kappa_sim([ag1, ag2, ag3, ag4, ag5], [cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2], similar
```

5.5.62 gaml_file

5.5.62.1 Possible use:

- `gaml_file` (`string`) —> `file`

5.5.62.2 Result:

Constructs a file of type gaml. Allowed extensions are limited to gaml, experiment

---

5.5.63 gaml_type

5.5.63.1 Possible use:

- `gaml_type` (`any`) —> `gaml_type`

5.5.63.2 Result:

Casts the operand into the type gaml_type

5.5.64 gamma

5.5.64.1 Possible use:

- `gamma` (`float`) —> `float`

5.5.64.2 Result:

Returns the value of the Gamma function at x.

5.5.65 gamma_distribution

5.5.65.1 Possible use:

- `gamma_distribution` (`float`, `float`, `float`) —> `float`

5.5.65.2 Result:

Returns the integral from zero to x of the gamma probability density function.

5.5.65.3 Comment:

incomplete_gamma(a,x) is equal to pgamma(a,1,x).

5.5.66 gamma_distribution_complemented

5.5.66.1 Possible use:

- `gamma_distribution_complemented` (`float`, `float`, `float`) —> `float`

5.5.66.2 Result:

Returns the integral from x to infinity of the gamma probability density function.

5.5.67 gamma_index

5.5.67.1 Possible use:

- `gamma_index` (`graph`) —> `float`

5.5.67.2 Result:

returns the gamma index of the graph (A measure of connectivity that considers the relationship between the number of observed links and the number of possible links: gamma = e/(3 * (v - 2)) - for planar graph.

5.5.67.3 Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- gamma_index(graphEpidemio); // var1 equals the gamma index of the graph
```

5.5.67.4 See also:

alpha_index, beta_index, nb_cycles, connectivity_index,

5.5.68 gamma_rnd

5.5.68.1 Possible use:

- `float` **`gamma_rnd`** `float` —> `float`
- **`gamma_rnd`** (`float` , `float`) —> `float`

5.5.68.2 Result:

returns a random value from a gamma distribution with specified values of the shape and scale parameters

5.5.68.3 Examples:

```
gamma_rnd(10.0,5.0)
```

5.5.69 gauss

5.5.69.1 Possible use:

- **`gauss`** (`point`) —> `float`
- `float` **`gauss`** `float` —> `float`
- **`gauss`** (`float` , `float`) —> `float`

5.5.69.2 Result:

A value from a normally distributed random variable with expected value (mean) and variance (standard-Deviation). The probability density function of such a variable is a Gaussian. 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.

5.5.69.3 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
- when the operand is a point, it is read as {mean, standardDeviation}
- when standardDeviation value is 0.0, it always returns the mean value

5.5.69.4 Examples:

```
float var0 <- gauss({0,0.3}); // var0 equals 0.22354
float var1 <- gauss({0,0.3}); // var1 equals -0.1357
float var2 <- gauss(0,0.3); // var2 equals 0.22354
float var3 <- gauss(0,0.3); // var3 equals -0.1357
```

5.5.69.5 See also:

truncated_gauss, poisson, skew_gauss,

5.5.70 generate_barabasi_albert

5.5.70.1 Possible use:

- **generate_barabasi_albert** (container<agent>, species, int, bool) —> graph
- **generate_barabasi_albert** (species, species, int, int, bool) —> graph

5.5.70.2 Result:

returns a random scale-free network (following Barabasi-Albert (BA) model). returns a random scale-free network (following Barabasi-Albert (BA) model).

5.5.70.3 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: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:

5.5.70.4 Special cases:

- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “size”: the graph will contain (size + 1) nodes
- “m”: the number of edges added per novel node
- “synchronized”: is the graph and the species of vertices and edges synchronized?

- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain (size + 1) nodes
- "m": the number of edges added per novel node
- "synchronized": is the graph and the species of vertices and edges synchronized?

5.5.70.5 Examples:

```
graph<yourNodeSpecy,yourEdgeSpecy> graphEpidemio <- generate_barabasi_albert(     yourListOfNodes,     y
```

5.5.70.6 See also:

generate_watts_strogatz,

5.5.71 generate_complete_graph

5.5.71.1 Possible use:

- **generate_complete_graph** (container<agent>, species, bool) —> graph
- **generate_complete_graph** (species, species, int, bool) —> graph
- **generate_complete_graph** (container<agent>, species, float, bool) —> graph
- **generate_complete_graph** (species, species, int, float, bool) —> graph

5.5.71.2 Result:

returns a fully connected graph. returns a fully connected graph. returns a fully connected graph. returns a fully connected graph.

5.5.71.3 Comment:

Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:

5.5.71.4 Special cases:

- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain size nodes.
- "synchronized": is the graph and the species of vertices and edges synchronized?

- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain size nodes.
- "layoutRadius": nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "agents": list of existing node agents
- "edges_species": the species of edges
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "agents": list of existing node agents
- "edges_species": the species of edges
- "layoutRadius": nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- "synchronized": is the graph and the species of vertices and edges synchronized?

5.5.71.5 Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(        myVertexSpecy,       myEdgeSp
```

5.5.71.6 See also:

generate_barabasi_albert, generate_watts_strogatz,

5.5.72 generate_watts_strogatz

5.5.72.1 Possible use:

- **generate_watts_strogatz** (container<agent>, species, float, int, bool) —> graph
- **generate_watts_strogatz** (species, species, int, float, int, bool) —> graph

5.5.72.2 Result:

returns a random small-world network (following Watts-Strogatz model). returns a random small-world network (following Watts-Strogatz model).

5.5.72.3 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: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:

5.5.72.4 Special cases:

- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “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.
- “synchronized”: is the graph and the species of vertices and edges synchronized?
- “vertices_specy”: the species of vertices
- “edges_species”: the species of edges
- “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.
- “synchronized”: is the graph and the species of vertices and edges synchronized?

5.5.72.5 Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(        myListOfNodes,      myEdgeSp
```

5.5.72.6 See also:

generate_barabasi_albert,

5.5.73 geojson_file

5.5.73.1 Possible use:

- `geojson_file` (`string`) —> `file`

5.5.73.2 Result:

Constructs a file of type geojson. Allowed extensions are limited to json, geojson, geo.json

---

5.5.74 geometric_mean

5.5.74.1 Possible use:

- `geometric_mean` (`container`) —> `float`

5.5.74.2 Result:

the geometric mean of the elements of the operand. See Geometric_mean for more details.

5.5.74.3 Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

5.5.74.4 Examples:

```
float var0 <- geometric_mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 4.962326343467649
```

5.5.74.5 See also:

mean, median, harmonic_mean,

5.5.75 geometry

5.5.75.1 Possible use:

- `geometry` (`any`) —> `geometry`

5.5.75.2 Result:

Casts the operand into the type geometry

5.5.76 geometry_collection

5.5.76.1 Possible use:

- `geometry_collection` (`container<geometry>`) —> `geometry`

5.5.76.2 Result:

A geometry collection (multi-geometry) composed of the given list of geometries.

5.5.76.3 Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single geometry, returns a copy of the geometry.

5.5.76.4 Examples:

```
geometry var0 <- geometry_collection([{0,0}, {0,10}, {10,10}, {10,0}]); // var0 equals a geometry composed o
```

5.5.76.5 See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle, line,

---

5.5.77 get

Same signification as read

5.5.77.1 Possible use:

- `agent get string` —> `unknown`
- `get` (`agent` , `string`) —> `unknown`
- `geometry get string` —> `unknown`
- `get` (`geometry` , `string`) —> `unknown`

5.5.77.2 Result:

Reads an attribute of the specified agent (left operand). The attribute name is specified by the right operand. Reads an attribute of the specified geometry (left operand). The attribute name is specified by the right operand.

5.5.77.3 Special cases:

- Reading the attribute of another agent

```
string agent_name <- an_agent get('name');    // reads then 'name' attribute of an_agent then assigns the ret
```

- Reading the attribute of a geometry

```
string geom_area <- a_geometry get('area');    // reads then 'area' attribute of 'a_geometry' variable then a
```

5.5.78 get_about

5.5.78.1 Possible use:

- `get_about` (`emotion`) —> `predicate`

5.5.78.2 Result:

get the about value of the given emotion

5.5.78.3 Examples:

```
get_about(emotion)
```

5.5.79 get_agent

5.5.79.1 Possible use:

- `get_agent` (`msi.gaml.architecture.simplebdi.SocialLink`) —> `agent`

5.5.79.2 Result:

get the agent value of the given social link

5.5.79.3 Examples:

```
get_agent(social_link1)
```

5.5.80 get_agent_cause

5.5.80.1 Possible use:

- `get_agent_cause` (`predicate`) —> `agent`
- `get_agent_cause` (`emotion`) —> `agent`

5.5.80.2 Result:

get the agent cause value of the given emotion

5.5.80.3 Examples:

```
get_agent_cause(emotion)
```

5.5.81 get_belief_op

5.5.81.1 Possible use:

- agent **get_belief_op** predicate —> mental_state
- **get_belief_op** (agent , predicate) —> mental_state

5.5.81.2 Result:

get the belief in the belief base with the given predicate.

5.5.81.3 Examples:

```
get_belief_op(self,has_water)
```

5.5.82 get_belief_with_name_op

5.5.82.1 Possible use:

- agent **get_belief_with_name_op** string —> mental_state
- **get_belief_with_name_op** (agent , string) —> mental_state

5.5.82.2 Result:

get the belief in the belief base with the given name.

5.5.82.3 Examples:

```
get_belief_with_name_op(self,"has_water")
```

5.5.83 get_beliefs_op

5.5.83.1 Possible use:

- agent **get_beliefs_op** predicate —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState>
- **get_beliefs_op** (agent , predicate) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState

5.5.83.2 Result:

get the beliefs in the belief base with the given predicate.

5.5.83.3 Examples:

```
get_beliefs_op(self,has_water)
```

5.5.84 get_beliefs_with_name_op

5.5.84.1 Possible use:

- agent **get_beliefs_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalS
- **get_beliefs_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Ment

5.5.84.2 Result:

get the list of beliefs in the belief base which predicate has the given name.

5.5.84.3 Examples:

```
get_beliefs_with_name_op(self,"has_water")
```

5.5.85 get_current_intention_op

5.5.85.1 Possible use:

- **get_current_intention_op** (agent) —> mental_state

5.5.85.2 Result:

get the current intention.

5.5.85.3 Examples:

```
get_current_intention_op(self,has_water)
```

5.5.86 get_decay

5.5.86.1 Possible use:

- `get_decay` (`emotion`) —> `float`

5.5.86.2 Result:

get the decay value of the given emotion

5.5.86.3 Examples:

```
get_decay(emotion)
```

5.5.87 get_desire_op

5.5.87.1 Possible use:

- `agent` `get_desire_op` `predicate` —> `mental_state`
- `get_desire_op` (`agent` , `predicate`) —> `mental_state`

5.5.87.2 Result:

get the desire in the desire base with the given predicate.

5.5.87.3 Examples:

```
get_belief_op(self,has_water)
```

5.5.88 get_desire_with_name_op

5.5.88.1 Possible use:

- `agent` `get_desire_with_name_op` `string` —> `mental_state`
- `get_desire_with_name_op` (`agent` , `string`) —> `mental_state`

5.5.88.2 Result:

get the desire in the desire base with the given name.

5.5.88.3 Examples:

```
mental_state var0 <- get_desire_with_name_op(self,"has_water"); // var0 equals nil
```

5.5.89 get_desires_op

5.5.89.1 Possible use:

- agent **get_desires_op** predicate —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState>
- **get_desires_op** (agent , predicate) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState

5.5.89.2 Result:

get the desires in the desire base with the given predicate.

5.5.89.3 Examples:

```
get_desires_op(self,has_water)
```

5.5.90 get_desires_with_name_op

5.5.90.1 Possible use:

- agent **get_desires_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalS
- **get_desires_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Ment

5.5.90.2 Result:

get the list of desires in the desire base which predicate has the given name.

5.5.90.3 Examples:

```
get_desires_with_name_op(self,"has_water")
```

5.5.91 get_dominance

5.5.91.1 Possible use:

- **get_dominance** (msi.gaml.architecture.simplebdi.SocialLink) —> float

5.5.91.2 Result:

get the dominance value of the given social link

5.5.91.3 Examples:

```
get_dominance(social_link1)
```

5.5.92 get_familiarity

5.5.92.1 Possible use:

- `get_familiarity` (`msi.gaml.architecture.simplebdi.SocialLink`) —> `float`

5.5.92.2 Result:

get the familiarity value of the given social link

5.5.92.3 Examples:

```
get_familiarity(social_link1)
```

5.5.93 get_ideal_op

5.5.93.1 Possible use:

- `agent` `get_ideal_op` `predicate` —> `mental_state`
- `get_ideal_op` (`agent` , `predicate`) —> `mental_state`

5.5.93.2 Result:

get the ideal in the ideal base with the given name.

5.5.93.3 Examples:

```
get_ideal_op(self,has_water)
```

5.5.94 get_ideal_with_name_op

5.5.94.1 Possible use:

- `agent` `get_ideal_with_name_op` `string` —> `mental_state`
- `get_ideal_with_name_op` (`agent` , `string`) —> `mental_state`

5.5.94.2 Result:

get the ideal in the ideal base with the given name.

5.5.94.3 Examples:

```
get_ideal_with_name_op(self,"has_water")
```

5.5.95 get_ideals_op

5.5.95.1 Possible use:

- agent **get_ideals_op** predicate —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState>
- **get_ideals_op** (agent , predicate) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState>

5.5.95.2 Result:

get the ideal in the ideal base with the given name.

5.5.95.3 Examples:

```
get_ideals_op(self,has_water)
```

5.5.96 get_ideals_with_name_op

5.5.96.1 Possible use:

- agent **get_ideals_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalSt
- **get_ideals_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Menta

5.5.96.2 Result:

get the list of ideals in the ideal base which predicate has the given name.

5.5.96.3 Examples:

```
get_ideals_with_name_op(self,"has_water")
```

5.5.97 get_intensity

5.5.97.1 Possible use:

- **get_intensity** (emotion) —> float

5.5.97.2 Result:

get the intensity value of the given emotion

5.5.97.3 Examples:

```
emotion set_intensity 12
```

5.5.98 get_intention_op

5.5.98.1 Possible use:

- agent **get_intention_op** predicate —> mental_state
- **get_intention_op** (agent , predicate) —> mental_state

5.5.98.2 Result:

get the intention in the intention base with the given predicate.

5.5.98.3 Examples:

```
get_intention_op(self,has_water)
```

5.5.99 get_intention_with_name_op

5.5.99.1 Possible use:

- agent **get_intention_with_name_op** string —> mental_state
- **get_intention_with_name_op** (agent , string) —> mental_state

5.5.99.2 Result:

get the intention in the intention base with the given name.

5.5.99.3 Examples:

```
get_intention_with_name_op(self,"has_water")
```

5.5.100 get_intentions_op

5.5.100.1 Possible use:

- agent **get_intentions_op** predicate —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalState
- **get_intentions_op** (agent , predicate) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalSt

5.5.100.2 Result:

get the intentions in the intention base with the given predicate.

5.5.100.3 Examples:

```
get_intentions_op(self,has_water)
```

5.5.101 get_intentions_with_name_op

5.5.101.1 Possible use:

- agent **get_intentions_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Ment
- **get_intentions_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.M

5.5.101.2 Result:

get the list of intentions in the intention base which predicate has the given name.

5.5.101.3 Examples:

```
get_intentions_with_name_op(self,"has_water")
```

5.5.102 get_lifetime

5.5.102.1 Possible use:

- **get_lifetime** (predicate) —> int
- **get_lifetime** (mental_state) —> int

5.5.102.2 Result:

get the lifetime value of the given mental state

5.5.102.3 Examples:

```
get_lifetime(mental_state1)
```

5.5.103 get_liking

5.5.103.1 Possible use:

- **get_liking** (msi.gaml.architecture.simplebdi.SocialLink) —> float

5.5.103.2 Result:

get the liking value of the given social link

5.5.103.3 Examples:

```
get_liking(social_link1)
```

5.5.104 get_modality

5.5.104.1 Possible use:

- `get_modality` (`mental_state`) —> `string`

5.5.104.2 Result:

get the modality value of the given mental state

5.5.104.3 Examples:

```
get_modality(mental_state1)
```

5.5.105 get_obligation_op

5.5.105.1 Possible use:

- `agent` **`get_obligation_op`** `predicate` —> `mental_state`
- **`get_obligation_op`** (`agent` , `predicate`) —> `mental_state`

5.5.105.2 Result:

get the obligation in the obligation base with the given predicate.

5.5.105.3 Examples:

```
get_obligation_op(self,has_water)
```

5.5.106 get_obligation_with_name_op

5.5.106.1 Possible use:

- `agent` **`get_obligation_with_name_op`** `string` —> `mental_state`
- **`get_obligation_with_name_op`** (`agent` , `string`) —> `mental_state`

5.5.106.2 Result:

get the obligation in the obligation base with the given name.

5.5.106.3 Examples:

```
get_obligation_with_name_op(self,"has_water")
```

5.5.107 get_obligations_op

5.5.107.1 Possible use:

- agent **get_obligations_op** predicate —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalStat
- **get_obligations_op** (agent , predicate) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalS

5.5.107.2 Result:

get the obligations in the obligation base with the given predicate.

5.5.107.3 Examples:

```
get_obligations_op(self,has_water)
```

5.5.108 get_obligations_with_name_op

5.5.108.1 Possible use:

- agent **get_obligations_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Mer
- **get_obligations_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.

5.5.108.2 Result:

get the list of obligations in the obligation base which predicate has the given name.

5.5.108.3 Examples:

```
get_obligations_with_name_op(self,"has_water")
```

5.5.109 get_plan_name

5.5.109.1 Possible use:

- **get_plan_name** (BDIPlan) —> string

5.5.109.2 Result:

get the name of a given plan

5.5.109.3 Examples:

```
get_plan_name(agent.current_plan)
```

5.5.110 get_predicate

5.5.110.1 Possible use:

- `get_predicate` (`mental_state`) —> `predicate`

5.5.110.2 Result:

get the predicate value of the given mental state

5.5.110.3 Examples:

```
get_predicate(mental_state1)
```

5.5.111 get_solidarity

5.5.111.1 Possible use:

- `get_solidarity` (`msi.gaml.architecture.simplebdi.SocialLink`) —> `float`

5.5.111.2 Result:

get the solidarity value of the given social link

5.5.111.3 Examples:

```
get_solidarity(social_link1)
```

5.5.112 get_strength

5.5.112.1 Possible use:

- `get_strength` (`mental_state`) —> `float`

5.5.112.2 Result:

get the strength value of the given mental state

5.5.112.3 Examples:

```
get_strength(mental_state1)
```

5.5.113 get_super_intention

5.5.113.1 Possible use:

- `get_super_intention` (`predicate`) —> `mental_state`

5.5.114 get_trust

5.5.114.1 Possible use:

- `get_trust` (`msi.gaml.architecture.simplebdi.SocialLink`) —> `float`

5.5.114.2 Result:

get the familiarity value of the given social link

5.5.114.3 Examples:

```
get_familiarity(social_link1)
```

5.5.115 get_truth

5.5.115.1 Possible use:

- `get_truth` (`predicate`) —> `bool`

5.5.116 get_uncertainties_op

5.5.116.1 Possible use:

- `agent` **`get_uncertainties_op`** `predicate` —> `msi.gama.util.IList<msi.gaml.architecture.simplebdi.MentalSt`
- **`get_uncertainties_op`** (`agent` , `predicate`) —> `msi.gama.util.IList<msi.gaml.architecture.simplebdi.Menta`

5.5.116.2 Result:

get the uncertainties in the uncertainty base with the given predicate.

5.5.116.3 Examples:

```
get_uncertinties_op(self,has_water)
```

5.5.117 get_uncertainties_with_name_op

5.5.117.1 Possible use:

- agent **get_uncertainties_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.M
- **get_uncertainties_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebd

5.5.117.2 Result:

get the list of uncertainties in the uncertainty base which predicate has the given name.

5.5.117.3 Examples:

```
get_uncertainties_with_name_op(self,"has_water")
```

5.5.118 get_uncertainty_op

5.5.118.1 Possible use:

- agent **get_uncertainty_op** predicate —> mental_state
- **get_uncertainty_op** (agent , predicate) —> mental_state

5.5.118.2 Result:

get the uncertainty in the uncertainty base with the given predicate.

5.5.118.3 Examples:

```
get_uncertainty_op(self,has_water)
```

5.5.119 get_uncertainty_with_name_op

5.5.119.1 Possible use:

- agent **get_uncertainty_with_name_op** string —> mental_state
- **get_uncertainty_with_name_op** (agent , string) —> mental_state

5.5.119.2 Result:

get the uncertainty in the uncertainty base with the given name.

5.5.119.3 Examples:

```
get_uncertainty_with_name_op(self,"has_water")
```

5.5.120 gif_file

5.5.120.1 Possible use:

- `gif_file` (`string`) —> `file`

5.5.120.2 Result:

Constructs a file of type gif. Allowed extensions are limited to gif

---

5.5.121 gini

5.5.121.1 Possible use:

- `gini` (`list<float>`) —> `float`

5.5.121.2 Special cases:

- return the Gini Index of the given list of values (list of floats)

```
float var0 <- gini([1.0, 0.5, 2.0]); // var0 equals the gini index computed
```

5.5.122 gml_file

5.5.122.1 Possible use:

- `gml_file` (`string`) —> `file`

5.5.122.2 Result:

Constructs a file of type gml. Allowed extensions are limited to gml

---

5.5.123 graph

5.5.123.1 Possible use:

- `graph` (`any`) —> `graph`

5.5.123.2 Result:

Casts the operand into the type graph

5.5.124 grayscale

5.5.124.1 Possible use:

- `grayscale` (`rgb`) —> `rgb`

5.5.124.2 Result:

Converts rgb color to grayscale value

5.5.124.3 Comment:

r=red, g=green, b=blue. Between 0 and 255 and gray = 0.299 * red + 0.587 * green + 0.114 * blue (Photoshop value)

5.5.124.4 Examples:

```
rgb var0 <- grayscale (rgb(255,0,0)); // var0 equals to a dark grey
```

5.5.124.5 See also:

rgb, hsb,

5.5.125 grid_at

5.5.125.1 Possible use:

- `species` `grid_at` `point` —> `agent`
- `grid_at` (`species` , `point`) —> `agent`

5.5.125.2 Result:

returns the cell of the grid (right-hand operand) at the position given by the right-hand operand

5.5.125.3 Comment:

If the left-hand operand is a point of floats, it is used as a point of ints.

5.5.125.4 Special cases:

- if the left-hand operand is not a grid cell species, returns nil

5.5.125.5 Examples:

```
agent var0 <- grid_cell grid_at {1,2}; // var0 equals the agent grid_cell with grid_x=1 and grid_y = 2
```

5.5.126 grid_cells_to_graph

5.5.126.1 Possible use:

- **grid_cells_to_graph** (container) —> graph

5.5.126.2 Result:

creates a graph from a list of cells (operand). An edge is created between neighbors.

5.5.126.3 Examples:

```
my_cell_graph<-grid_cells_to_graph(cells_list)
```

5.5.127 grid_file

5.5.127.1 Possible use:

- **grid_file** (string) —> file

5.5.127.2 Result:

Constructs a file of type grid. Allowed extensions are limited to asc, tif

---

5.5.128 group_by

5.5.128.1 Possible use:

- container **group_by** any expression —> map
- **group_by** (container , any expression) —> map

5.5.128.2 Result:

Returns 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

5.5.128.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

5.5.128.4 Special cases:

- if the left-hand operand is nil, group_by throws an error

5.5.128.5 Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] group_by (each > 3); // var0 equals [false::[1, 2, 3], true::[4, 5, 6, 7, 8]]
map var1 <- g2 group_by (length(g2 out_edges_of each) ); // var1 equals [ 0::[node9, node7, node10, node8, no
map var2 <- (list(node) group_by (round(node(each).location.x)); // var2 equals [32::[node5], 21::[node1],
map<bool,list> var3 <- [1::2, 3::4, 5::6] group_by (each > 4); // var3 equals [false::[2, 4], true::[6]]
```

5.5.128.6 See also:

first_with, last_with, where,

5.5.129 harmonic_mean

5.5.129.1 Possible use:

- **harmonic_mean** (container) —> float

5.5.129.2 Result:

the harmonic mean of the elements of the operand. See Harmonic_mean for more details.

5.5.129.3 Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

5.5.129.4 Examples:

```
float var0 <- harmonic_mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 4.804159445407279
```

5.5.129.5 See also:

mean, median, geometric_mean,

5.5.130 has_belief_op

5.5.130.1 Possible use:

- agent **has_belief_op** predicate —> bool
- **has_belief_op** (agent , predicate) —> bool

5.5.130.2 Result:

indicates if there already is a belief about the given predicate.

5.5.130.3 Examples:

```
has_belief_op(self,has_water)
```

5.5.131 has_belief_with_name_op

5.5.131.1 Possible use:

- agent **has_belief_with_name_op** string —> bool
- **has_belief_with_name_op** (agent , string) —> bool

5.5.131.2 Result:

indicates if there already is a belief about the given name.

5.5.131.3 Examples:

```
has_belief_with_name_op(self,"has_water")
```

5.5.132 has_desire_op

5.5.132.1 Possible use:

- agent **has_desire_op** predicate —> bool
- **has_desire_op** (agent , predicate) —> bool

5.5.132.2 Result:

indicates if there already is a desire about the given predicate.

5.5.132.3 Examples:

```
has_desire_op(self,has_water)
```

5.5.133 has_desire_with_name_op

5.5.133.1 Possible use:

- agent **has_desire_with_name_op** string —> bool
- **has_desire_with_name_op** (agent , string) —> bool

5.5.133.2 Result:

indicates if there already is a desire about the given name.

5.5.133.3 Examples:

```
has_desire_with_name_op(self,"has_water")
```

5.5.134 has_ideal_op

5.5.134.1 Possible use:

- agent **has_ideal_op** predicate —> bool
- **has_ideal_op** (agent , predicate) —> bool

5.5.134.2 Result:

indicates if there already is an ideal about the given predicate.

5.5.134.3 Examples:

```
has_ideal_op(self,has_water)
```

5.5.135 has_ideal_with_name_op

5.5.135.1 Possible use:

- agent **has_ideal_with_name_op** string —> bool
- **has_ideal_with_name_op** (agent , string) —> bool

5.5.135.2 Result:

indicates if there already is an ideal about the given name.

5.5.135.3 Examples:

```
has_ideal_with_name_op(self,"has_water")
```

5.5.136 has_intention_op

5.5.136.1 Possible use:

- agent **has_intention_op** predicate —> bool
- **has_intention_op** (agent , predicate) —> bool

5.5.136.2 Result:

indicates if there already is an intention about the given predicate.

5.5.136.3 Examples:

```
has_intention_op(self,has_water)
```

5.5.137 has_intention_with_name_op

5.5.137.1 Possible use:

- agent **has_intention_with_name_op** string —> bool
- **has_intention_with_name_op** (agent , string) —> bool

5.5.137.2 Result:

indicates if there already is an intention about the given name.

5.5.137.3 Examples:

```
has_intention_with_name_op(self,"has_water")
```

5.5.138 has_obligation_op

5.5.138.1 Possible use:

- agent **has_obligation_op** predicate —> bool
- **has_obligation_op** (agent , predicate) —> bool

5.5.138.2 Result:

indicates if there already is an obligation about the given predicate.

5.5.138.3 Examples:

```
has_obligation_op(self,has_water)
```

5.5.139 has_obligation_with_name_op

5.5.139.1 Possible use:

- agent **has_obligation_with_name_op** string —> bool
- **has_obligation_with_name_op** (agent , string) —> bool

5.5.139.2 Result:

indicates if there already is an obligation about the given name.

5.5.139.3 Examples:

```
has_obligation_with_name_op(self,"has_water")
```

5.5.140 has_uncertainty_op

5.5.140.1 Possible use:

- agent **has_uncertainty_op** predicate —> bool
- **has_uncertainty_op** (agent , predicate) —> bool

5.5.140.2 Result:

indicates if there already is an uncertainty about the given predicate.

5.5.140.3 Examples:

```
has_uncertainty_op(self,has_water)
```

5.5.141 has_uncertainty_with_name_op

5.5.141.1 Possible use:

- agent **has_uncertainty_with_name_op** string —> bool
- **has_uncertainty_with_name_op** (agent , string) —> bool

5.5.141.2 Result:

indicates if there already is an uncertainty about the given name.

5.5.141.3 Examples:

```
has_uncertainty_with_name_op(self,"has_water")
```

5.5.142 hexagon

5.5.142.1 Possible use:

- **hexagon** (point) —> geometry
- **hexagon** (float) —> geometry
- float **hexagon** float —> geometry
- **hexagon** (float , float) —> geometry

5.5.142.2 Result:

A hexagon geometry which the given with and height

5.5.142.3 Comment:

the center of the hexagon is by default the location of the current agent in which has been called this operator.the center of the hexagon is by default the location of the current agent in which has been called this operator.the center of the hexagon is by default the location of the current agent in which has been called this operator.

5.5.142.4 Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.
- returns nil if the operand is nil.

5.5.142.5 Examples:

```
geometry var0 <- hexagon({10,5}); // var0 equals a geometry as a hexagon of width of 10 and height of 5.
geometry var1 <- hexagon(10); // var1 equals a geometry as a hexagon of width of 10 and height of 10.
geometry var2 <- hexagon(10,5); // var2 equals a geometry as a hexagon of width of 10 and height of 5.
```

5.5.142.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

5.5.143 hierarchical_clustering

5.5.143.1 Possible use:

- container<agent> **hierarchical_clustering** float —> list
- **hierarchical_clustering** (container<agent> , float) —> list

5.5.143.2 Result:

A tree (list of list) contained groups of agents clustered by distance considering a distance min between two groups.

5.5.143.3 Comment:

use of hierarchical clustering with Minimum for linkage criterion between two groups of agents.

5.5.143.4 Examples:

```
list var0 <- [ag1, ag2, ag3, ag4, ag5] hierarchical_clustering 20.0; // var0 equals for example, can return [
```

5.5.143.5 See also:

simple_clustering_by_distance,

5.5.144 horizontal

5.5.144.1 Possible use:

- `horizontal` (`msi.gama.util.GamaMap<java.lang.Object,java.lang.Integer>`) —> `msi.gama.util.tree.GamaNo`

5.5.145 hsb

5.5.145.1 Possible use:

- `hsb` (`float`, `float`, `float`) —> `rgb`
- `hsb` (`float`, `float`, `float`, `int`) —> `rgb`
- `hsb` (`float`, `float`, `float`, `float`) —> `rgb`

5.5.145.2 Result:

Converts hsb (h=hue, s=saturation, b=brightness) value to Gama color

5.5.145.3 Comment:

h,s and b components should be floating-point values between 0.0 and 1.0 and when used alpha should be an integer (between 0 and 255) or a float (between 0 and 1) . Examples: Red=(0.0,1.0,1.0), Yellow=(0.16,1.0,1.0), Green=(0.33,1.0,1.0), Cyan=(0.5,1.0,1.0), Blue=(0.66,1.0,1.0), Magenta=(0.83,1.0,1.0)

5.5.145.4 Examples:

```
rgb var0 <- hsb (0.5,1.0,1.0,0.0); // var0 equals rgb("cyan",0)
rgb var1 <- hsb (0.0,1.0,1.0); // var1 equals rgb("red")
```

5.5.145.5 See also:

rgb,

5.5.146 `hypot`

5.5.146.1 Possible use:

- **`hypot`** (`float`, `float`, `float`, `float`) —> `float`

5.5.146.2 Result:

Returns sqrt(x2 +y2) without intermediate overflow or underflow.

5.5.146.3 Special cases:

- If either argument is infinite, then the result is positive infinity. If either argument is NaN and neither argument is infinite, then the result is NaN.

5.5.146.4 Examples:

```
float var0 <- hypot(0,1,0,1); // var0 equals sqrt(2)
```

Chapter 6

Operators (I to M)

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

6.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

6.2

6.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

6.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

6.5 Operators

6.5.1 IDW

6.5.1.1 Possible use:

- **IDW** (container<agent>, map<point,float>, int) —> map<agent,float>

6.5.1.2 Result:

Inverse Distance Weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to each geometry are calculated with a weighted average of the values available at the known points. See: http://en.wikipedia.org/wiki/Inverse_distance_weighting Usage: IDW (list of geometries, map of points (key: point, value: value), power parameter)

6.5.1.3 Examples:

```
map<agent,float> var0 <- IDW([ag1, ag2, ag3, ag4, ag5],[{10,10}::25.0, {10,80}::10.0, {100,10}::15.0], 2);
```

6.5.2 image_file

6.5.2.1 Possible use:

- **image_file** (string) —> file

6.5.2.2 Result:

Constructs a file of type image. Allowed extensions are limited to tiff, jpg, jpeg, png, pict, bmp

---

6.5.3 improved_generator

6.5.3.1 Possible use:

- **improved_generator** (float, float, float, float) —> float

6.5.3.2 Result:

take a x, y, z and a bias parameters and gives a value

6.5.3.3 Examples:

```
float var0 <- improved_generator(2,3,4,253); // var0 equals 10.2
```

6.5.4 in

6.5.4.1 Possible use:

- `string` **`in`** `string` —> `bool`
- **`in`** (`string` , `string`) —> `bool`
- `unknown` **`in`** `container` —> `bool`
- **`in`** (`unknown` , `container`) —> `bool`

6.5.4.2 Result:

true if the right operand contains the left operand, false otherwise

6.5.4.3 Comment:

the definition of in depends on the container

6.5.4.4 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

6.5.4.5 Examples:

```
bool var0 <-  'bc' in 'abcded'; // var0 equals true
bool var1 <- 2 in [1,2,3,4,5,6]; // var1 equals true
bool var2 <- 7 in [1,2,3,4,5,6]; // var2 equals false
bool var3 <- 3 in [1::2, 3::4, 5::6]; // var3 equals false
bool var4 <- 6 in [1::2, 3::4, 5::6]; // var4 equals true
```

6.5.4.6 See also:

contains,

6.5.5 in_degree_of

6.5.5.1 Possible use:

- graph **in_degree_of** unknown —> int
- **in_degree_of** (graph , unknown) —> int

6.5.5.2 Result:

returns the in degree of a vertex (right-hand operand) in the graph given as left-hand operand.

6.5.5.3 Examples:

```
int var1 <- graphFromMap in_degree_of (node(3)); // var1 equals 2
```

6.5.5.4 See also:

out_degree_of, degree_of,

6.5.6 in_edges_of

6.5.6.1 Possible use:

- graph **in_edges_of** unknown —> list
- **in_edges_of** (graph , unknown) —> list

6.5.6.2 Result:

returns the list of the in-edges of a vertex (right-hand operand) in the graph given as left-hand operand.

6.5.6.3 Examples:

```
list var1 <- graphFromMap in_edges_of node({12,45}); // var1 equals [LineString]
```

6.5.6.4 See also:

out_edges_of,

6.5.7 incomplete_beta

6.5.7.1 Possible use:

- **incomplete_beta** (float, float, float) —> float

6.5.7.2 Result:

Returns the regularized integral of the beta function with arguments a and b, from zero to x.

6.5.8 `incomplete_gamma`

6.5.8.1 Possible use:

- `float` **`incomplete_gamma`** `float` —> `float`
- **`incomplete_gamma`** (`float` , `float`) —> `float`

6.5.8.2 Result:

Returns the regularized integral of the Gamma function with argument a to the integration end point x.

6.5.9 `incomplete_gamma_complement`

6.5.9.1 Possible use:

- `float` **`incomplete_gamma_complement`** `float` —> `float`
- **`incomplete_gamma_complement`** (`float` , `float`) —> `float`

6.5.9.2 Result:

Returns the complemented regularized incomplete Gamma function of the argument a and integration start point x.

6.5.10 `indented_by`

6.5.10.1 Possible use:

- `string` **`indented_by`** `int` —> `string`
- **`indented_by`** (`string` , `int`) —> `string`

6.5.10.2 Result:

Converts a (possibly multiline) string by indenting it by a number – specified by the second operand – of tabulations to the right

---

6.5.11 index_by

6.5.11.1 Possible use:

- container **index_by** any expression —> map
- **index_by** (container , any expression) —> map

6.5.11.2 Result:

produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand

6.5.11.3 Special cases:

- if the left-hand operand is nil, index_by throws an error. If the operation results in duplicate keys, only the first value corresponding to the key is kept

6.5.11.4 Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] index_by (each - 1); // var0 equals [0::1, 1::2, 2::3, 3::4, 4::5, 5::6, 6::7,
```

6.5.12 index_of

6.5.12.1 Possible use:

- map **index_of** unknown —> unknown
- **index_of** (map , unknown) —> unknown
- matrix **index_of** unknown —> point
- **index_of** (matrix , unknown) —> point
- list **index_of** unknown —> int
- **index_of** (list , unknown) —> int
- string **index_of** string —> int
- **index_of** (string , string) —> int
- species **index_of** unknown —> int
- **index_of** (species , unknown) —> int

6.5.12.2 Result:

the index of the first occurence of the right operand in the left operand container the index of the first occurence of the right operand in the left operand container

6.5.12.3 Comment:

The definition of index_of and the type of the index depend on the container

6.5.12.4 Special cases:

- if the left operand is a map, index_of returns the index of a value or nil if the value is not mapped
- if the left operator is a species, returns the index of an agent in a species. If the argument is not an agent of this species, returns -1. Use int(agent) instead
- if the left operand is a matrix, index_of returns the index as a point

```
point var1 <- matrix([[1,2,3],[4,5,6]]) index_of 4; // var1 equals {1.0,0.0}
```

- if the left operand is a list, index_of returns the index as an integer

```
int var2 <- [1,2,3,4,5,6] index_of 4; // var2 equals 3
int var3 <- [4,2,3,4,5,4] index_of 4; // var3 equals 0
```

- if both operands are strings, returns the index within the left-hand string of the first occurrence of the given right-hand string

```
int var4 <-  "abcabcabc" index_of "ca"; // var4 equals 2
```

6.5.12.5 Examples:

```
unknown var0 <- [1::2, 3::4, 5::6] index_of 4; // var0 equals 3
```

6.5.12.6 See also:

at, last_index_of,

6.5.13 inside

6.5.13.1 Possible use:

- `container<agent>` **`inside`** `geometry` —> `list<geometry>`
- **`inside`** (`container<agent>` , `geometry`) —> `list<geometry>`

6.5.13.2 Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), covered by the operand (casted as a geometry).

6.5.13.3 Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] inside(self); // var0 equals the agents among ag1, ag2 and ag3 that ar
list<geometry> var1 <- (species1 + species2) inside (self); // var1 equals the agents among species species1
```

6.5.13.4 See also:

neighbors_at, neighbors_of, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

6.5.14 `int`

6.5.14.1 Possible use:

- `int` (`any`) —> `int`

6.5.14.2 Result:

Casts the operand into the type int

6.5.15 `inter`

6.5.15.1 Possible use:

- `container` **`inter`** `container` —> `list`
- **`inter`** (`container` , `container`) —> `list`
- `geometry` **`inter`** `geometry` —> `geometry`
- **`inter`** (`geometry` , `geometry`) —> `geometry`

6.5.15.2 Result:

the intersection of the two operands A geometry resulting from the intersection between the two geometries

6.5.15.3 Comment:

both containers are transformed into sets (so without duplicated element, cf. remove_deplicates operator) before the set intersection is computed.

6.5.15.4 Special cases:

- if an operand is a graph, it will be transformed into the set of its nodes
- returns nil if one of the operands is nil
- if an operand is a map, it will be transformed into the set of its values

```
list var0 <- [1::2, 3::4, 5::6] inter [2,4]; // var0 equals [2,4]
list var1 <- [1::2, 3::4, 5::6] inter [1,3]; // var1 equals []
```

- if an operand is a matrix, it will be transformed into the set of the lines

```
list var2 <- matrix([[3,2,1],[4,5,4]]) inter [3,4]; // var2 equals [3,4]
```

6.5.15.5 Examples:

```
list var3 <- [1,2,3,4,5,6] inter [2,4]; // var3 equals [2,4]
list var4 <- [1,2,3,4,5,6] inter [0,8]; // var4 equals []
geometry var5 <- square(10) inter circle(5); // var5 equals circle(5)
```

6.5.15.6 See also:

remove_duplicates, union, +, -,

6.5.16 interleave

6.5.16.1 Possible use:

- **interleave** (container) —> list

6.5.16.2 Result:

a new list containing the interleaved elements of the containers contained in the operand

6.5.16.3 Comment:

the operand should be a list of lists of elements. The result is a list of elements.

6.5.16.4 Examples:

```
list var0 <- interleave([1,2,4,3,5,7,6,8]); // var0 equals [1,2,4,3,5,7,6,8]
list var1 <- interleave([['e11','e12','e13'],['e21','e22','e23'],['e31','e32','e33']]); // var1 equals ['e
```

6.5.17 internal_at

6.5.17.1 Possible use:

- container<KeyType,ValueType> **internal_at** list<KeyType> —> ValueType
- **internal_at** (container<KeyType,ValueType> , list<KeyType>) —> ValueType
- agent **internal_at** list —> unknown
- **internal_at** (agent , list) —> unknown
- geometry **internal_at** list —> unknown
- **internal_at** (geometry , list) —> unknown

6.5.17.2 Result:

For internal use only. Corresponds to the implementation of the access to containers with [index] For internal use only. Corresponds to the implementation, for agents, of the access to containers with [index] For internal use only. Corresponds to the implementation, for geometries, of the access to containers with [index]

6.5.17.3 See also:

at,

6.5.18 `internal_integrated_value`

6.5.18.1 Possible use:

- `any expression` **`internal_integrated_value`** `any expression` —> `list`
- **`internal_integrated_value`** (`any expression` , `any expression`) —> `list`

6.5.18.2 Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with [index]

6.5.19 `internal_zero_order_equation`

6.5.19.1 Possible use:

- **`internal_zero_order_equation`** (`any expression`) —> `float`

6.5.19.2 Result:

An internal placeholder function

6.5.20 `intersection`

Same signification as inter

6.5.21 `intersects`

6.5.21.1 Possible use:

- `geometry` **`intersects`** `geometry` —> `bool`
- **`intersects`** (`geometry` , `geometry`) —> `bool`

6.5.21.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) intersects the right-geometry (or agent/point).

6.5.21.3 Special cases:

- if one of the operand is null, returns false.

6.5.21.4 Examples:

```
bool var0 <- square(5) intersects {10,10}; // var0 equals false
```

6.5.21.5 See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

6.5.22 inverse

6.5.22.1 Possible use:

- `inverse` (`matrix`) —> `matrix<float>`

6.5.22.2 Result:

The inverse matrix of the given matrix. If no inverse exists, returns a matrix that has properties that resemble that of an inverse.

6.5.22.3 Examples:

```
matrix<float> var0 <- inverse(matrix([[4,3],[3,2]])); // var0 equals matrix([[-2.0,3.0],[3.0,-4.0]])
```

6.5.23 inverse_distance_weighting

Same signification as IDW

6.5.24 is

6.5.24.1 Possible use:

- `unknown is any expression` —> `bool`
- `is` (`unknown` , `any expression`) —> `bool`

6.5.24.2 Result:

returns true if the left operand is of the right operand type, false otherwise

6.5.24.3 Examples:

```
bool var0 <- 0 is int; // var0 equals true
bool var1 <- an_agent is node; // var1 equals true
bool var2 <- 1 is float; // var2 equals false
```

6.5.25 `is_csv`

6.5.25.1 Possible use:

- `is_csv` (`any`) —> `bool`

6.5.25.2 Result:

Tests whether the operand is a csv file.

6.5.26 `is_dxf`

6.5.26.1 Possible use:

- `is_dxf` (`any`) —> `bool`

6.5.26.2 Result:

Tests whether the operand is a dxf file.

6.5.27 `is_error`

6.5.27.1 Possible use:

- `is_error` (`any expression`) —> `bool`

6.5.27.2 Result:

Returns whether or not the argument raises an error when evaluated

---

6.5.28 is_finite

6.5.28.1 Possible use:

- `is_finite` (`float`) —> `bool`

6.5.28.2 Result:

Returns whether the argument is a finite number or not

6.5.28.3 Examples:

```
bool var0 <- is_finite(4.66); // var0 equals true
bool var1 <- is_finite(#infinity); // var1 equals false
```

6.5.29 is_gaml

6.5.29.1 Possible use:

- `is_gaml` (`any`) —> `bool`

6.5.29.2 Result:

Tests whether the operand is a gaml file.

6.5.30 is_geojson

6.5.30.1 Possible use:

- `is_geojson` (`any`) —> `bool`

6.5.30.2 Result:

Tests whether the operand is a geojson file.

6.5.31 is_gif

6.5.31.1 Possible use:

- `is_gif` (`any`) —> `bool`

6.5.31.2 Result:

Tests whether the operand is a gif file.

6.5.32 is_gml

6.5.32.1 Possible use:

- `is_gml` (`any`) —> `bool`

6.5.32.2 Result:

Tests whether the operand is a gml file.

6.5.33 is_grid

6.5.33.1 Possible use:

- `is_grid` (`any`) —> `bool`

6.5.33.2 Result:

Tests whether the operand is a grid file.

6.5.34 is_image

6.5.34.1 Possible use:

- `is_image` (`any`) —> `bool`

6.5.34.2 Result:

Tests whether the operand is a image file.

6.5.35 is_json

6.5.35.1 Possible use:

- `is_json` (`any`) —> `bool`

6.5.35.2 Result:

Tests whether the operand is a json file.

6.5.36 `is_number`

6.5.36.1 Possible use:

- `is_number` (`string`) —> `bool`
- `is_number` (`float`) —> `bool`

6.5.36.2 Result:

tests whether the operand represents a numerical value Returns whether the argument is a real number or not

6.5.36.3 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 #.

6.5.36.4 Examples:

```
bool var0 <- is_number("test"); // var0 equals false
bool var1 <- is_number("123.56"); // var1 equals true
bool var2 <- is_number("-1.2e5"); // var2 equals true
bool var3 <- is_number("1,2"); // var3 equals false
bool var4 <- is_number("#12FA"); // var4 equals true
bool var5 <- is_number(4.66); // var5 equals true
bool var6 <- is_number(#infinity); // var6 equals true
bool var7 <- is_number(#nan); // var7 equals false
```

6.5.37 `is_obj`

6.5.37.1 Possible use:

- `is_obj` (`any`) —> `bool`

6.5.37.2 Result:

Tests whether the operand is a obj file.

6.5.38 is_osm

6.5.38.1 Possible use:

- `is_osm` (`any`) —> `bool`

6.5.38.2 Result:

Tests whether the operand is a osm file.

6.5.39 is_pgm

6.5.39.1 Possible use:

- `is_pgm` (`any`) —> `bool`

6.5.39.2 Result:

Tests whether the operand is a pgm file.

6.5.40 is_property

6.5.40.1 Possible use:

- `is_property` (`any`) —> `bool`

6.5.40.2 Result:

Tests whether the operand is a property file.

6.5.41 is_R

6.5.41.1 Possible use:

- `is_R` (`any`) —> `bool`

6.5.41.2 Result:

Tests whether the operand is a R file.

6.5.42 is_shape

6.5.42.1 Possible use:

- `is_shape` (`any`) —> `bool`

6.5.42.2 Result:

Tests whether the operand is a shape file.

6.5.43 is_skill

6.5.43.1 Possible use:

- `unknown is_skill string` —> `bool`
- `is_skill` (`unknown` , `string`) —> `bool`

6.5.43.2 Result:

returns true if the left operand is an agent whose species implements the right-hand skill name

6.5.43.3 Examples:

```
bool var0 <- agentA is_skill 'moving'; // var0 equals true
```

6.5.44 is_svg

6.5.44.1 Possible use:

- `is_svg` (`any`) —> `bool`

6.5.44.2 Result:

Tests whether the operand is a svg file.

6.5.45 is_text

6.5.45.1 Possible use:

- `is_text` (`any`) —> `bool`

6.5.45.2 Result:

Tests whether the operand is a text file.

6.5.46 is_threeds

6.5.46.1 Possible use:

- `is_threeds` (`any`) —> `bool`

6.5.46.2 Result:

Tests whether the operand is a threeds file.

6.5.47 is_URL

6.5.47.1 Possible use:

- `is_URL` (`any`) —> `bool`

6.5.47.2 Result:

Tests whether the operand is a URL file.

6.5.48 is_warning

6.5.48.1 Possible use:

- `is_warning` (`any expression`) —> `bool`

6.5.48.2 Result:

Returns whether or not the argument raises a warning when evaluated

---

6.5.49 is_xml

6.5.49.1 Possible use:

- `is_xml` (`any`) —> `bool`

6.5.49.2 Result:

Tests whether the operand is a xml file.

6.5.50 json_file

6.5.50.1 Possible use:

- `json_file` (`string`) —> `file`

6.5.50.2 Result:

Constructs a file of type json. Allowed extensions are limited to json

---

6.5.51 kappa

6.5.51.1 Possible use:

- `kappa` (`list`, `list`, `list`) —> `float`
- `kappa` (`list`, `list`, `list`, `list`) —> `float`

6.5.51.2 Result:

kappa indicator for 2 map comparisons: kappa(list_vals1,list_vals2,categories). Reference: Cohen, J. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 1960, 20. kappa indicator for 2 map comparisons: kappa(list_vals1,list_vals2,categories, weights). Reference: Cohen, J. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 1960, 20.

6.5.51.3 Examples:

```
kappa([cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],[cat1,cat2,cat3])
float var1 <- kappa([1,3,5,1,5],[1,1,1,1,5],[1,3,5]); // var1 equals the similarity between 0 and 1
float var2 <- kappa([1,1,1,1,5],[1,1,1,1,5],[1,3,5]); // var2 equals 1.0kappa([cat1,cat1,cat2,cat3,cat2],[
```

6.5.52 kappa_sim

6.5.52.1 Possible use:

- `kappa_sim` (`list`, `list`, `list`, `list`) —> `float`
- `kappa_sim` (`list`, `list`, `list`, `list`, `list`) —> `float`

6.5.52.2 Result:

kappa simulation indicator for 2 map comparisons: kappa(list_valsInits,list_valsObs,list_valsSim, categories, weights). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8) kappa simulation indicator for 2 map comparisons: kappa(list_valsInits,list_valsObs,list_valsSim, categories). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8).

6.5.52.3 Examples:

```
kappa([cat1,cat1,cat2,cat2,cat2],[cat2,cat1,cat2,cat1,cat3],[cat2,cat1,cat2,cat3,cat3], [cat1,cat2,cat3]
```

6.5.53 kmeans

6.5.53.1 Possible use:

- `list` **`kmeans`** `int` —> `list<list>`
- **`kmeans`** (`list` , `int`) —> `list<list>`
- **`kmeans`** (`list`, `int`, `int`) —> `list<list>`

6.5.53.2 Result:

returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k) and the maximum number of iterations to run the algorithm for (If negative, no maximum will be used) (maxIt). Usage: kmeans(data,k,maxit) returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k). Usage: kmeans(data,k)

6.5.53.3 Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0
- if the lengths of two vectors in the right-hand aren't equal, returns 0

6.5.53.4 Examples:

```
kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2,10)
list<list> var1 <- kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2); // var1 equals []
```

6.5.54 kurtosis

6.5.54.1 Possible use:

- **`kurtosis`** (`list`) —> `float`

6.5.54.2 Result:

returns kurtosis value computed from the operand list of values

6.5.54.3 Special cases:

- if the length of the list is lower than 3, returns NaN

6.5.54.4 Examples:

```
float var0 <- kurtosis ([1,2,3,4,5]); // var0 equals 1.0
```

6.5.55 kurtosis

6.5.55.1 Possible use:

- **kurtosis** (container) —> float
- float **kurtosis** float —> float
- **kurtosis** (float , float) —> float

6.5.55.2 Result:

Returns the kurtosis (aka excess) of a data sequence Returns the kurtosis (aka excess) of a data sequence

---

6.5.56 last

6.5.56.1 Possible use:

- **last** (string) —> string
- **last** (container<KeyType,ValueType>) —> ValueType
- int **last** container —> list
- **last** (int , container) —> list

6.5.56.2 Result:

the last element of the operand

6.5.56.3 Comment:

the last operator behavior depends on the nature of the operand

6.5.56.4 Special cases:

- if it is a map, last returns the value of the last pair (in insertion order)
- 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 a list containing the last edge created
- 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 nil if the matrix is empty
- if it is a string, last returns a string composed of its last character, or an empty string if the operand is empty

```
string var0 <- last ('abce'); // var0 equals 'e'
```

- if it is a list, last returns the last element of the list, or nil if the list is empty

```
int var1 <- last ([1, 2, 3]); // var1 equals 3
```

6.5.56.5 See also:

first,

6.5.57 last_index_of

6.5.57.1 Possible use:

- species **last_index_of** unknown —> int
- **last_index_of** (species , unknown) —> int
- map **last_index_of** unknown —> unknown
- **last_index_of** (map , unknown) —> unknown
- list **last_index_of** unknown —> int
- **last_index_of** (list , unknown) —> int
- matrix **last_index_of** unknown —> point
- **last_index_of** (matrix , unknown) —> point
- string **last_index_of** string —> int
- **last_index_of** (string , string) —> int

6.5.57.2 Result:

the index of the last occurence of the right operand in the left operand container

6.5.57.3 Comment:

The definition of last_index_of and the type of the index depend on the container

6.5.57.4 Special cases:

- if the left operand is a species, the last index of an agent is the same as its index
- if the left operand is a map, last_index_of returns the index as an int (the key of the pair)

```
unknown var0 <- [1::2, 3::4, 5::4] last_index_of 4; // var0 equals 5
```

- if the left operand is a list, last_index_of returns the index as an integer

```
int var1 <- [1,2,3,4,5,6] last_index_of 4; // var1 equals 3
int var2 <- [4,2,3,4,5,4] last_index_of 4; // var2 equals 5
```

- if the left operand is a matrix, last_index_of returns the index as a point

```
point var3 <- matrix([[1,2,3],[4,5,4]]) last_index_of 4; // var3 equals {1.0,2.0}
```

- if both operands are strings, returns the index within the left-hand string of the rightmost occurrence of the given right-hand string

```
int var4 <- "abcabcabc" last_index_of "ca"; // var4 equals 5
```

6.5.57.5 See also:

at, index_of, last_index_of,

6.5.58 last_of

Same signification as last

6.5.59 last_with

6.5.59.1 Possible use:

- `container last_with any expression —> unknown`
- `last_with (container , any expression) —> unknown`

6.5.59.2 Result:

the last element of the left-hand operand that makes the right-hand operand evaluate to true.

6.5.59.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

6.5.59.4 Special cases:

- if the left-hand operand is nil, last_with throws an error.
- If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] last_with (each >= 4); // var4 equals 6
unknown var5 <- [1::2, 3::4, 5::6].pairs last_with (each.value >= 4); // var5 equals (5::6)
```

6.5.59.5 Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] last_with (each > 3); // var0 equals 8
unknown var2 <- g2 last_with (length(g2 out_edges_of each) = 0 ); // var2 equals node11
unknown var3 <- (list(node) last_with (round(node(each).location.x) > 32); // var3 equals node3
```

6.5.59.6 See also:

group_by, first_with, where,

6.5.60 `layout`

6.5.60.1 Possible use:

- `graph` **`layout`** `string` —> `graph`
- **`layout`** (`graph` , `string`) —> `graph`
- **`layout`** (`graph`, `string`, `int`) —> `graph`
- **`layout`** (`graph`, `string`, `int`, `map<string,unknown>`) —> `graph`

6.5.60.2 Result:

layouts a GAMA graph.

6.5.61 `length`

6.5.61.1 Possible use:

- **`length`** (`container<KeyType,ValueType>`) —> `int`
- **`length`** (`string`) —> `int`

6.5.61.2 Result:

the number of elements contained in the operand

6.5.61.3 Comment:

the length operator behavior depends on the nature of the operand

6.5.61.4 Special cases:

- if it is a population, length returns number of agents of the population
- if it is a graph, length returns the number of vertexes or of edges (depending on the way it was created)
- if it is a list or a map, length returns the number of elements in the list or map

```
int var0 <- length([12,13]); // var0 equals 2
int var1 <- length([]); // var1 equals 0
```

- if it is a matrix, length returns the number of cells

```
int var2 <- length(matrix([["c11","c12","c13"],["c21","c22","c23"]])); // var2 equals 6
```

- if it is a string, length returns the number of characters

```
int var3 <- length ('I am an agent'); // var3 equals 13
```

6.5.62 lgamma

Same signification as log_gamma

6.5.63 line

6.5.63.1 Possible use:

- `line` (`container<geometry>`) —> `geometry`
- `container<geometry>` `line` `float` —> `geometry`
- `line` (`container<geometry>` , `float`) —> `geometry`

6.5.63.2 Result:

A polyline geometry from the given list of points represented as a cylinder of radius r. A polyline geometry from the given list of points.

6.5.63.3 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 nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.
- if a radius is added, the given list of points represented as a cylinder of radius r

```
geometry var0 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}],0.2); // var0 equals a polyline geometry composed
```

6.5.63.4 Examples:

```
geometry var1 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}]); // var1 equals a polyline geometry composed of
```

6.5.63.5 See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle,

---

6.5.64 `link`

6.5.64.1 Possible use:

- `geometry` **`link`** `geometry` —> `geometry`
- **`link`** (`geometry` , `geometry`) —> `geometry`

6.5.64.2 Result:

A dynamic line geometry between the location of the two operands

6.5.64.3 Comment:

The geometry of the link is a line between the locations of the two operands, which is built and maintained dynamically

6.5.64.4 Special cases:

- if one of the operands is nil, link returns a point geometry at the location of the other. If both are null, it returns a point geometry at {0,0}

6.5.64.5 Examples:

```
geometry var0 <- link (geom1,geom2); // var0 equals a link geometry between geom1 and geom2.
```

6.5.64.6 See also:

around, circle, cone, line, norm, point, polygon, polyline, rectangle, square, triangle,

---

6.5.65 `list`

6.5.65.1 Possible use:

- `list` (`any`) —> `list`

6.5.65.2 Result:

Casts the operand into the type list

6.5.66 `list_with`

6.5.66.1 Possible use:

- `int list_with any expression` —> `list`
- `list_with` (`int` , `any expression`) —> `list`

6.5.66.2 Result:

creates a list with a size provided by the first operand, and filled with the second operand

6.5.66.3 Comment:

Note that the right operand should be positive, and that the second one is evaluated for each position in the list.

6.5.66.4 See also:

list,

6.5.67 ln

6.5.67.1 Possible use:

- ln (float) —> float
- ln (int) —> float

6.5.67.2 Result:

Returns the natural logarithm (base e) of the operand.

6.5.67.3 Special cases:

- an exception is raised if the operand is less than zero.

6.5.67.4 Examples:

```
float var0 <- ln(exp(1)); // var0 equals 1.0
float var1 <- ln(1); // var1 equals 0.0
```

6.5.67.5 See also:

exp,

6.5.68 load_graph_from_file

6.5.68.1 Possible use:

- load_graph_from_file (string) —> graph
- string load_graph_from_file file —> graph
- load_graph_from_file (string , file) —> graph
- string load_graph_from_file string —> graph
- load_graph_from_file (string , string) —> graph
- load_graph_from_file (string, species, species) —> graph
- load_graph_from_file (string, file, species, species) —> graph
- load_graph_from_file (string, string, species, species) —> graph
- load_graph_from_file (string, string, species, species, bool) —> graph

6.5.68.2 Result:

returns a graph loaded from a given file encoded into a given format. The last boolean parameter indicates whether the resulting graph will be considered as spatial or not by GAMA loads a graph from a file

6.5.68.3 Comment:

Available formats: “pajek”: 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.“lgl”: LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details.“dot”: 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.“edge”: This format is a simple text file with numeric vertex ids defining the edges.“gexf”: 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.“graphml”: GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details.“tlp” or “tulip”: TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. “ncol”: 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:Available formats: “pajek”: 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.“lgl”: LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details.“dot”: 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.“edge”: This format is a simple text file with numeric vertex ids defining the edges.“gexf”: 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.“graphml”: GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details.“tlp” or “tulip”: TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. “ncol”: 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:

6.5.68.4 Special cases:

- “format”: the format of the file
- “filename”: the filename of the file containing the network
- “edges_species”: the species of edges
- “vertices_specy”: the species of vertices
- “format”: the format of the file
- “filename”: the filename of the file containing the network
- “edges_species”: the species of edges

- "vertices_specy": the species of vertices

- "format": the format of the file, "file": the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "example_of_Paje
```

- "format": the format of the file, "filename": the filename of the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "example_of_Paje
```

- "filename": the filename of the file containing the network, "edges_species": the species of edges, "vertices_specy": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "./example_of_Pa
```

- "format": the format of the file, "file": the file containing the network, "edges_species": the species of edges, "vertices_specy": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "example_of_Paje
```

- "file": the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "example_of_Paje
```

6.5.68.5 Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(        "pajek",        "./example_of_Pa
```

6.5.69 load_shortest_paths

6.5.69.1 Possible use:

- graph **load_shortest_paths** matrix —> graph
- **load_shortest_paths** (graph , matrix) —> graph

6.5.69.2 Result:

put in the graph cache the computed shortest paths contained in the matrix (rows: source, columns: target)

6.5.69.3 Examples:

```
graph var0 <- load_shortest_paths(shortest_paths_matrix); // var0 equals return my_graph with all the short
```

6.5.70 load_sub_model

6.5.70.1 Possible use:

- string **load_sub_model** string —> msi.gama.kernel.experiment.IExperimentAgent
- **load_sub_model** (string , string) —> msi.gama.kernel.experiment.IExperimentAgent

6.5.70.2 Result:

Load a submodel

6.5.70.3 Comment:

loaded submodel

6.5.71 `log`

6.5.71.1 Possible use:

- `log` (`float`) —> `float`
- `log` (`int`) —> `float`

6.5.71.2 Result:

Returns the logarithm (base 10) of the operand.

6.5.71.3 Special cases:

- an exception is raised if the operand is equals or less than zero.

6.5.71.4 Examples:

```
float var0 <- log(10); // var0 equals 1.0
float var1 <- log(1); // var1 equals 0.0
```

6.5.71.5 See also:

ln,

6.5.72 `log_gamma`

6.5.72.1 Possible use:

- `log_gamma` (`float`) —> `float`

6.5.72.2 Result:

Returns the log of the value of the Gamma function at x.

6.5.73 lower_case

6.5.73.1 Possible use:

- **lower_case** (string) —> string

6.5.73.2 Result:

Converts all of the characters in the string operand to lower case

6.5.73.3 Examples:

```
string var0 <- lower_case("Abc"); // var0 equals 'abc'
```

6.5.73.4 See also:

upper_case,

6.5.74 main_connected_component

6.5.74.1 Possible use:

- **main_connected_component** (graph) —> graph

6.5.74.2 Result:

returns the sub-graph corresponding to the main connected components of the graph

6.5.74.3 Examples:

```
graph var0 <- main_connected_component(my_graph); // var0 equals the sub-graph corresponding to the main cor
```

6.5.74.4 See also:

connected_components_of,

6.5.75 map

6.5.75.1 Possible use:

- **map** (any) —> map

6.5.75.2 Result:

Casts the operand into the type map

6.5.76 masked_by

6.5.76.1 Possible use:

- geometry **masked_by** container<geometry> —> geometry
- **masked_by** (geometry , container<geometry>) —> geometry
- **masked_by** (geometry, container<geometry>, int) —> geometry

6.5.76.2 Examples:

```
geometry var0 <- perception_geom masked_by obstacle_list; // var0 equals the geometry representing the part
geometry var1 <- perception_geom masked_by obstacle_list; // var1 equals the geometry representing the part
```

6.5.77 material

6.5.77.1 Possible use:

- float **material** float —> msi.gama.util.GamaMaterial
- **material** (float , float) —> msi.gama.util.GamaMaterial

6.5.77.2 Result:

Returns

6.5.77.3 Examples:

6.5.77.4 See also:

,

6.5.78 material

6.5.78.1 Possible use:

- **material** (any) —> material

6.5.78.2 Result:

Casts the operand into the type material

6.5.79 `matrix`

6.5.79.1 Possible use:

- `matrix` (`any`) —> `matrix`

6.5.79.2 Result:

Casts the operand into the type matrix

6.5.80 `matrix_with`

6.5.80.1 Possible use:

- `point` `matrix_with` `any expression` —> `matrix`
- `matrix_with` (`point` , `any expression`) —> `matrix`

6.5.80.2 Result:

creates a matrix with a size provided by the first operand, and filled with the second operand

6.5.80.3 Comment:

Note that both components of the right operand point should be positive, otherwise an exception is raised.

6.5.80.4 See also:

matrix, as_matrix,

6.5.81 `max`

6.5.81.1 Possible use:

- `max` (`container`) —> `unknown`

6.5.81.2 Result:

the maximum element found in the operand

6.5.81.3 Comment:

the max operator behavior depends on the nature of the operand

6.5.81.4 Special cases:

- 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
- if it is a list of int of float, max returns the maximum of all the elements

```
unknown var0 <- max ([100, 23.2, 34.5]); // var0 equals 100.0
```

- 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.)

```
unknown var1 <- max([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]); // var1 equals {9.0,1.0}
```

6.5.81.5 See also:

min,

6.5.82 max_flow_between

6.5.82.1 Possible use:

- **max_flow_between** (graph, unknown, unknown) —> msi.gama.util.GamaMap<java.lang.Object,java.lang.Double>

6.5.82.2 Result:

The max flow (map in a graph between the source and the sink using Edmonds-Karp algorithm

6.5.82.3 Examples:

```
max_flow_between(my_graph, vertice1, vertice2)
```

6.5.83 max_of

6.5.83.1 Possible use:

- container **max_of** any expression —> unknown
- **max_of** (container , any expression) —> unknown

6.5.83.2 Result:

the maximum value of the right-hand expression evaluated on each of the elements of the left-hand operand

6.5.83.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

6.5.83.4 Special cases:

- As of GAMA 1.6, if the left-hand operand is nil or empty, max_of throws an error
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] max_of (each + 3); // var4 equals 9
```

6.5.83.5 Examples:

```
unknown var0 <- [1,2,4,3,5,7,6,8] max_of (each * 100 ); // var0 equals 800graph g2 <- as_edge_graph([{1,5}::
unknown var2 <- g2.vertices max_of (g2 degree_of( each )); // var2 equals 2
unknown var3 <- (list(node) max_of (round(node(each).location.x)); // var3 equals 96
```

6.5.83.6 See also:

min_of,

6.5.84 maximal_cliques_of

6.5.84.1 Possible use:

- **maximal_cliques_of** (graph) —> list<list>

6.5.84.2 Result:

returns the maximal cliques of a graph using the Bron-Kerbosch clique detection algorithm: A clique is maximal if it is impossible to enlarge it by adding another vertex from the graph. Note that a maximal clique is not necessarily the biggest clique in the graph.

6.5.84.3 Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- maximal_cliques_of (my_graph); // var1 equals the list of all the maximal cliques as list
```

6.5.84.4 See also:

biggest_cliques_of,

6.5.85 mean

6.5.85.1 Possible use:

- **mean** (container) —> unknown

6.5.85.2 Result:

the mean of all the elements of the operand

6.5.85.3 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.

6.5.85.4 Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

6.5.85.5 Examples:

```
unknown var0 <- mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 5.125
```

6.5.85.6 See also:

sum,

6.5.86 mean_deviation

6.5.86.1 Possible use:

- `mean_deviation` (`container`) —> `float`

6.5.86.2 Result:

the deviation from the mean of all the elements of the operand. See Mean_deviation for more details.

6.5.86.3 Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

6.5.86.4 Examples:

```
float var0 <- mean_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.125
```

6.5.86.5 See also:

mean, standard_deviation,

6.5.87 mean_of

6.5.87.1 Possible use:

- `container mean_of any expression` —> `unknown`
- `mean_of` (`container , any expression`) —> `unknown`

6.5.87.2 Result:

the mean of the right-hand expression evaluated on each of the elements of the left-hand operand

6.5.87.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

6.5.87.4 Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] mean_of (each); // var1 equals 4
```

6.5.87.5 Examples:

```
unknown var0 <- [1,2] mean_of (each * 10 ); // var0 equals 15
```

6.5.87.6 See also:

min_of, max_of, sum_of, product_of,

6.5.88 meanR

6.5.88.1 Possible use:

- **meanR** (container) —> unknown

6.5.88.2 Result:

returns the mean value of given vector (right-hand operand) in given variable (left-hand operand).

6.5.88.3 Examples:

```
list<int> X <- [2, 3, 1];
int var1 <- meanR(X); // var1 equals 2
```

6.5.89 median

6.5.89.1 Possible use:

- **median** (container) —> unknown

6.5.89.2 Result:

the median of all the elements of the operand.

6.5.89.3 Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

6.5.89.4 Examples:

```
unknown var0 <- median ([4.5, 3.5, 5.5, 3.4, 7.0]); // var0 equals 4.5
```

6.5.89.5 See also:

mean,

6.5.90 mental_state

6.5.90.1 Possible use:

- `mental_state` (`any`) —> `mental_state`

6.5.90.2 Result:

Casts the operand into the type mental_state

6.5.91 message

6.5.91.1 Possible use:

- `message` (`unknown`) —> `msi.gama.extensions.messaging.GamaMessage`

6.5.91.2 Result:

to be added

6.5.92 milliseconds_between

6.5.92.1 Possible use:

- `date milliseconds_between date` —> `float`
- `milliseconds_between` (`date , date`) —> `float`

6.5.92.2 Result:

Provide the exact number of milliseconds between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

6.5.92.3 Examples:

```
float var0 <- milliseconds_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 2.6784E9
```

6.5.93 `min`

6.5.93.1 Possible use:

- `min` (`container`) —> `unknown`

6.5.93.2 Result:

the minimum element found in the operand.

6.5.93.3 Comment:

the min operator behavior depends on the nature of the operand

6.5.93.4 Special cases:

- 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
- if it is a list of int or float: min returns the minimum of all the elements

```
unknown var0 <- min ([100, 23.2, 34.5]); // var0 equals 23.2
```

6.5.93.5 See also:

max,

6.5.94 min_of

6.5.94.1 Possible use:

- container **min_of** any expression —> unknown
- **min_of** (container , any expression) —> unknown

6.5.94.2 Result:

the minimum value of the right-hand expression evaluated on each of the elements of the left-hand operand

6.5.94.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

6.5.94.4 Special cases:

- if the left-hand operand is nil or empty, min_of throws an error
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] min_of (each + 3); // var4 equals 5
```

6.5.94.5 Examples:

```
unknown var0 <- [1,2,4,3,5,7,6,8] min_of (each * 100 ); // var0 equals 100graph g2 <- as_edge_graph([{1,5}::
unknown var2 <- g2 min_of (length(g2 out_edges_of each) ); // var2 equals 0
unknown var3 <- (list(node) min_of (round(node(each).location.x)); // var3 equals 4
```

6.5.94.6 See also:

max_of,

6.5.95 minus_days

6.5.95.1 Possible use:

- date **minus_days** int —> date
- **minus_days** (date , int) —> date

6.5.95.2 Result:

Subtract a given number of days from a date

6.5.95.3 Examples:

```
date var0 <- date('2000-01-01') minus_days 20; // var0 equals date('1999-12-12')
```

6.5.96 minus_hours

6.5.96.1 Possible use:

- `date` **`minus_hours`** `int` —> `date`
- **`minus_hours`** (`date` , `int`) —> `date`

6.5.96.2 Result:

Remove a given number of hours from a date

6.5.96.3 Examples:

```
// equivalent to date1 - 15 #h
date var1 <- date('2000-01-01') minus_hours 15 ; // var1 equals date('1999-12-31 09:00:00')
```

6.5.97 minus_minutes

6.5.97.1 Possible use:

- `date` **`minus_minutes`** `int` —> `date`
- **`minus_minutes`** (`date` , `int`) —> `date`

6.5.97.2 Result:

Subtract a given number of minutes from a date

6.5.97.3 Examples:

```
// date('2000-01-01') to date1 - 5#mn
date var1 <- date('2000-01-01') minus_minutes 5 ; // var1 equals date('1999-12-31 23:55:00')
```

6.5.98 minus_months

6.5.98.1 Possible use:

- `date` **`minus_months`** `int` —> `date`
- **`minus_months`** (`date` , `int`) —> `date`

6.5.98.2 Result:

Subtract a given number of months from a date

6.5.98.3 Examples:

```
date var0 <- date('2000-01-01') minus_months 5; // var0 equals date('1999-08-01')
```

6.5.99 minus_ms

6.5.99.1 Possible use:

- `date` **`minus_ms`** `int` —> `date`
- **`minus_ms`** (`date` , `int`) —> `date`

6.5.99.2 Result:

Remove a given number of milliseconds from a date

6.5.99.3 Examples:

```
// equivalent to date1 - 15 #ms
date var1 <- date('2000-01-01') minus_ms 1000 ; // var1 equals date('1999-12-31 23:59:59')
```

6.5.100 minus_seconds

Same signification as -

6.5.101 minus_weeks

6.5.101.1 Possible use:

- `date` **`minus_weeks`** `int` —> `date`
- **`minus_weeks`** (`date` , `int`) —> `date`

6.5.101.2 Result:

Subtract a given number of weeks from a date

6.5.101.3 Examples:

```
date var0 <- date('2000-01-01') minus_weeks 15; // var0 equals date('1999-09-18')
```

6.5.102 minus_years

6.5.102.1 Possible use:

- `date` **`minus_years`** `int` —> `date`
- **`minus_years`** (`date` , `int`) —> `date`

6.5.102.2 Result:

Subtract a given number of year from a date

6.5.102.3 Examples:

```
date var0 <- date('2000-01-01') minus_years 3; // var0 equals date('1997-01-01')
```

6.5.103 mod

6.5.103.1 Possible use:

- `int` **`mod`** `int` —> `int`
- **`mod`** (`int` , `int`) —> `int`

6.5.103.2 Result:

Returns the remainder of the integer division of the left-hand operand by the right-hand operand.

6.5.103.3 Special cases:

- if operands are float, they are truncated
- if the right-hand operand is equal to zero, raises an exception.

6.5.103.4 Examples:

```
int var0 <- 40 mod 3; // var0 equals 1
```

6.5.103.5 See also:

div,

6.5.104 moment

6.5.104.1 Possible use:

- **moment** (container, int, float) —> float

6.5.104.2 Result:

Returns the moment of k-th order with constant c of a data sequence

---

6.5.105 months_between

6.5.105.1 Possible use:

- date **months_between** date —> int
- **months_between** (date , date) —> int

6.5.105.2 Result:

Provide the exact number of months between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

6.5.105.3 Examples:

```
int var0 <- months_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 1
```

6.5.106 moran

6.5.106.1 Possible use:

- list<float> **moran** matrix<float> —> float
- **moran** (list<float> , matrix<float>) —> float

6.5.106.2 Special cases:

- return the Moran Index of the given list of interest points (list of floats) and the weight matrix (matrix of float)

```
float var0 <- moran([1.0, 0.5, 2.0], weight_matrix); // var0 equals the Moran index computed
```

6.5.107 `mul`

6.5.107.1 Possible use:

- `mul` (`container`) —> `unknown`

6.5.107.2 Result:

the product of all the elements of the operand

6.5.107.3 Comment:

the mul operator behavior depends on the nature of the operand

6.5.107.4 Special cases:

- 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
- if it is a list of int or float: mul returns the product of all the elements

```
unknown var0 <- mul ([100, 23.2, 34.5]); // var0 equals 80040.0
```

6.5.107.5 See also:

sum,

Chapter 7

Operators (N to R)

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

7.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

7.2

7.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

7.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

7.5 Operators

7.5.1 nb_cycles

7.5.1.1 Possible use:

- **nb_cycles** (graph) —> int

7.5.1.2 Result:

returns the maximum number of independent cycles in a graph. This number (u) is estimated through the number of nodes (v), links (e) and of sub-graphs (p): u = e - v + p.

7.5.1.3 Examples:

```
graph graphEpidemio <- graph([]);
int var1 <- nb_cycles(graphEpidemio); // var1 equals the number of cycles in the graph
```

7.5.1.4 See also:

alpha_index, beta_index, gamma_index, connectivity_index,

7.5.2 neighbors_at

7.5.2.1 Possible use:

- geometry **neighbors_at** float —> list
- **neighbors_at** (geometry , float) —> list

7.5.2.2 Result:

a list, containing all the agents of the same species than the left argument (if it is an agent) located at a distance inferior or equal to the right-hand operand to the left-hand operand (geometry, agent, point).

7.5.2.3 Comment:

The topology used to compute the neighborhood is the one of the left-operand if this one is an agent; otherwise the one of the agent applying the operator.

7.5.2.4 Examples:

```
list var0 <- (self neighbors_at (10)); // var0 equals all the agents located at a distance lower or equal to 1
```

7.5.2.5 See also:

neighbors_of, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to, at_distance,

---

7.5.3 neighbors_of

7.5.3.1 Possible use:

- graph **neighbors_of** unknown —> list
- **neighbors_of** (graph , unknown) —> list
- topology **neighbors_of** agent —> list
- **neighbors_of** (topology , agent) —> list
- **neighbors_of** (topology, geometry, float) —> list

7.5.3.2 Result:

a list, containing all the agents of the same species than the argument (if it is an agent) located at a distance inferior or equal to 1 to the right-hand operand agent considering the left-hand operand topology.

7.5.3.3 Special cases:

- a list, containing all the agents of the same species than the left argument (if it is an agent) located at a distance inferior or equal to the third argument to the second argument (agent, geometry or point) considering the first operand topology.

```
list var3 <- neighbors_of (topology(self), self,10); // var3 equals all the agents located at a distance lowe
```

7.5.3.4 Examples:

```
list var0 <- graphEpidemio neighbors_of (node(3)); // var0 equals [node0,node2]
list var1 <- graphFromMap neighbors_of node({12,45}); // var1 equals [{1.0,5.0},{34.0,56.0}]
list var2 <- topology(self) neighbors_of self; // var2 equals returns all the agents located at a distance lo
```

7.5.3.5 See also:

predecessors_of, successors_of, neighbors_at, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

7.5.4 new_emotion

7.5.4.1 Possible use:

- **new_emotion** (string) —> emotion
- string **new_emotion** predicate —> emotion
- **new_emotion** (string , predicate) —> emotion

- `string` **`new_emotion`** `float` —> `emotion`
- **`new_emotion`** (`string` , `float`) —> `emotion`
- `string` **`new_emotion`** `agent` —> `emotion`
- **`new_emotion`** (`string` , `agent`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `float`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `agent`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `predicate`) —> `emotion`
- **`new_emotion`** (`string`, `predicate`, `agent`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `float`, `agent`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `predicate`, `agent`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `predicate`, `float`) —> `emotion`
- **`new_emotion`** (`string`, `float`, `predicate`, `float`, `agent`) —> `emotion`

7.5.4.2 Result:

a new emotion with the given properties (name) a new emotion with the given properties (name,intensity,decay) a new emotion with the given properties (name,about) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name,intensity,about) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name, intensity) a new emotion with the given properties (name) a new emotion with the given properties (name)

7.5.4.3 Examples:

```
emotion("joy",12.3,eatFood,4) emotion("joy",12.3,4) emotion("joy",eatFood) emotion("joy",12.3,eatFood,4)
```

7.5.5 `new_folder`

7.5.5.1 Possible use:

- **`new_folder`** (`string`) —> `file`

7.5.5.2 Result:

opens an existing repository or create a new folder if it does not exist.

7.5.5.3 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.

7.5.5.4 Examples:

```
file dirNewT <- new_folder("incl/");   // dirNewT represents the repository "../incl/"
```

7.5.5.5 See also:

folder, file,

7.5.6 new_mental_state

7.5.6.1 Possible use:

- **new_mental_state** (string) —> mental_state
- string **new_mental_state** predicate —> mental_state
- **new_mental_state** (string , predicate) —> mental_state
- string **new_mental_state** emotion —> mental_state
- **new_mental_state** (string , emotion) —> mental_state
- string **new_mental_state** mental_state —> mental_state
- **new_mental_state** (string , mental_state) —> mental_state
- **new_mental_state** (string, predicate, int) —> mental_state
- **new_mental_state** (string, mental_state, agent) —> mental_state
- **new_mental_state** (string, emotion, agent) —> mental_state
- **new_mental_state** (string, emotion, float) —> mental_state
- **new_mental_state** (string, mental_state, float) —> mental_state
- **new_mental_state** (string, predicate, agent) —> mental_state
- **new_mental_state** (string, mental_state, int) —> mental_state
- **new_mental_state** (string, emotion, int) —> mental_state
- **new_mental_state** (string, predicate, float) —> mental_state
- **new_mental_state** (string, predicate, int, agent) —> mental_state
- **new_mental_state** (string, emotion, float, int) —> mental_state
- **new_mental_state** (string, predicate, float, int) —> mental_state
- **new_mental_state** (string, mental_state, float, int) —> mental_state
- **new_mental_state** (string, mental_state, float, agent) —> mental_state
- **new_mental_state** (string, emotion, int, agent) —> mental_state
- **new_mental_state** (string, mental_state, int, agent) —> mental_state
- **new_mental_state** (string, emotion, float, agent) —> mental_state
- **new_mental_state** (string, predicate, float, agent) —> mental_state
- **new_mental_state** (string, emotion, float, int, agent) —> mental_state
- **new_mental_state** (string, mental_state, float, int, agent) —> mental_state
- **new_mental_state** (string, predicate, float, int, agent) —> mental_state

7.5.6.2 Result:

a new mental state a new mental state

7.5.6.3 Examples:

```
new_mental-state(belief) new_mental-state(belief) new_mental-state(belief) new_mental-state(belief) new_m
```

7.5.7 new_predicate

7.5.7.1 Possible use:

- **new_predicate** (string) —> predicate
- string **new_predicate** int —> predicate
- **new_predicate** (string , int) —> predicate
- string **new_predicate** map —> predicate
- **new_predicate** (string , map) —> predicate
- string **new_predicate** bool —> predicate
- **new_predicate** (string , bool) —> predicate
- string **new_predicate** agent —> predicate
- **new_predicate** (string , agent) —> predicate
- **new_predicate** (string, map, int) —> predicate
- **new_predicate** (string, map, agent) —> predicate
- **new_predicate** (string, map, bool) —> predicate
- **new_predicate** (string, map, int, bool) —> predicate
- **new_predicate** (string, map, int, agent) —> predicate
- **new_predicate** (string, map, bool, agent) —> predicate
- **new_predicate** (string, map, int, bool, agent) —> predicate

7.5.7.2 Result:

a new predicate with the given properties (name) a new predicate with the given is_true (name, lifetime) a new predicate with the given properties (name, values, lifetime) a new predicate with the given properties (name, values, agentCause) a new predicate with the given properties (name, values, lifetime, is_true) a new predicate with the given properties (name, values, is_true) a new predicate with the given properties (name, values) a new predicate with the given properties (name, values, lifetime, agentCause) a new predicate with the given properties (name, values, lifetime, is_true, agentCause) a new predicate with the given properties (name, values, is_true, agentCause) a new predicate with the given is_true (name, is_true) a new predicate with the given properties (name, values, lifetime)

7.5.7.3 Examples:

```
predicate("people to meet") predicate("hasWater", 10 predicate("people to meet", ["time"::10], true) predi
```

7.5.8 new_social_link

7.5.8.1 Possible use:

- **new_social_link** (agent) —> msi.gaml.architecture.simplebdi.SocialLink
- **new_social_link** (agent, float, float, float, float) —> msi.gaml.architecture.simplebdi.SocialLink

7.5.8.2 Result:

a new social link a new social link

7.5.8.3 Examples:

```
new_social_link(agentA) new_social_link(agentA,0.0,-0.1,0.2,0.1)
```

7.5.9 node

7.5.9.1 Possible use:

- **node** (unknown) —> unknown
- unknown **node** float —> unknown
- **node** (unknown , float) —> unknown

7.5.10 nodes

7.5.10.1 Possible use:

- **nodes** (container) —> container

7.5.11 norm

7.5.11.1 Possible use:

- **norm** (point) —> float

7.5.11.2 Result:

the norm of the vector with the coordinates of the point operand.

7.5.11.3 Examples:

```
float var0 <- norm({3,4}); // var0 equals 5.0
```

7.5.12 Norm

7.5.12.1 Possible use:

- Norm (any) —> Norm

7.5.12.2 Result:

Casts the operand into the type Norm

7.5.13 normal_area

7.5.13.1 Possible use:

- **normal_area** (float, float, float) —> float

7.5.13.2 Result:

Returns the area to the left of x in the normal distribution with the given mean and standard deviation.

7.5.14 normal_density

7.5.14.1 Possible use:

- **normal_density** (float, float, float) —> float

7.5.14.2 Result:

Returns the probability of x in the normal distribution with the given mean and standard deviation.

7.5.15 normal_inverse

7.5.15.1 Possible use:

- **normal_inverse** (float, float, float) —> float

7.5.15.2 Result:

Returns the x in the normal distribution with the given mean and standard deviation, to the left of which lies the given area. normal.Inverse returns the value in terms of standard deviations from the mean, so we need to adjust it for the given mean and standard deviation.

7.5.16 not

Same signification as !

7.5.17 obj_file

7.5.17.1 Possible use:

- **obj_file** (string) —> file

7.5.17.2 Result:

Constructs a file of type obj. Allowed extensions are limited to obj, OBJ

7.5.18 of

Same signification as .

7.5.19 of_generic_species

7.5.19.1 Possible use:

- container **of_generic_species** species —> list
- **of_generic_species** (container , species) —> list

7.5.19.2 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

7.5.19.3 Examples:

```
// species test {} // species sous_test parent: test {}
list var2 <- [sous_test(0),sous_test(1),test(2),test(3)] of_generic_species test; // var2 equals [sous_tes
list var3 <- [sous_test(0),sous_test(1),test(2),test(3)] of_generic_species sous_test; // var3 equals [sou
list var4 <- [sous_test(0),sous_test(1),test(2),test(3)] of_species test; // var4 equals [test2,test3]
list var5 <- [sous_test(0),sous_test(1),test(2),test(3)] of_species sous_test; // var5 equals [sous_test0,
```

7.5.19.4 See also:

of_species,

7.5.20 of_species

7.5.20.1 Possible use:

- container **of_species** species —> list
- **of_species** (container , species) —> list

7.5.20.2 Result:

a list, containing the agents of the left-hand operand whose species is the one 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 explicitly if it is to be used in an ask statement, for instance).

7.5.20.3 Special cases:

- if the right operand is nil, of_species returns the right operand

7.5.20.4 Examples:

```
list var0 <- (self neighbors_at 10) of_species (species (self)); // var0 equals all the neighboring agents of
list var1 <- [test(0),test(1),node(1),node(2)] of_species test; // var1 equals [test0,test1]
```

7.5.20.5 See also:

of_generic_species,

7.5.21 one_of

7.5.21.1 Possible use:

- `one_of` (`container<KeyType,ValueType>`) —> `ValueType`

7.5.21.2 Result:

one of the values stored in this container at a random key

7.5.21.3 Comment:

the one_of operator behavior depends on the nature of the operand

7.5.21.4 Special cases:

- if it is a graph, one_of returns one of the lists of edges
- if it is a file, one_of returns one of the elements of the content of the file (that is also a container)
- if the operand is empty, one_of returns nil
- if it is a list or a matrix, one_of returns one of the values of the list or of the matrix

```
int
i <- any ([1,2,3]); //i equals 1, 2 or 3string sMat <- one_of(matrix([["c11","c12","c13"],["c21","c22","c23
```

- if it is a map, one_of returns one the value of a random pair of the map

```
int im <- one_of ([2::3, 4::5, 6::7]);  // im equals 3, 5 or 7
bool var6 <- [2::3, 4::5, 6::7].values contains im; // var6 equals true
```

- if it is a population, one_of returns one of the agents of the population

```
bug b <- one_of(bug);  // Given a previously defined species bug, b is one of the created bugs, e.g. bug3
```

7.5.21.5 See also:

contains,

7.5.22 open_simplex_generator

7.5.22.1 Possible use:

- `open_simplex_generator` (`float`, `float`, `float`) —> `float`

7.5.22.2 Result:

take a x, y and a bias parameters and gives a value

7.5.22.3 Examples:

```
float var0 <- open_simplex_generator(2,3,253); // var0 equals 10.2
```

7.5.23 or

7.5.23.1 Possible use:

- `bool` **`or`** `any expression` —> `bool`
- **`or`** (`bool` , `any expression`) —> `bool`

7.5.23.2 Result:

a bool value, equal to the logical or between the left-hand operand and the right-hand operand.

7.5.23.3 Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like 1 or 0 is accepted and returns true.

7.5.23.4 See also:

bool, and, !,

7.5.24 or

7.5.24.1 Possible use:

- `predicate` **`or`** `predicate` —> `predicate`
- **`or`** (`predicate` , `predicate`) —> `predicate`

7.5.24.2 Result:

create a new predicate from two others by including them as subintentions. It's an exclusive "or"

7.5.24.3 Examples:

```
predicate1 or predicate2
```

7.5.25 osm_file

7.5.25.1 Possible use:

- `string` **`osm_file`** `map<string,list>` —> `file`
- **`osm_file`** (`string` , `map<string,list>`) —> `file`
- **`osm_file`** (`string`, `map<string,list>`, `int`) —> `file`

7.5.25.2 Result:

opens a file that a is a kind of OSM file with some filtering, forcing the initial CRS to be the one indicated by the second int parameter (see http://spatialreference.org/ref/epsg/). If this int parameter is equal to 0, the data is considered as already projected. opens a file that a is a kind of OSM file with some filtering.

7.5.25.3 Comment:

The file should have a OSM file extension, cf. file type definition for supported file extensions.The file should have a OSM file extension, cf. file type definition for supported file extensions.

7.5.25.4 Special cases:

- If the specified string does not refer to an existing OSM file, an exception is risen.
- If the specified string does not refer to an existing OSM file, an exception is risen.

7.5.25.5 Examples:

```
file myOSMfile2 <- osm_file("../includes/rouen.osm",["highway"::["primary","motorway"]], 0); file myOSMfil
```

7.5.25.6 See also:

file,

7.5.26 out_degree_of

7.5.26.1 Possible use:

- `graph out_degree_of unknown —> int`
- `out_degree_of (graph , unknown) —> int`

7.5.26.2 Result:

returns the out degree of a vertex (right-hand operand) in the graph given as left-hand operand.

7.5.26.3 Examples:

```
int var1 <- graphFromMap out_degree_of (node(3)); // var1 equals 4
```

7.5.26.4 See also:

in_degree_of, degree_of,

7.5.27 out_edges_of

7.5.27.1 Possible use:

- `graph out_edges_of unknown —> list`
- `out_edges_of (graph , unknown) —> list`

7.5.27.2 Result:

returns the list of the out-edges of a vertex (right-hand operand) in the graph given as left-hand operand.

7.5.27.3 Examples:

```
list var1 <- graphFromMap out_edges_of (node(3)); // var1 equals 3
```

7.5.27.4 See also:

in_edges_of,

7.5.28 overlapping

7.5.28.1 Possible use:

- container<agent> **overlapping** geometry —> list<geometry>
- **overlapping** (container<agent> , geometry) —> list<geometry>

7.5.28.2 Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), overlapping the operand (casted as a geometry).

7.5.28.3 Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] overlapping(self); // var0 equals return the agents among ag1, ag2 and
```

7.5.28.4 See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, agents_overlapping,

---

7.5.29 overlaps

7.5.29.1 Possible use:

- geometry **overlaps** geometry —> bool
- **overlaps** (geometry , geometry) —> bool

7.5.29.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) overlaps the right-geometry (or agent/point).

7.5.29.3 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

7.5.29.4 Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) overlaps polyline([{15,15},{25,25}]); // var0 equals true
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},{25,25},{25,15}
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps {25,25}; // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{35,35},{35,45},{45,45},{45,35}
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polyline([{10,10},{20,20}]); // var4 equa
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps {15,15}; // var5 equals true
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{0,0},{0,30},{30,30}, {30,0}]);
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},{25,25},{25,15}
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{10,20},{20,20},{20,30},{10,30}
```

7.5.29.5 See also:

disjoint_from, crosses, intersects, partially_overlaps, touches,

---

7.5.30 pair

7.5.30.1 Possible use:

- **pair** (any) —> pair

7.5.30.2 Result:

Casts the operand into the type pair

7.5.31 partially_overlaps

7.5.31.1 Possible use:

- geometry **partially_overlaps** geometry —> bool
- **partially_overlaps** (geometry , geometry) —> bool

7.5.31.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) partially overlaps the right-geometry (or agent/point).

7.5.31.3 Comment:

if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).

7.5.31.4 Special cases:

- if one of the operand is null, returns false.

7.5.31.5 Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) partially_overlaps polyline([{15,15},{25,25}]); // var0 equals tru
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{15,15},{15,25},{25,2
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps {25,25}; // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{35,35},{35,45},{45,4
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polyline([{10,10},{20,20}]); /
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps {15,15}; // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{0,0},{0,30},{30,30},
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{15,15},{15,25},{25,2
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{10,20},{20,20},{20,3
```

7.5.31.6 See also:

disjoint_from, crosses, overlaps, intersects, touches,

7.5.32 path

7.5.32.1 Possible use:

- **path** (any) —> path

7.5.32.2 Result:

Casts the operand into the type path

7.5.33 path_between

7.5.33.1 Possible use:

- list<agent> **path_between** container<geometry> —> path
- **path_between** (list<agent> , container<geometry>) —> path
- topology **path_between** container<geometry> —> path
- **path_between** (topology , container<geometry>) —> path
- msi.gama.util.GamaMap<msi.gama.metamodel.agent.IAgent,java.lang.Object> **path_between** container<geometry> —> path
- **path_between** (msi.gama.util.GamaMap<msi.gama.metamodel.agent.IAgent,java.lang.Object> , container<geometry>) —> path
- **path_between** (topology, geometry, geometry) —> path
- **path_between** (msi.gama.util.GamaMap<msi.gama.metamodel.agent.IAgent,java.lang.Object>, geometry, geometry) —> path
- **path_between** (list<agent>, geometry, geometry) —> path

- **path_between** (graph, geometry, geometry) —> path

7.5.33.2 Result:

The shortest path between several objects according to set of cells The shortest path between two objects according to set of cells with corresponding weights The shortest path between two objects according to set of cells The shortest path between several objects according to set of cells with corresponding weights The shortest path between a list of two objects in a graph

7.5.33.3 Examples:

```
path var0 <- path_between (cell_grid where each.is_free, [ag1, ag2, ag3]); // var0 equals A path between ag1
path var1 <- my_topology path_between (ag1, ag2); // var1 equals A path between ag1 and ag2
path var2 <- path_between (cell_grid as_map (each::each.is_obstacle ? 9999.0 : 1.0), ag1, ag2); // var2 equal
path var3 <- path_between (cell_grid where each.is_free, ag1, ag2); // var3 equals A path between ag1 and ag2
path var4 <- my_topology path_between [ag1, ag2]; // var4 equals A path between ag1 and ag2
path var5 <- path_between (cell_grid as_map (each::each.is_obstacle ? 9999.0 : 1.0), [ag1, ag2, ag3]); // va
path var6 <- path_between (my_graph, ag1, ag2); // var6 equals A path between ag1 and ag2
```

7.5.33.4 See also:

towards, direction_to, distance_between, direction_between, path_to, distance_to,

---

7.5.34 path_to

7.5.34.1 Possible use:

- geometry **path_to** geometry —> path
- **path_to** (geometry , geometry) —> path
- point **path_to** point —> path
- **path_to** (point , point) —> path

7.5.34.2 Result:

A path between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

7.5.34.3 Examples:

```
path var0 <- ag1 path_to ag2; // var0 equals the path between ag1 and ag2 considering the topology of the agen
```

7.5.34.4 See also:

towards, direction_to, distance_between, direction_between, path_between, distance_to,

---

7.5.35 paths_between

7.5.35.1 Possible use:

- **paths_between** (graph, pair, int) —> msi.gama.util.IList<msi.gama.util.path.GamaSpatialPath>

7.5.35.2 Result:

The K shortest paths between a list of two objects in a graph

7.5.35.3 Examples:

```
msi.gama.util.IList<msi.gama.util.path.GamaSpatialPath> var0 <- paths_between(my_graph, ag1:: ag2, 2); //
```

7.5.36 pbinom

Same signification as binomial_sum

7.5.37 pchisq

Same signification as chi_square

7.5.38 percent_absolute_deviation

7.5.38.1 Possible use:

- list<float> **percent_absolute_deviation** list<float> —> float
- **percent_absolute_deviation** (list<float> , list<float>) —> float

7.5.38.2 Result:

percent absolute deviation indicator for 2 series of values: percent_absolute_deviation(list_vals_observe,list_vals_sim)

7.5.38.3 Examples:

```
percent_absolute_deviation([200,300,150,150,200],[250,250,100,200,200])
```

7.5.39 percentile

Same signification as quantile_inverse

7.5.40 pgamma

Same signification as gamma_distribution

7.5.41 pgm_file

7.5.41.1 Possible use:

- `pgm_file` (`string`) —> `file`

7.5.41.2 Result:

Constructs a file of type pgm. Allowed extensions are limited to pgm

---

7.5.42 plan

7.5.42.1 Possible use:

- `container<geometry> plan float` —> `geometry`
- `plan` (`container<geometry>` , `float`) —> `geometry`

7.5.42.2 Result:

A polyline geometry from the given list of points.

7.5.42.3 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.

7.5.42.4 Examples:

```
geometry var0 <- polyplan([{0,0}, {0,10}, {10,10}, {10,0}],10); // var0 equals a polyline geometry composed
```

7.5.42.5 See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle,

---

7.5.43 plus_days

7.5.43.1 Possible use:

- `date` **`plus_days`** `int` —> `date`
- **`plus_days`** (`date` , `int`) —> `date`

7.5.43.2 Result:

Add a given number of days to a date

7.5.43.3 Examples:

```
date var0 <- date('2000-01-01') plus_days 12; // var0 equals date('2000-01-13')
```

7.5.44 plus_hours

7.5.44.1 Possible use:

- `date` **`plus_hours`** `int` —> `date`
- **`plus_hours`** (`date` , `int`) —> `date`

7.5.44.2 Result:

Add a given number of hours to a date

7.5.44.3 Examples:

```
// equivalent to date1 + 15 #h
date var1 <- date('2000-01-01') plus_hours 24; // var1 equals date('2000-01-02')
```

7.5.45 plus_minutes

7.5.45.1 Possible use:

- `date` **`plus_minutes`** `int` —> `date`
- **`plus_minutes`** (`date` , `int`) —> `date`

7.5.45.2 Result:

Add a given number of minutes to a date

7.5.45.3 Examples:

```
// equivalent to date1 + 5 #mn
date var1 <- date('2000-01-01') plus_minutes 5 ; // var1 equals date('2000-01-01 00:05:00')
```

7.5.46 plus_months

7.5.46.1 Possible use:

- `date` **`plus_months`** `int` —> `date`
- **`plus_months`** (`date` , `int`) —> `date`

7.5.46.2 Result:

Add a given number of months to a date

7.5.46.3 Examples:

```
date var0 <- date('2000-01-01') plus_months 5; // var0 equals date('2000-06-01')
```

7.5.47 plus_ms

7.5.47.1 Possible use:

- `date` **`plus_ms`** `int` —> `date`
- **`plus_ms`** (`date` , `int`) —> `date`

7.5.47.2 Result:

Add a given number of milliseconds to a date

7.5.47.3 Examples:

```
// equivalent to date('2000-01-01') + 15 #ms
date var1 <- date('2000-01-01') plus_ms 1000 ; // var1 equals date('2000-01-01 00:00:01')
```

7.5.48 plus_seconds

Same signification as +

7.5.49 plus_weeks

7.5.49.1 Possible use:

- `date` **`plus_weeks`** `int` —> `date`
- **`plus_weeks`** (`date` , `int`) —> `date`

7.5.49.2 Result:

Add a given number of weeks to a date

7.5.49.3 Examples:

```
date var0 <- date('2000-01-01') plus_weeks 15; // var0 equals date('2000-04-15')
```

7.5.50 plus_years

7.5.50.1 Possible use:

- `date` **`plus_years`** `int` —> `date`
- **`plus_years`** (`date` , `int`) —> `date`

7.5.50.2 Result:

Add a given number of years to a date

7.5.50.3 Examples:

```
date var0 <- date('2000-01-01') plus_years 15; // var0 equals date('2015-01-01')
```

7.5.51 pnorm

Same signification as normal_area

7.5.52 point

7.5.52.1 Possible use:

- `float point float —> point`
- `point (float , float) —> point`
- `int point int —> point`
- `point (int , int) —> point`
- `float point int —> point`
- `point (float , int) —> point`
- `int point float —> point`
- `point (int , float) —> point`
- `point (float, int, float) —> point`
- `point (float, float, float) —> point`
- `point (int, int, int) —> point`
- `point (int, float, float) —> point`
- `point (int, int, float) —> point`
- `point (float, float, int) —> point`
- `point (float, int, int) —> point`

7.5.52.2 Result:

internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead.

7.5.53 points_along

7.5.53.1 Possible use:

- `geometry points_along list<float> —> list`
- `points_along (geometry , list<float>) —> list`

7.5.53.2 Result:

A list of points along the operand-geometry given its location in terms of rate of distance from the starting points of the geometry.

7.5.53.3 Examples:

```
list var0 <- line([{10,10},{80,80}]) points_along ([0.3, 0.5, 0.9]); // var0 equals the list of following po
```

7.5.53.4 See also:

closest_points_with, farthest_point_to, points_at, points_on,

---

7.5.54 points_at

7.5.54.1 Possible use:

- `int points_at float —> list<point>`
- `points_at (int , float) —> list<point>`

7.5.54.2 Result:

A list of left-operand number of points located at a the right-operand distance to the agent location.

7.5.54.3 Examples:

```
list<point> var0 <- 3 points_at(20.0); // var0 equals returns [pt1, pt2, pt3] with pt1, pt2 and pt3 located a
```

7.5.54.4 See also:

any_location_in, any_point_in, closest_points_with, farthest_point_to,

---

7.5.55 points_on

7.5.55.1 Possible use:

- `geometry points_on float —> list`
- `points_on (geometry , float) —> list`

7.5.55.2 Result:

A list of points of the operand-geometry distant from each other to the float right-operand .

7.5.55.3 Examples:

```
list var0 <- square(5) points_on(2); // var0 equals a list of points belonging to the exterior ring of the sq
```

7.5.55.4 See also:

closest_points_with, farthest_point_to, points_at,

7.5.56 poisson

7.5.56.1 Possible use:

- `poisson` (`float`) —> `int`

7.5.56.2 Result:

A value from a random variable following a Poisson distribution (with the positive expected number of occurence lambda as operand).

7.5.56.3 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.

7.5.56.4 Examples:

```
int var0 <- poisson(3.5); // var0 equals a random positive integer
```

7.5.56.5 See also:

binomial, gauss,

7.5.57 polygon

7.5.57.1 Possible use:

- `polygon` (`container<agent>`) —> `geometry`

7.5.57.2 Result:

A polygon geometry from the given list of points.

7.5.57.3 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.

7.5.57.4 Examples:

```
geometry var0 <- polygon([{0,0}, {0,10}, {10,10}, {10,0}]); // var0 equals a polygon geometry composed of th
```

7.5.57.5 See also:

around, circle, cone, line, link, norm, point, polyline, rectangle, square, triangle,

---

7.5.58 polyhedron

7.5.58.1 Possible use:

- `container<geometry>` **`polyhedron`** `float` —> `geometry`
- **`polyhedron`** (`container<geometry>` , `float`) —> `geometry`

7.5.58.2 Result:

A polyhedron geometry from the given list of points.

7.5.58.3 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.

7.5.58.4 Examples:

```
geometry var0 <- polyhedron([{0,0}, {0,10}, {10,10}, {10,0}],10); // var0 equals a polygon geometry composed
```

7.5.58.5 See also:

around, circle, cone, line, link, norm, point, polyline, rectangle, square, triangle,

---

7.5.59 polyline

Same signification as line

7.5.60 polyplan

Same signification as plan

7.5.61 predecessors_of

7.5.61.1 Possible use:

- graph **predecessors_of** unknown —> list
- **predecessors_of** (graph , unknown) —> list

7.5.61.2 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)

7.5.61.3 Examples:

```
list var1 <- graphEpidemio predecessors_of ({1,5}); // var1 equals []
list var2 <- graphEpidemio predecessors_of node({34,56}); // var2 equals [{12;45}]
```

7.5.61.4 See also:

neighbors_of, successors_of,

7.5.62 predicate

7.5.62.1 Possible use:

- **predicate** (any) —> predicate

7.5.62.2 Result:

Casts the operand into the type predicate

7.5.63 predict

7.5.63.1 Possible use:

- regression **predict** list<float> —> float
- **predict** (regression , list<float>) —> float

7.5.63.2 Result:

returns the value predict by the regression parameters for a given instance. Usage: predict(regression, instance)

7.5.63.3 Examples:

```
predict(my_regression, [1,2,3])
```

7.5.64 product

Same signification as mul

7.5.65 product_of

7.5.65.1 Possible use:

- `container` **`product_of`** `any expression` —> `unknown`
- **`product_of`** (`container` , `any expression`) —> `unknown`

7.5.65.2 Result:

the product of the right-hand expression evaluated on each of the elements of the left-hand operand

7.5.65.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

7.5.65.4 Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] product_of (each); // var1 equals 48
```

7.5.65.5 Examples:

```
unknown var0 <- [1,2] product_of (each * 10 ); // var0 equals 200
```

7.5.65.6 See also:

min_of, max_of, sum_of, mean_of,

7.5.66 promethee_DM

7.5.66.1 Possible use:

- msi.gama.util.IList<java.util.List> **promethee_DM** msi.gama.util.IList<java.util.Map<java.lang.String, —> int
- **promethee_DM** (msi.gama.util.IList<java.util.List> , msi.gama.util.IList<java.util.Map<java.lang.Strin —> int

7.5.66.2 Result:

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 [http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6VCT-4VF56TV-1&_user=10&_coverDate=01%2F01%2F2010&_rdoc=1&_fmt=high&_orig=search&_sort=d&_docanchor=&view=c&_searchStrId=1389284642&_rerunOrigin=google&_acct=C000050 Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research(2009)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: 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.

7.5.66.3 Special cases:

- returns -1 is the list of candidates is nil or empty

7.5.66.4 Examples:

```
int var0 <- promethee_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"utility", "weight" :: 2.0,"p"::0.5,
```

7.5.66.5 See also:

weighted_means_DM, electre_DM, evidence_theory_DM,

7.5.67 property_file

7.5.67.1 Possible use:

- **property_file** (string) —> file

7.5.67.2 Result:

Constructs a file of type property. Allowed extensions are limited to properties

---

7.5.68 pValue_for_fStat

7.5.68.1 Possible use:

- **pValue_for_fStat** (float, int, int) —> float

7.5.68.2 Result:

Returns the P value of F statistic fstat with numerator degrees of freedom dfn and denominator degress of freedom dfd. Uses the incomplete Beta function.

7.5.69 pValue_for_tStat

7.5.69.1 Possible use:

- float **pValue_for_tStat** int —> float
- **pValue_for_tStat** (float , int) —> float

7.5.69.2 Result:

Returns the P value of the T statistic tstat with df degrees of freedom. This is a two-tailed test so we just double the right tail which is given by studentT of -|tstat|.

7.5.70 pyramid

7.5.70.1 Possible use:

- **pyramid** (float) —> geometry

7.5.70.2 Result:

A square geometry which side size is given by the operand.

7.5.70.3 Comment:

the center of the pyramid is by default the location of the current agent in which has been called this operator.

7.5.70.4 Special cases:

- returns nil if the operand is nil.

7.5.70.5 Examples:

```
geometry var0 <- pyramid(5); // var0 equals a geometry as a square with side_size = 5.
```

7.5.70.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, square,

---

7.5.71 quantile

7.5.71.1 Possible use:

- `container` **`quantile`** `float` —> `float`
- **`quantile`** (`container` , `float`) —> `float`

7.5.71.2 Result:

Returns the phi-quantile; that is, an element elem for which holds that phi percent of data elements are less than elem. The quantile need not necessarily be contained in the data sequence, it can be a linear interpolation.

7.5.72 quantile_inverse

7.5.72.1 Possible use:

- `container` **`quantile_inverse`** `float` —> `float`
- **`quantile_inverse`** (`container` , `float`) —> `float`

7.5.72.2 Result:

Returns how many percent of the elements contained in the receiver are <= element. Does linear interpolation if the element is not contained but lies in between two contained elements.

7.5.73 R_correlation

Same signification as corR

7.5.74 R_file

7.5.74.1 Possible use:

- **`R_file`** (`string`) —> `file`

7.5.74.2 Result:

Constructs a file of type R. Allowed extensions are limited to r

---

7.5.75 R_mean

Same signification as meanR

7.5.76 range

7.5.76.1 Possible use:

- **range** (int) —> list
- int **range** int —> list
- **range** (int , int) —> list
- **range** (int, int, int) —> list

7.5.76.2 Result:

Allows to build a list of int representing all contiguous values from the first to the second argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value Allows to build a list of int representing all contiguous values from zero to the argument. The range can be increasing or decreasing. Passing 0 will return a singleton list with 0 Allows to build a list of int representing all contiguous values from the first to the second argument, using the step represented by the third argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value. Passing a step of 0 will result in an exception. Attempting to build infinite ranges (e.g. end > start with a negative step) will similarly not be accepted and yield an exception

---

7.5.77 rank_interpolated

7.5.77.1 Possible use:

- container **rank_interpolated** float —> float
- **rank_interpolated** (container , float) —> float

7.5.77.2 Result:

Returns the linearly interpolated number of elements in a list less or equal to a given element. The rank is the number of elements <= element. Ranks are of the form {0, 1, 2,..., sortedList.size()}. If no element is <= element, then the rank is zero. If the element lies in between two contained elements, then linear interpolation is used and a non integer value is returned.

7.5.78 read

7.5.78.1 Possible use:

- **read** (string) —> unknown

7.5.78.2 Result:

Reads an attribute of the agent. The attribute's name is specified by the operand.

7.5.78.3 Examples:

```
unknown
agent_name <- read ('name'); //agent_name equals reads the 'name' variable of agent then assigns the returne
```

7.5.79 rectangle

7.5.79.1 Possible use:

- **rectangle** (point) —> geometry
- float **rectangle** float —> geometry
- **rectangle** (float , float) —> geometry
- point **rectangle** point —> geometry
- **rectangle** (point , point) —> geometry

7.5.79.2 Result:

A rectangle geometry which side sizes are given by the operands.

7.5.79.3 Comment:

the center of the rectangle is by default the location of the current agent in which has been called this operator.the center of the rectangle is by default the location of the current agent in which has been called this operator.

7.5.79.4 Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.
- returns nil if the operand is nil.

7.5.79.5 Examples:

```
geometry var0 <- rectangle(10, 5); // var0 equals a geometry as a rectangle with width = 10 and height = 5.
geometry var1 <- rectangle({10, 5}); // var1 equals a geometry as a rectangle with width = 10 and height = 5.
geometry var2 <- rectangle({2.0,6.0}, {6.0,20.0}); // var2 equals a geometry as a rectangle with {2.0,6.0} as
```

7.5.79.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, square, triangle,

---

7.5.80 reduced_by

Same signification as -

7.5.81 regression

7.5.81.1 Possible use:

- **regression** (any) —> regression

7.5.81.2 Result:

Casts the operand into the type regression

7.5.82 remove_duplicates

Same signification as distinct

7.5.83 remove_node_from

7.5.83.1 Possible use:

- geometry **remove_node_from** graph —> graph
- **remove_node_from** (geometry , graph) —> graph

7.5.83.2 Result:

removes a node from a graph.

7.5.83.3 Comment:

all the edges containing this node are also removed.

7.5.83.4 Examples:

```
graph var0 <- node(0) remove_node_from graphEpidemio; // var0 equals the graph without node(0)
```

7.5.84 replace

7.5.84.1 Possible use:

- **replace** (string, string, string) —> string

7.5.84.2 Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding the second operand by the third operand

7.5.84.3 Examples:

```
string var0 <- replace('to be or not to be,that is the question','to', 'do'); // var0 equals 'do be or not do b
```

7.5.84.4 See also:

replace_regex,

7.5.85 replace_regex

7.5.85.1 Possible use:

- **replace_regex** (string, string, string) —> string

7.5.85.2 Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding to the regular expression given in the second operand by the third operand

7.5.85.3 Examples:

```
string var0 <- replace_regex("colour, color", "colou?r", "col"); // var0 equals 'col, col'
```

7.5.85.4 See also:

replace,

7.5.86 reverse

7.5.86.1 Possible use:

- **reverse** (msi.gama.util.GamaMap<K,V>) —> container
- **reverse** (container<KeyType,ValueType>) —> container
- **reverse** (string) —> string

7.5.86.2 Result:

the operand elements in the reversed order in a copy of the operand.

7.5.86.3 Comment:

the reverse operator behavior depends on the nature of the operand

7.5.86.4 Special cases:

- 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 list, reverse returns a copy of the operand list with elements in the reversed order

```
container var0 <- reverse ([10,12,14]); // var0 equals [14, 12, 10]
```

- 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)

```
map<int,string> var1 <- reverse (['k1'::44, 'k2'::32, 'k3'::12]); // var1 equals [44::'k1', 32::'k2', 12::'
```

- if it is a matrix, reverse returns a new matrix containing the transpose of the operand.

```
container var2 <- reverse(matrix([["c11","c12","c13"],["c21","c22","c23"]])); // var2 equals matrix([["c11
```

- if it is a string, reverse returns a new string with characters in the reversed order

```
string var3 <- reverse ('abcd'); // var3 equals 'dcba'
```

7.5.87 rewire_n

7.5.87.1 Possible use:

- graph **rewire_n** int —> graph
- **rewire_n** (graph , int) —> graph

7.5.87.2 Result:

rewires the given count of edges.

7.5.87.3 Comment:

If there are too many edges, all the edges will be rewired.

7.5.87.4 Examples:

```
graph var1 <- graphEpidemio rewire_n 10; // var1 equals the graph with 3 edges rewired
```

7.5.88 rgb

7.5.88.1 Possible use:

- rgb **rgb** int —> rgb
- **rgb** (rgb , int) —> rgb
- rgb **rgb** float —> rgb
- **rgb** (rgb , float) —> rgb
- string **rgb** int —> rgb
- **rgb** (string , int) —> rgb
- **rgb** (int, int, int) —> rgb
- **rgb** (int, int, int, float) —> rgb
- **rgb** (int, int, int, int) —> rgb

7.5.88.2 Result:

Returns a color defined by red, green, blue components and an alpha blending value.

7.5.88.3 Special cases:

- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0.0 and 1.0)
- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0 and 255)
- It can be used with r=red, g=green, b=blue, each between 0 and 255
- It can be used with a color and an alpha between 0 and 255

- It can be used with a color and an alpha between 0 and 1
- It can be used with a name of color and alpha (between 0 and 255)

7.5.88.4 Examples:

```
rgb var0 <- rgb (255,0,0,0.5); // var0 equals a light red color
rgb var1 <- rgb (255,0,0,125); // var1 equals a light red color
rgb var3 <- rgb (255,0,0); // var3 equals #red
rgb var4 <- rgb(rgb(255,0,0),125); // var4 equals a light red color
rgb var5 <- rgb(rgb(255,0,0),0.5); // var5 equals a light red color
rgb var6 <- rgb ("red"); // var6 equals rgb(255,0,0)
```

7.5.88.5 See also:

hsb,

7.5.89 `rgb`

7.5.89.1 Possible use:

- `rgb` (`any`) —> `rgb`

7.5.89.2 Result:

Casts the operand into the type rgb

7.5.90 `rgb_to_xyz`

7.5.90.1 Possible use:

- `rgb_to_xyz` (`file`) —> `list<point>`

7.5.90.2 Result:

A list of point corresponding to RGB value of an image (r:x , g:y, b:z)

7.5.90.3 Examples:

```
list<point> var0 <- rgb_to_xyz(texture); // var0 equals a list of points
```

7.5.91 rms

7.5.91.1 Possible use:

- `int rms float —> float`
- `rms (int , float) —> float`

7.5.91.2 Result:

Returns the RMS (Root-Mean-Square) of a data sequence. The RMS of data sequence is the square-root of the mean of the squares of the elements in the data sequence. It is a measure of the average size of the elements of a data sequence.

7.5.92 rnd

7.5.92.1 Possible use:

- `rnd (float) —> float`
- `rnd (int) —> int`
- `rnd (point) —> point`
- `float rnd float —> float`
- `rnd (float , float) —> float`
- `point rnd point —> point`
- `rnd (point , point) —> point`
- `int rnd int —> int`
- `rnd (int , int) —> int`
- `rnd (float, float, float) —> float`
- `rnd (point, point, float) —> point`
- `rnd (int, int, int) —> int`

7.5.92.2 Result:

a random integer in the interval [0, operand]

7.5.92.3 Comment:

to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision

7.5.92.4 Special cases:

- if the operand is a float, returns an uniformly distributed float random number in [0.0, to]
- if the operand is a point, returns a point with three random float ordinates, each in the interval [0, ordinate of argument]

7.5.92.5 Examples:

```
float var0 <- rnd(3.4); // var0 equals a random float between 0.0 and 3.4
float var1 <- rnd (2.0, 4.0); // var1 equals a float number between 2.0 and 4.0
float var2 <- rnd (2.0, 4.0, 0.5); // var2 equals a float number between 2.0 and 4.0 every 0.5
int var3 <- rnd (2); // var3 equals 0, 1 or 2
float var4 <- rnd (1000) / 1000; // var4 equals a float between 0 and 1 with a precision of 0.001
point var5 <- rnd ({2.5,3, 0.0}); // var5 equals {x,y} with x in [0.0,2.0], y in [0.0,3.0], z = 0.0
point var6 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0}); // var6 equals a point with x = 2.0, y between 2.0 and 4.0 an
int var7 <- rnd (2, 4); // var7 equals 2, 3 or 4
point var8 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0}, 1); // var8 equals a point with x = 2.0, y equal to 2.0, 3.0 o
int var9 <- rnd (2, 12, 4); // var9 equals 2, 6 or 10
```

7.5.92.6 See also:

flip,

7.5.93 rnd_choice

7.5.93.1 Possible use:

- **rnd_choice** (list) —> int

7.5.93.2 Result:

returns an index of the given list with a probability following the (normalized) distribution described in the list (a form of lottery)

7.5.93.3 Examples:

```
int var0 <- rnd_choice([0.2,0.5,0.3]); // var0 equals 2/10 chances to return 0, 5/10 chances to return 1, 3/1
```

7.5.93.4 See also:

rnd,

7.5.94 rnd_color

7.5.94.1 Possible use:

- **rnd_color** (int) —> rgb
- int **rnd_color** int —> rgb
- **rnd_color** (int , int) —> rgb

7.5.94.2 Result:

rgb color rgb color

7.5.94.3 Comment:

Return a random color equivalent to rgb(rnd(first_op, last_op),rnd(first_op, last_op),rnd(first_op, last_op))Return a random color equivalent to rgb(rnd(operand),rnd(operand),rnd(operand))

7.5.94.4 Examples:

```
rgb var0 <- rnd_color(100, 200); // var0 equals a random color, equivalent to rgb(rnd(100, 200),rnd(100, 200
rgb var1 <- rnd_color(255); // var1 equals a random color, equivalent to rgb(rnd(255),rnd(255),rnd(255))
```

7.5.94.5 See also:

rgb, hsb,

7.5.95 rotated_by

7.5.95.1 Possible use:

- geometry **rotated_by** float —> geometry
- **rotated_by** (geometry , float) —> geometry
- geometry **rotated_by** int —> geometry
- **rotated_by** (geometry , int) —> geometry
- **rotated_by** (geometry, float, point) —> geometry

7.5.95.2 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) A geometry resulting from the application of a rotation by the right-hand operand angles (degree) along the three axis (x,y,z) to the left-hand operand (geometry, agent, point)

7.5.95.3 Comment:

the right-hand operand can be a float or a int

7.5.95.4 Examples:

```
geometry var0 <- self rotated_by 45; // var0 equals the geometry resulting from a 45 degrees rotation to the g
geometry var1 <- rotated_by(pyramid(10),45, {1,0,0}); // var1 equals the geometry resulting from a 45 degree
```

7.5.95.5 See also:

transformed_by, translated_by,

7.5.96 round

7.5.96.1 Possible use:

- `round` (`int`) —> `int`
- `round` (`point`) —> `point`
- `round` (`float`) —> `int`

7.5.96.2 Result:

Returns the rounded value of the operand.

7.5.96.3 Special cases:

- if the operand is an int, round returns it

7.5.96.4 Examples:

```
point var0 <- {12345.78943, 12345.78943, 12345.78943} with_precision 2; // var0 equals {12345.79,12345.79,
int var1 <- round (0.51); // var1 equals 1
int var2 <- round (100.2); // var2 equals 100
int var3 <- round(-0.51); // var3 equals -1
```

7.5.96.5 See also:

round, int, with_precision,

7.5.97 row_at

7.5.97.1 Possible use:

- `matrix` `row_at` `int` —> `list`
- `row_at` (`matrix` , `int`) —> `list`

7.5.97.2 Result:

returns the row at a num_line (right-hand operand)

7.5.97.3 Examples:

```
list var0 <- matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]) row_at 2; // v
```

7.5.97.4 See also:

column_at, columns_list,

7.5.98 `rows_list`

7.5.98.1 Possible use:

- **`rows_list`** (`matrix`) —> `list<list>`

7.5.98.2 Result:

returns a list of the rows of the matrix, with each row as a list of elements

7.5.98.3 Examples:

```
list<list> var0 <- rows_list(matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]
```

7.5.98.4 See also:

columns_list,

Chapter 8

Operators (S to Z)

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

8.1 Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any 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.

8.2

8.3 Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

8.4 Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

8.5 Operators

8.5.1 sample

8.5.1.1 Possible use:

- **sample** (any expression) —> string
- string **sample** any expression —> string
- **sample** (string , any expression) —> string
- **sample** (list, int, bool) —> list
- **sample** (list, int, bool, list) —> list

8.5.1.2 Result:

takes a sample of the specified size from the elements of x using either with or without replacement with given weights takes a sample of the specified size from the elements of x using either with or without replacement

8.5.1.3 Examples:

```
list var0 <- sample([2,10,1],2,false,[0.1,0.7,0.2]); // var0 equals [10,2]
list var1 <- sample([2,10,1],2,false); // var1 equals [1,2]
```

8.5.2 Sanction

8.5.2.1 Possible use:

- **Sanction** (any) —> Sanction

8.5.2.2 Result:

Casts the operand into the type Sanction

8.5.3 saveSimulation

8.5.3.1 Possible use:

- **saveSimulation** (string) —> int

8.5.4 scaled_by

Same signification as *

8.5.5 scaled_to

8.5.5.1 Possible use:

- `geometry scaled_to point` —> `geometry`
- `scaled_to (geometry , point)` —> `geometry`

8.5.5.2 Result:

allows to restrict the size of a geometry so that it fits in the envelope {width, height, depth} defined by the second operand

8.5.5.3 Examples:

```
geometry var0 <- shape scaled_to {10,10}; // var0 equals a geometry corresponding to the geometry of the agen
```

8.5.6 select

Same signification as where

8.5.7 serialize

8.5.7.1 Possible use:

- `serialize (unknown)` —> `string`

8.5.7.2 Result:

It serializes any object, i.e. transform it into a string.

8.5.8 serializeAgent

8.5.8.1 Possible use:

- `serializeAgent (agent)` —> `string`

8.5.9 set_about

8.5.9.1 Possible use:

- emotion **set_about** predicate —> emotion
- **set_about** (emotion , predicate) —> emotion

8.5.9.2 Result:

change the about value of the given emotion

8.5.9.3 Examples:

```
emotion set_about predicate1
```

8.5.10 set_agent

8.5.10.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_agent** agent —> msi.gaml.architecture.simplebdi.Social
- **set_agent** (msi.gaml.architecture.simplebdi.SocialLink , agent) —> msi.gaml.architecture.simplebdi.Soc

8.5.10.2 Result:

change the agent value of the given social link

8.5.10.3 Examples:

```
social_link set_agent agentA
```

8.5.11 set_agent_cause

8.5.11.1 Possible use:

- predicate **set_agent_cause** agent —> predicate
- **set_agent_cause** (predicate , agent) —> predicate
- emotion **set_agent_cause** agent —> emotion
- **set_agent_cause** (emotion , agent) —> emotion

8.5.11.2 Result:

change the agentCause value of the given predicate change the agentCause value of the given emotion

8.5.11.3 Examples:

```
predicate set_agent_cause agentA emotion set_agent_cause agentA
```

8.5.12 set_decay

8.5.12.1 Possible use:

- emotion **set_decay** float —> emotion
- **set_decay** (emotion , float) —> emotion

8.5.12.2 Result:

change the decay value of the given emotion

8.5.12.3 Examples:

```
emotion set_decay 12
```

8.5.13 set_dominance

8.5.13.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_dominance** float —> msi.gaml.architecture.simplebdi.So
- **set_dominance** (msi.gaml.architecture.simplebdi.SocialLink , float) —> msi.gaml.architecture.simplebdi

8.5.13.2 Result:

change the dominance value of the given social link

8.5.13.3 Examples:

```
social_link set_dominance 0.4
```

8.5.14 set_familiarity

8.5.14.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_familiarity** float —> msi.gaml.architecture.simplebdi
- **set_familiarity** (msi.gaml.architecture.simplebdi.SocialLink , float) —> msi.gaml.architecture.simpleb

8.5.14.2 Result:

change the familiarity value of the given social link

8.5.14.3 Examples:

```
social_link set_familiarity 0.4
```

8.5.15 set_intensity

8.5.15.1 Possible use:

- `emotion` **`set_intensity`** `float` —> `emotion`
- **`set_intensity`** (`emotion` , `float`) —> `emotion`

8.5.15.2 Result:

change the intensity value of the given emotion

8.5.15.3 Examples:

```
emotion set_intensity 12
```

8.5.16 set_lifetime

8.5.16.1 Possible use:

- `mental_state` **`set_lifetime`** `int` —> `mental_state`
- **`set_lifetime`** (`mental_state` , `int`) —> `mental_state`

8.5.16.2 Result:

change the lifetime value of the given mental state

8.5.16.3 Examples:

```
mental state set_lifetime 1
```

8.5.17 set_liking

8.5.17.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_liking** float —> msi.gaml.architecture.simplebdi.Socia
- **set_liking** (msi.gaml.architecture.simplebdi.SocialLink , float) —> msi.gaml.architecture.simplebdi.So

8.5.17.2 Result:

change the liking value of the given social link

8.5.17.3 Examples:

```
social_link set_liking 0.4
```

8.5.18 set_modality

8.5.18.1 Possible use:

- mental_state **set_modality** string —> mental_state
- **set_modality** (mental_state , string) —> mental_state

8.5.18.2 Result:

change the modality value of the given mental state

8.5.18.3 Examples:

```
mental state set_modality belief
```

8.5.19 set_predicate

8.5.19.1 Possible use:

- mental_state **set_predicate** predicate —> mental_state
- **set_predicate** (mental_state , predicate) —> mental_state

8.5.19.2 Result:

change the predicate value of the given mental state

8.5.19.3 Examples:

```
mental state set_predicate pred1
```

8.5.20 set_solidarity

8.5.20.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_solidarity** float —> msi.gaml.architecture.simplebdi.S
- **set_solidarity** (msi.gaml.architecture.simplebdi.SocialLink , float) —> msi.gaml.architecture.simplebd

8.5.20.2 Result:

change the solidarity value of the given social link

8.5.20.3 Examples:

```
social_link set_solidarity 0.4
```

8.5.21 set_strength

8.5.21.1 Possible use:

- mental_state **set_strength** float —> mental_state
- **set_strength** (mental_state , float) —> mental_state

8.5.21.2 Result:

change the strength value of the given mental state

8.5.21.3 Examples:

```
mental state set_strength 1.0
```

8.5.22 set_trust

8.5.22.1 Possible use:

- msi.gaml.architecture.simplebdi.SocialLink **set_trust** float —> msi.gaml.architecture.simplebdi.Social
- **set_trust** (msi.gaml.architecture.simplebdi.SocialLink , float) —> msi.gaml.architecture.simplebdi.Soc

8.5.22.2 Result:

change the trust value of the given social link

8.5.22.3 Examples:

```
social_link set_familiarity 0.4
```

8.5.23 set_truth

8.5.23.1 Possible use:

- predicate **set_truth** bool —> predicate
- **set_truth** (predicate , bool) —> predicate

8.5.23.2 Result:

change the is_true value of the given predicate

8.5.23.3 Examples:

```
predicate set_truth false
```

8.5.24 set_z

8.5.24.1 Possible use:

- geometry **set_z** container<float> —> geometry
- **set_z** (geometry , container<float>) —> geometry
- **set_z** (geometry, int, float) —> geometry

8.5.24.2 Result:

Sets the z ordinate of the n-th point of a geometry to the value provided by the third argument

8.5.24.3 Examples:

```
loop i from: 0 to: length(shape.points) - 1{set shape <-  set_z (shape, i, 3.0);} shape <- triangle(3) set_z |
```

8.5.25 shape_file

8.5.25.1 Possible use:

- **shape_file** (string) —> file

8.5.25.2 Result:

Constructs a file of type shape. Allowed extensions are limited to shp

---

8.5.26 shuffle

8.5.26.1 Possible use:

- **shuffle** (container) —> list
- **shuffle** (matrix) —> matrix
- **shuffle** (string) —> string

8.5.26.2 Result:

The elements of the operand in random order.

8.5.26.3 Special cases:

- if the operand is empty, returns an empty list (or string, matrix)

8.5.26.4 Examples:

```
list var0 <- shuffle ([12, 13, 14]); // var0 equals [14,12,13] (for example)
matrix var1 <- shuffle (matrix([["c11","c12","c13"],["c21","c22","c23"]])); // var1 equals matrix([["c12",
string var2 <- shuffle ('abc'); // var2 equals 'bac' (for example)
```

8.5.26.5 See also:

reverse,

8.5.27 signum

8.5.27.1 Possible use:

- **signum** (float) —> int

8.5.27.2 Result:

Returns -1 if the argument is negative, +1 if it is positive, 0 if it is equal to zero or not a number

8.5.27.3 Examples:

```
int var0 <- signum(-12); // var0 equals -1
int var1 <- signum(14); // var1 equals 1
int var2 <- signum(0); // var2 equals 0
```

8.5.28 simple_clustering_by_distance

8.5.28.1 Possible use:

- container<agent> **simple_clustering_by_distance** float —> list<list<agent>>
- **simple_clustering_by_distance** (container<agent> , float) —> list<list<agent>>

8.5.28.2 Result:

A list of agent groups clustered by distance considering a distance min between two groups.

8.5.28.3 Examples:

```
list<list<agent>> var0 <- [ag1, ag2, ag3, ag4, ag5] simpleClusteringByDistance 20.0; // var0 equals for exam
```

8.5.28.4 See also:

hierarchical_clustering,

8.5.29 simple_clustering_by_envelope_distance

Same signification as simple_clustering_by_distance

8.5.30 simplex_generator

8.5.30.1 Possible use:

- **simplex_generator** (float, float, float) —> float

8.5.30.2 Result:

take a x, y and a bias parameters and gives a value

8.5.30.3 Examples:

```
float var0 <- simplex_generator(2,3,253); // var0 equals 10.2
```

8.5.31 simplification

8.5.31.1 Possible use:

- `geometry simplification float —> geometry`
- `simplification (geometry , float) —> geometry`

8.5.31.2 Result:

A geometry corresponding to the simplification of the operand (geometry, agent, point) considering a tolerance distance.

8.5.31.3 Comment:

The algorithm used for the simplification is Douglas-Peucker

8.5.31.4 Examples:

```
geometry var0 <- self simplification 0.1; // var0 equals the geometry resulting from the application of the D
```

8.5.32 sin

8.5.32.1 Possible use:

- `sin (int) —> float`
- `sin (float) —> float`

8.5.32.2 Result:

Returns the value (in [-1,1]) of the sinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

8.5.32.3 Special cases:

- Operand values out of the range [0-359] are normalized.

8.5.32.4 Examples:

```
float var0 <- sin (0); // var0 equals 0.0
float var1 <- sin(360) with_precision 10 with_precision 10; // var1 equals 0.0
```

8.5.32.5 See also:

cos, tan,

8.5.33 sin_rad

8.5.33.1 Possible use:

- `sin_rad` (`float`) —> `float`

8.5.33.2 Result:

Returns the value (in [-1,1]) of the sinus of the operand (in radians).

8.5.33.3 Examples:

```
float var0 <- sin_rad(#pi); // var0 equals 0.0
```

8.5.33.4 See also:

cos_rad, tan_rad,

8.5.34 since

8.5.34.1 Possible use:

- `since` (`date`) —> `bool`
- `any expression since date` —> `bool`
- `since` (`any expression` , `date`) —> `bool`

8.5.34.2 Result:

Returns true if the current_date of the model is after (or equal to) the date passed in argument. Synonym of 'current_date >= argument'. Can be used, like 'after', in its composed form with 2 arguments to express the lowest boundary of the computation of a frequency. However, contrary to 'after', there is a subtle difference: the lowest boundary will be tested against the frequency as well

8.5.34.3 Examples:

```
reflex when: since(starting_date) {}   // this reflex will always be run every(2#days) since (starting_date
```

8.5.35 skeletonize

8.5.35.1 Possible use:

- **skeletonize** (geometry) —> list<geometry>
- geometry **skeletonize** float —> list<geometry>
- **skeletonize** (geometry , float) —> list<geometry>
- **skeletonize** (geometry, float, float) —> list<geometry>

8.5.35.2 Result:

A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent) with the given tolerance for the clipping A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent) with the given tolerance for the clipping and for the triangulation A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent)

8.5.35.3 Examples:

```
list<geometry> var0 <- skeletonize(self); // var0 equals the list of geometries corresponding to the skeleto
list<geometry> var1 <- skeletonize(self); // var1 equals the list of geometries corresponding to the skeleto
list<geometry> var2 <- skeletonize(self); // var2 equals the list of geometries corresponding to the skeleto
```

8.5.36 skew

8.5.36.1 Possible use:

- **skew** (container) —> float
- float **skew** float —> float
- **skew** (float , float) —> float

8.5.36.2 Result:

Returns the skew of a data sequence. Returns the skew of a data sequence, which is moment(data,3,mean) / standardDeviation3

8.5.37 skew_gauss

8.5.37.1 Possible use:

- **skew_gauss** (float, float, float, float) —> float

8.5.37.2 Result:

A value from a skew normally distributed random variable with min value (the minimum skewed value possible), max value (the maximum skewed value possible), skew (the degree to which the values cluster around the mode of the distribution; higher values mean tighter clustering) and bias (the tendency of the mode to approach the min, max or midpoint value; positive values bias toward max, negative values toward min).The algorithm was taken from http://stackoverflow.com/questions/5853187/skewing-java-random-number-generation-toward-a-certain-number

8.5.37.3 Examples:

```
float var0 <- skew_gauss(0.0, 1.0, 0.7,0.1); // var0 equals 0.1729218460343077
```

8.5.37.4 See also:

gauss, truncated_gauss, poisson,

8.5.38 skewness

8.5.38.1 Possible use:

- **skewness** (list) —> float

8.5.38.2 Result:

returns skewness value computed from the operand list of values

8.5.38.3 Special cases:

- if the length of the list is lower than 3, returns NaN

8.5.38.4 Examples:

```
float var0 <- skewness ([1,2,3,4,5]); // var0 equals 0.0
```

8.5.39 skill

8.5.39.1 Possible use:

- **skill** (any) —> skill

8.5.39.2 Result:

Casts the operand into the type skill

8.5.40 `smooth`

8.5.40.1 Possible use:

- `geometry` **`smooth`** `float` —> `geometry`
- **`smooth`** (`geometry` , `float`) —> `geometry`

8.5.40.2 Result:

Returns a 'smoothed' geometry, where straight lines are replaces by polynomial (bicubic) curves. The first parameter is the original geometry, the second is the 'fit' parameter which can be in the range 0 (loose fit) to 1 (tightest fit).

8.5.40.3 Examples:

```
geometry var0 <- smooth(square(10), 0.0); // var0 equals a 'rounded' square
```

8.5.41 `social_link`

8.5.41.1 Possible use:

- **`social_link`** (`any`) —> `social_link`

8.5.41.2 Result:

Casts the operand into the type social_link

8.5.42 `solid`

Same signification as without_holes

8.5.43 `sort`

Same signification as sort_by

8.5.44 sort_by

8.5.44.1 Possible use:

- `container sort_by any expression —> list`
- `sort_by (container , any expression) —> list`

8.5.44.2 Result:

Returns 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.

8.5.44.3 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.

8.5.44.4 Special cases:

- if the left-hand operand is nil, sort_by throws an error

8.5.44.5 Examples:

```
list var0 <- [1,2,4,3,5,7,6,8] sort_by (each); // var0 equals [1,2,3,4,5,6,7,8]
list var2 <- g2 sort_by (length(g2 out_edges_of each) ); // var2 equals [node9, node7, node10, node8, node11,
list var3 <- (list(node) sort_by (round(node(each).location.x)); // var3 equals [node5, node1, node0, node2
list var4 <- [1::2, 5::6, 3::4] sort_by (each); // var4 equals [2, 4, 6]
```

8.5.44.6 See also:

group_by,

8.5.45 source_of

8.5.45.1 Possible use:

- `graph source_of unknown —> unknown`
- `source_of (graph , unknown) —> unknown`

8.5.45.2 Result:

returns the source of the edge (right-hand operand) contained in the graph given in left-hand operand.

8.5.45.3 Special cases:

- if the lef-hand operand (the graph) is nil, throws an Exception

8.5.45.4 Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"::edge,"vertices_specy"::node,"size"::3,
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1graph graphFromMap <- as_edge_graph(
point var3 <- graphFromMap source_of(link({1,5},{12,45})); // var3 equals {1,5}
```

8.5.45.5 See also:

target_of,

8.5.46 spatial_graph

8.5.46.1 Possible use:

- **spatial_graph** (container) —> graph

8.5.46.2 Result:

allows to create a spatial graph from a container of vertices, without trying to wire them. The container can be empty. Emits an error if the contents of the container are not geometries, points or agents

8.5.46.3 See also:

graph,

8.5.47 species

8.5.47.1 Possible use:

- **species** (unknown) —> species

8.5.47.2 Result:

casting of the operand to a species.

8.5.47.3 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

8.5.47.4 Examples:

```
species var0 <- species(self); // var0 equals the species of the current agent
species var1 <- species('node'); // var1 equals node
species var2 <- species([1,5,9,3]); // var2 equals nil
species var3 <- species(node1); // var3 equals node
```

8.5.48 species_of

Same signification as species

8.5.49 sphere

8.5.49.1 Possible use:

- **sphere** (float) —> geometry

8.5.49.2 Result:

A sphere geometry which radius is equal to the operand.

8.5.49.3 Comment:

the centre of the sphere is by default the location of the current agent in which has been called this operator.

8.5.49.4 Special cases:

- returns a point if the operand is lower or equal to 0.

8.5.49.5 Examples:

```
geometry var0 <- sphere(10); // var0 equals a geometry as a circle of radius 10 but displays a sphere.
```

8.5.49.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

8.5.50 split

8.5.50.1 Possible use:

- `split` (`list`) —> `list<list>`

8.5.50.2 Result:

Splits a list of numbers into n=(1+3.3*log10(elements)) bins. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith bound

8.5.50.3 See also:

split_in, split_using,

8.5.51 split_at

8.5.51.1 Possible use:

- `geometry` `split_at` `point` —> `list<geometry>`
- `split_at` (`geometry` , `point`) —> `list<geometry>`

8.5.51.2 Result:

The two part of the left-operand lines split at the given right-operand point

8.5.51.3 Special cases:

- if the left-operand is a point or a polygon, returns an empty list

8.5.51.4 Examples:

```
list<geometry> var0 <- polyline([{1,2},{4,6}]) split_at {7,6}; // var0 equals [polyline([{1.0,2.0},{7.0,6.0
```

8.5.52 split_geometry

8.5.52.1 Possible use:

- `geometry` `split_geometry` `float` —> `list<geometry>`
- `split_geometry` (`geometry` , `float`) —> `list<geometry>`
- `geometry` `split_geometry` `point` —> `list<geometry>`
- `split_geometry` (`geometry` , `point`) —> `list<geometry>`
- `split_geometry` (`geometry`, `int`, `int`) —> `list<geometry>`

8.5.52.2 Result:

A list of geometries that result from the decomposition of the geometry by square cells of the given side size (geometry, size) A list of geometries that result from the decomposition of the geometry according to a grid with the given number of rows and columns (geometry, nb_cols, nb_rows) A list of geometries that result from the decomposition of the geometry by rectangle cells of the given dimension (geometry, {size_x, size_y})

8.5.52.3 Examples:

```
list<geometry> var0 <- to_squares(self, 10.0); // var0 equals the list of the geometries corresponding to th
list<geometry> var1 <- to_rectangles(self, 10,20); // var1 equals the list of the geometries corresponding t
list<geometry> var2 <- to_rectangles(self, {10.0, 15.0}); // var2 equals the list of the geometries correspo
```

8.5.53 split_in

8.5.53.1 Possible use:

- `list` **`split_in`** `int` —> `list<list>`
- **`split_in`** (`list` , `int`) —> `list<list>`
- **`split_in`** (`list`, `int`, `bool`) —> `list<list>`

8.5.53.2 Result:

Splits a list of numbers into n bins defined by n-1 bounds between the minimum and maximum values found in the first argument. The boolean argument controls whether or not the splitting is strict (if true, elements are in the ith bin if they are strictly smaller than the ith bound Splits a list of numbers into n bins defined by n-1 bounds between the minimum and maximum values found in the first argument. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith bound

8.5.53.3 See also:

split, split_using,

8.5.54 split_lines

8.5.54.1 Possible use:

- **`split_lines`** (`container<geometry>`) —> `list<geometry>`
- `container<geometry>` **`split_lines`** `bool` —> `list<geometry>`
- **`split_lines`** (`container<geometry>` , `bool`) —> `list<geometry>`

8.5.54.2 Result:

A list of geometries resulting after cutting the lines at their intersections. A list of geometries resulting after cutting the lines at their intersections. if the last boolean operand is set to true, the split lines will import the attributes of the initial lines

8.5.54.3 Examples:

```
list<geometry> var0 <- split_lines([line([{0,10}, {20,10}]), line([{0,10}, {20,10}])]); // var0 equals a li
list<geometry> var1 <- split_lines([line([{0,10}, {20,10}]), line([{0,10}, {20,10}])]); // var1 equals a li
```

8.5.55 split_using

8.5.55.1 Possible use:

- `list` **`split_using`** `msi.gama.util.IList<? extends java.lang.Comparable>` —> `list<list>`
- **`split_using`** (`list` , `msi.gama.util.IList<? extends java.lang.Comparable>`) —> `list<list>`
- **`split_using`** (`list`, `msi.gama.util.IList<? extends java.lang.Comparable>`, `bool`) —> `list<list>`

8.5.55.2 Result:

Splits a list of numbers into n+1 bins using a set of n bounds passed as the second argument. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith bound Splits a list of numbers into n+1 bins using a set of n bounds passed as the second argument. The boolean argument controls whether or not the splitting is strict (if true, elements are in the ith bin if they are strictly smaller than the ith bound

8.5.55.3 See also:

split, split_in,

8.5.56 split_with

8.5.56.1 Possible use:

- `string` **`split_with`** `string` —> `list`
- **`split_with`** (`string` , `string`) —> `list`
- **`split_with`** (`string`, `string`, `bool`) —> `list`

8.5.56.2 Result:

Returns a list containing the sub-strings (tokens) of the left-hand operand delimited by each of the characters of the right-hand operand. Returns a list containing the sub-strings (tokens) of the left-hand operand delimited either by each of the characters of the right-hand operand (false) or by the whole right-hand operand (true).

8.5.56.3 Comment:

Delimiters themselves are excluded from the resulting list.Delimiters themselves are excluded from the resulting list.

8.5.56.4 Examples:

```
list var0 <- 'to be or not to be,that is the question' split_with ' ,'; // var0 equals ['to','be','or','not',
list var1 <- 'aa::bb:cc' split_with ('::', true); // var1 equals ['aa','bb:cc']
```

8.5.57 sqrt

8.5.57.1 Possible use:

- **sqrt** (int) —> float
- **sqrt** (float) —> float

8.5.57.2 Result:

Returns the square root of the operand.

8.5.57.3 Special cases:

- if the operand is negative, an exception is raised

8.5.57.4 Examples:

```
float var0 <- sqrt(4); // var0 equals 2.0
float var1 <- sqrt(4); // var1 equals 2.0
```

8.5.58 square

8.5.58.1 Possible use:

- **square** (float) —> geometry

8.5.58.2 Result:

A square geometry which side size is equal to the operand.

8.5.58.3 Comment:

the centre of the square is by default the location of the current agent in which has been called this operator.

8.5.58.4 Special cases:

- returns nil if the operand is nil.

8.5.58.5 Examples:

```
geometry var0 <- square(10); // var0 equals a geometry as a square of side size 10.
```

8.5.58.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

8.5.59 squircle

8.5.59.1 Possible use:

- `float` **`squircle`** `float` —> `geometry`
- **`squircle`** (`float` , `float`) —> `geometry`

8.5.59.2 Result:

A mix of square and circle geometry (see : http://en.wikipedia.org/wiki/Squircle), which side size is equal to the first operand and power is equal to the second operand

8.5.59.3 Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

8.5.59.4 Special cases:

- returns a point if the side operand is lower or equal to 0.

8.5.59.5 Examples:

```
geometry var0 <- squircle(4,4); // var0 equals a geometry as a squircle of side 4 with a power of 4.
```

8.5.59.6 See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

8.5.60 stack

8.5.60.1 Possible use:

- **stack** (msi.gama.util.IList<java.lang.Integer>) —> msi.gama.util.tree.GamaNode<java.lang.String>

8.5.61 standard_deviation

8.5.61.1 Possible use:

- **standard_deviation** (container) —> float

8.5.61.2 Result:

the standard deviation on the elements of the operand. See Standard_deviation for more details.

8.5.61.3 Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

8.5.61.4 Examples:

```
float var0 <- standard_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.2930100540985752
```

8.5.61.5 See also:

mean, mean_deviation,

8.5.62 step_sub_model

8.5.62.1 Possible use:

- **step_sub_model** (msi.gama.kernel.experiment.IExperimentAgent) —> int

8.5.62.2 Result:

Load a submodel

8.5.62.3 Comment:

loaded submodel

8.5.63 strahler

8.5.63.1 Possible use:

- `strahler` (`graph`) —> `map`

8.5.63.2 Result:

retur for each edge, its strahler number

8.5.64 string

8.5.64.1 Possible use:

- `date` **`string`** `string` —> `string`
- **`string`** (`date` , `string`) —> `string`
- **`string`** (`date`, `string`, `string`) —> `string`

8.5.64.2 Result:

converts a date to astring following a custom pattern and using a specific locale (e.g.: 'fr', 'en', etc.). The pattern can use "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will return the complete date as defined by the ISO date & time format. The pattern can also follow the pattern definition found here, which gives much more control over the format of the date: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns. Different patterns are available by default as constants: #iso_local, #iso_simple, #iso_offset, #iso_zoned and #custom, which can be changed in the preferences converts a date to astring following a custom pattern. The pattern can use "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will return the complete date as defined by the ISO date & time format. The pattern can also follow the pattern definition found here, which gives much more control over the format of the date: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns. Different patterns are available by default as constants: #iso_local, #iso_simple, #iso_offset, #iso_zoned and #custom, which can be changed in the preferences

8.5.64.3 Examples:

```
format(#now, 'yyyy-MM-dd') format(#now, 'yyyy-MM-dd')
```

8.5.65 student_area

8.5.65.1 Possible use:

- `float` **`student_area`** `int` —> `float`
- **`student_area`** (`float` , `int`) —> `float`

8.5.65.2 Result:

Returns the area to the left of x in the Student T distribution with the given degrees of freedom.

8.5.66 `student_t_inverse`

8.5.66.1 Possible use:

- `float` **`student_t_inverse`** `int` —> `float`
- **`student_t_inverse`** (`float` , `int`) —> `float`

8.5.66.2 Result:

Returns the value, t, for which the area under the Student-t probability density function (integrated from minus infinity to t) is equal to x.

8.5.67 `subtract_days`

Same signification as minus_days

8.5.68 `subtract_hours`

Same signification as minus_hours

8.5.69 `subtract_minutes`

Same signification as minus_minutes

8.5.70 `subtract_months`

Same signification as minus_months

8.5.71 `subtract_ms`

Same signification as minus_ms

8.5.72 subtract_seconds

Same signification as -

8.5.73 subtract_weeks

Same signification as minus_weeks

8.5.74 subtract_years

Same signification as minus_years

8.5.75 successors_of

8.5.75.1 Possible use:

- `graph successors_of unknown —> list`
- `successors_of (graph , unknown) —> list`

8.5.75.2 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)

8.5.75.3 Examples:

```
list var1 <- graphEpidemio successors_of ({1,5}); // var1 equals [{12,45}]
list var2 <- graphEpidemio successors_of node({34,56}); // var2 equals []
```

8.5.75.4 See also:

predecessors_of, neighbors_of,

8.5.76 sum

8.5.76.1 Possible use:

- `sum (container) —> unknown`
- `sum (graph) —> float`

8.5.76.2 Result:

the sum of all the elements of the operand

8.5.76.3 Comment:

the behavior depends on the nature of the operand

8.5.76.4 Special cases:

- if it is a population or a list of other types: sum transforms all elements into float 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 total weight of 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 other types: sum transforms all elements into float and sums them
- if it is a list of colors: sum will sum them and return the blended resulting color
- if it is a list of int or float: sum returns the sum of all the elements

```
int var0 <- sum ([12,10,3]); // var0 equals 25
```

- 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)

```
unknown var1 <- sum([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]); // var1 equals {20.0,17.0}
```

8.5.76.5 See also:

mul,

8.5.77 sum_of

8.5.77.1 Possible use:

- `container` **`sum_of`** `any expression` —> `unknown`
- **`sum_of`** (`container` , `any expression`) —> `unknown`

8.5.77.2 Result:

the sum of the right-hand expression evaluated on each of the elements of the left-hand operand

8.5.77.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

8.5.77.4 Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] sum_of (each + 3); // var1 equals 21
```

8.5.77.5 Examples:

```
unknown var0 <- [1,2] sum_of (each * 100 ); // var0 equals 300
```

8.5.77.6 See also:

min_of, max_of, product_of, mean_of,

8.5.78 `svg_file`

8.5.78.1 Possible use:

- **`svg_file`** (`string`) —> `file`

8.5.78.2 Result:

Constructs a file of type svg. Allowed extensions are limited to svg

---

8.5.79 `tan`

8.5.79.1 Possible use:

- **`tan`** (`float`) —> `float`
- **`tan`** (`int`) —> `float`

8.5.79.2 Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in decimal degrees).

8.5.79.3 Special cases:

- Operand values out of the range [0-359] are normalized. Notice that tan(360) does not return 0.0 but -2.4492935982947064E-16
- The tangent is only defined for any real number except 90 + k * 180 (k an positive or negative integer). Nevertheless notice that tan(90) returns 1.633123935319537E16 (whereas we could except infinity).

8.5.79.4 Examples:

```
float var0 <- tan (0); // var0 equals 0.0
float var1 <- tan(90); // var1 equals 1.633123935319537E16
```

8.5.79.5 See also:

cos, sin,

8.5.80 tan_rad

8.5.80.1 Possible use:

- `tan_rad` (`float`) —> `float`

8.5.80.2 Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in radians).

8.5.80.3 See also:

cos_rad, sin_rad,

8.5.81 tanh

8.5.81.1 Possible use:

- `tanh` (`int`) —> `float`
- `tanh` (`float`) —> `float`

8.5.81.2 Result:

Returns the value (in the interval [-1,1]) of the hyperbolic tangent of the operand (which can be any real number, expressed in decimal degrees).

8.5.81.3 Examples:

```
float var0 <- tanh(0); // var0 equals 0.0
float var1 <- tanh(100); // var1 equals 1.0
```

8.5.82 target_of

8.5.82.1 Possible use:

- graph **target_of** unknown —> unknown
- **target_of** (graph , unknown) —> unknown

8.5.82.2 Result:

returns the target of the edge (right-hand operand) contained in the graph given in left-hand operand.

8.5.82.3 Special cases:

- if the lef-hand operand (the graph) is nil, returns nil

8.5.82.4 Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"::edge,"vertices_specy"::node,"size"::3,
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1graph graphFromMap <- as_edge_graph(
unknown var3 <- graphFromMap target_of(link({1,5},{12,45})); // var3 equals {12,45}
```

8.5.82.5 See also:

source_of,

8.5.83 teapot

8.5.83.1 Possible use:

- **teapot** (float) —> geometry

8.5.83.2 Result:

A teapot geometry which radius is equal to the operand.

8.5.83.3 Comment:

the centre of the teapot is by default the location of the current agent in which has been called this operator.

8.5.83.4 Special cases:

- returns a point if the operand is lower or equal to 0.

8.5.83.5 Examples:

```
geometry var0 <- teapot(10); // var0 equals a geometry as a circle of radius 10 but displays a teapot.
```

8.5.83.6 See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

8.5.84 `text_file`

8.5.84.1 Possible use:

- `text_file` (`string`) —> `file`

8.5.84.2 Result:

Constructs a file of type text. Allowed extensions are limited to txt, data, text

---

8.5.85 `TGauss`

Same signification as truncated_gauss

8.5.86 `threeds_file`

8.5.86.1 Possible use:

- `threeds_file` (`string`) —> `file`

8.5.86.2 Result:

Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

---

8.5.87 `to`

Same signification as until

8.5.87.1 Possible use:

- `date to date` —> `msi.gama.util.IList<msi.gama.util.GamaDate>`
- `to` (`date` , `date`) —> `msi.gama.util.IList<msi.gama.util.GamaDate>`

8.5.87.2 Result:

builds an interval between two dates (the first inclusive and the second exclusive, which behaves like a read-only list of dates. The default step between two dates is the step of the model

8.5.87.3 Comment:

The default step can be overruled by using the every operator applied to this interval

8.5.87.4 Examples:

```
date('2000-01-01') to date('2010-01-01') // builds an interval between these two dates (date('2000-01-01') t
```

8.5.87.5 See also:

every,

8.5.88 to_GAMA_CRS

8.5.88.1 Possible use:

- `to_GAMA_CRS` (`geometry`) —> `geometry`
- `geometry to_GAMA_CRS string` —> `geometry`
- `to_GAMA_CRS` (`geometry` , `string`) —> `geometry`

8.5.88.2 Special cases:

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by the current CRS, the one corresponding to the world's agent one

```
geometry var0 <- to_GAMA_CRS({121,14}); // var0 equals a geometry corresponding to the agent geometry transf
```

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by given CRS

```
geometry var1 <- to_GAMA_CRS({121,14}, "EPSG:4326"); // var1 equals a geometry corresponding to the agent ge
```

8.5.89 to_gaml

8.5.89.1 Possible use:

- `to_gaml` (unknown) —> `string`

8.5.89.2 Result:

returns the literal description of an expression or description – action, behavior, species, aspect, even model – in gaml

8.5.89.3 Examples:

```
string var0 <- to_gaml(0); // var0 equals '0'
string var1 <- to_gaml(3.78); // var1 equals '3.78'
string var2 <- to_gaml(true); // var2 equals 'true'
string var3 <- to_gaml({23, 4.0}); // var3 equals '{23.0,4.0,0.0}'
string var4 <- to_gaml(5::34); // var4 equals '5::34'
string var5 <- to_gaml(rgb(255,0,125)); // var5 equals 'rgb (255, 0, 125,255)'
string var6 <- to_gaml('hello'); // var6 equals "'hello'"
string var7 <- to_gaml([1,5,9,3]); // var7 equals '[1,5,9,3]'
string var8 <- to_gaml(['a'::345, 'b'::13, 'c'::12]); // var8 equals "map(['a'::345,'b'::13,'c'::12])"
string var9 <- to_gaml([[3,5,7,9],[2,4,6,8]]); // var9 equals '[[3,5,7,9],[2,4,6,8]]'
string var10 <- to_gaml(a_graph); // var10 equals ([((1 as node)::(3 as node))::(5 as edge),((0 as node)::(3
string var11 <- to_gaml(node1); // var11 equals  1 as node
```

8.5.90 to_rectangles

8.5.90.1 Possible use:

- `to_rectangles` (`geometry`, `point`, `bool`) —> `list<geometry>`
- `to_rectangles` (`geometry`, `int`, `int`, `bool`) —> `list<geometry>`

8.5.90.2 Result:

A list of rectangles of the size corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, dimension, overlaps), if overlaps = true, add the rectangles that overlap the border of the geometry A list of rectangles corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, nb_cols, nb_rows, overlaps) by a grid composed of the given number of columns and rows, if overlaps = true, add the rectangles that overlap the border of the geometry

8.5.90.3 Examples:

```
list<geometry> var0 <- to_rectangles(self, {10.0, 15.0}, true); // var0 equals the list of rectangles of siz
list<geometry> var1 <- to_rectangles(self, 5, 20, true); // var1 equals the list of rectangles corresponding
```

8.5.91 to_squares

8.5.91.1 Possible use:

- `to_squares` (`geometry`, `int`, `bool`) —> `list<geometry>`
- `to_squares` (`geometry`, `float`, `bool`) —> `list<geometry>`
- `to_squares` (`geometry`, `int`, `bool`, `float`) —> `list<geometry>`

8.5.91.2 Result:

A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps, precision_coefficient), if overlaps = true, add the squares that overlap the border of the geometry, coefficient_precision should be close to 1.0 A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps), if overlaps = true, add the squares that overlap the border of the geometry A list of squares of the size corresponding to the given size that result from the decomposition of the geometry into squares (geometry, size, overlaps), if overlaps = true, add the squares that overlap the border of the geometry

8.5.91.3 Examples:

```
list<geometry> var0 <- to_squares(self, 10, true, 0.99); // var0 equals the list of 10 squares corresponding
list<geometry> var1 <- to_squares(self, 10, true); // var1 equals the list of 10 squares corresponding to the
list<geometry> var2 <- to_squares(self, 10.0, true); // var2 equals the list of squares of side size 10.0 cor
```

8.5.92 to_sub_geometries

8.5.92.1 Possible use:

- `geometry` **`to_sub_geometries`** `list<float>` —> `list<geometry>`
- **`to_sub_geometries`** (`geometry` , `list<float>`) —> `list<geometry>`
- **`to_sub_geometries`** (`geometry`, `list<float>`, `float`) —> `list<geometry>`

8.5.92.2 Result:

A list of geometries resulting after spliting the geometry into sub-geometries. A list of geometries resulting after spliting the geometry into sub-geometries.

8.5.92.3 Examples:

```
list<geometry> var0 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4], 1.0); // var0 equals a list of
list<geometry> var1 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4]); // var1 equals a list of three
```

8.5.93 to_triangles

Same signification as triangulate

8.5.94 tokenize

Same signification as split_with

8.5.95 topology

8.5.95.1 Possible use:

- `topology` (`unknown`) —> `topology`

8.5.95.2 Result:

casting of the operand to a topology.

8.5.95.3 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.

8.5.95.4 Examples:

```
topology var0 <- topology(0); // var0 equals niltopology(a_graph)  --: Multiple topology in POLYGON ((24.71
```

8.5.95.5 See also:

geometry,

8.5.96 topology

8.5.96.1 Possible use:

- `topology` (`any`) —> `topology`

8.5.96.2 Result:

Casts the operand into the type topology

8.5.97 touches

8.5.97.1 Possible use:

- `geometry` **`touches`** `geometry` —> `bool`
- **`touches`** (`geometry` , `geometry`) —> `bool`

8.5.97.2 Result:

A boolean, equal to true if the left-geometry (or agent/point) touches the right-geometry (or agent/point).

8.5.97.3 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.

8.5.97.4 Special cases:

- if one of the operand is null, returns false.

8.5.97.5 Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) touches {15,15}; // var0 equals false
bool var1 <- polyline([{10,10},{20,20}]) touches {10,10}; // var1 equals true
bool var2 <- {15,15} touches {15,15}; // var2 equals false
bool var3 <- polyline([{10,10},{20,20}]) touches polyline([{10,10},{5,5}]); // var3 equals true
bool var4 <- polyline([{10,10},{20,20}]) touches polyline([{5,5},{15,15}]); // var4 equals false
bool var5 <- polyline([{10,10},{20,20}]) touches polyline([{15,15},{25,25}]); // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{15,15},{15,25},{25,25},{25,15}]
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,20},{20,20},{20,30},{10,30}]
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,10},{0,10},{0,0},{10,0}]); /
bool var9 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches {15,15}; // var9 equals false
bool var10 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches {10,15}; // var10 equals true
```

8.5.97.6 See also:

disjoint_from, crosses, overlaps, partially_overlaps, intersects,

---

8.5.98 towards

8.5.98.1 Possible use:

- `geometry towards geometry —> float`
- `towards (geometry , geometry) —> float`

8.5.98.2 Result:

The direction (in degree) between the two geometries (geometries, agents, points) considering the topology of the agent applying the operator.

8.5.98.3 Examples:

```
float var0 <- ag1 towards ag2; // var0 equals the direction between ag1 and ag2 and ag3 considering the topolo
```

8.5.98.4 See also:

distance_between, distance_to, direction_between, path_between, path_to,

---

8.5.99 trace

8.5.99.1 Possible use:

- `trace (matrix) —> float`

8.5.99.2 Result:

The trace of the given matrix (the sum of the elements on the main diagonal).

8.5.99.3 Examples:

```
float var0 <- trace(matrix([[1,2],[3,4]])); // var0 equals 5
```

8.5.100 `transformed_by`

8.5.100.1 Possible use:

- `geometry` **`transformed_by`** `point` —> `geometry`
- **`transformed_by`** (`geometry` , `point`) —> `geometry`

8.5.100.2 Result:

A geometry resulting from the application of a rotation and a scaling (right-operand : point {angle(degree), scale factor} of the left-hand operand (geometry, agent, point)

8.5.100.3 Examples:

```
geometry var0 <- self transformed_by {45, 0.5}; // var0 equals the geometry resulting from 45 degrees rotatio
```

8.5.100.4 See also:

rotated_by, translated_by,

8.5.101 `translated_by`

8.5.101.1 Possible use:

- `geometry` **`translated_by`** `point` —> `geometry`
- **`translated_by`** (`geometry` , `point`) —> `geometry`

8.5.101.2 Result:

A geometry resulting from the application of a translation by the right-hand operand distance to the left-hand operand (geometry, agent, point)

8.5.101.3 Examples:

```
geometry var0 <- self translated_by {10,10,10}; // var0 equals the geometry resulting from applying the trans
```

8.5.101.4 See also:

rotated_by, transformed_by,

8.5.102 translated_to

Same signification as at_location

8.5.103 transpose

8.5.103.1 Possible use:

- **transpose** (matrix) —> matrix

8.5.103.2 Result:

The transposition of the given matrix

8.5.103.3 Examples:

```
matrix var0 <- transpose(matrix([[5,-3],[6,-4]])); // var0 equals matrix([[5,6],[-3,-4]])
```

8.5.104 triangle

8.5.104.1 Possible use:

- **triangle** (float) —> geometry

8.5.104.2 Result:

A triangle geometry which side size is given by the operand.

8.5.104.3 Comment:

the center of the triangle is by default the location of the current agent in which has been called this operator.

8.5.104.4 Special cases:

- returns nil if the operand is nil.

8.5.104.5 Examples:

```
geometry var0 <- triangle(5); // var0 equals a geometry as a triangle with side_size = 5.
```

8.5.104.6 See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, square,

---

8.5.105 triangulate

8.5.105.1 Possible use:

- **triangulate** (list<geometry>) —> list<geometry>
- **triangulate** (geometry) —> list<geometry>
- geometry **triangulate** float —> list<geometry>
- **triangulate** (geometry , float) —> list<geometry>
- **triangulate** (geometry, float, float) —> list<geometry>

8.5.105.2 Result:

A list of geometries (triangles) corresponding to the Delaunay triangulation computed from the list of polylines A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point) with the given tolerance for the clipping A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point) with the given tolerance for the clipping and for the triangulation A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point)

8.5.105.3 Examples:

```
list<geometry> var0 <- triangulate([line([{0,50},{100,50}]), line([{50,0},{50,100}])); // var0 equals the l
list<geometry> var1 <- triangulate(self, 0.1); // var1 equals the list of geometries (triangles) correspondi
list<geometry> var2 <- triangulate(self,0.1, 1.0); // var2 equals the list of geometries (triangles) corresp
list<geometry> var3 <- triangulate(self); // var3 equals the list of geometries (triangles) corresponding to
```

8.5.106 truncated_gauss

8.5.106.1 Possible use:

- **truncated_gauss** (point) —> float
- **truncated_gauss** (list) —> float

8.5.106.2 Result:

A random value from a normally distributed random variable in the interval]mean - standardDeviation; mean + standardDeviation[.

8.5.106.3 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

8.5.106.4 Examples:

```
float var0 <- truncated_gauss ({0, 0.3}); // var0 equals a float between -0.3 and 0.3
float var1 <- truncated_gauss ([0.5, 0.0]); // var1 equals 0.5
```

8.5.106.5 See also:

gauss,

8.5.107 `type_of`

8.5.107.1 Possible use:

- `type_of` (`unknown`) —> `msi.gaml.types.IType<?>`

8.5.108 `undirected`

8.5.108.1 Possible use:

- `undirected` (`graph`) —> `graph`

8.5.108.2 Result:

the operand graph becomes an undirected graph.

8.5.108.3 Comment:

the operator alters the operand graph, it does not create a new one.

8.5.108.4 See also:

directed,

8.5.109 union

8.5.109.1 Possible use:

- **union** (container<geometry>) —> geometry
- container **union** container —> list
- **union** (container , container) —> list

8.5.109.2 Result:

returns a new list containing all the elements of both containers without duplicated elements.

8.5.109.3 Special cases:

- if the right-operand is a container of points, geometries or agents, returns the geometry resulting from the union all the geometries
- if the left or right operand is nil, union throws an error

8.5.109.4 Examples:

```
geometry var0 <- union([geom1, geom2, geom3]); // var0 equals a geometry corresponding to union between geom
list var1 <- [1,2,3,4,5,6] union [2,4,9]; // var1 equals [1,2,3,4,5,6,9]
list var2 <- [1,2,3,4,5,6] union [0,8]; // var2 equals [1,2,3,4,5,6,0,8]
list var3 <- [1,3,2,4,5,6,8,5,6] union [0,8]; // var3 equals [1,3,2,4,5,6,8,0]
```

8.5.109.5 See also:

inter, +,

8.5.110 unknown

8.5.110.1 Possible use:

- **unknown** (any) —> unknown

8.5.110.2 Result:

Casts the operand into the type unknown

8.5.111 unSerializeSimulation

8.5.111.1 Possible use:

- **unSerializeSimulation** (string) —> int

8.5.111.2 Result:

unSerializeSimulation

8.5.112 unSerializeSimulationFromFile

8.5.112.1 Possible use:

- `unSerializeSimulationFromFile` (`string`) —> `int`

8.5.113 until

8.5.113.1 Possible use:

- `until` (`date`) —> `bool`
- `any expression until date` —> `bool`
- `until` (`any expression` , `date`) —> `bool`

8.5.113.2 Result:

Returns true if the current_date of the model is before (or equel to) the date passed in argument. Synonym of 'current_date <= argument'

8.5.113.3 Examples:

```
reflex when: until(starting_date) {}   // This reflex will be run only once at the beginning of the simulatio
```

8.5.114 upper_case

8.5.114.1 Possible use:

- `upper_case` (`string`) —> `string`

8.5.114.2 Result:

Converts all of the characters in the string operand to upper case

8.5.114.3 Examples:

```
string var0 <- upper_case("Abc"); // var0 equals 'ABC'
```

8.5.114.4 See also:

lower_case,

8.5.115 URL_file

8.5.115.1 Possible use:

- `URL_file` (`string`) —> `file`

8.5.115.2 Result:

Constructs a file of type URL. Allowed extensions are limited to url

---

8.5.116 use_cache

8.5.116.1 Possible use:

- `graph` **`use_cache`** `bool` —> `graph`
- **`use_cache`** (`graph` , `bool`) —> `graph`

8.5.116.2 Result:

if the second operand is true, the operand graph will store in a cache all the previously computed shortest path (the cache be cleared if the graph is modified).

8.5.116.3 Comment:

the operator alters the operand graph, it does not create a new one.

8.5.116.4 See also:

path_between,

8.5.117 user_input

8.5.117.1 Possible use:

- **`user_input`** (`any expression`) —> `map<string,unknown>`
- `string` **`user_input`** `any expression` —> `map<string,unknown>`
- **`user_input`** (`string` , `any expression`) —> `map<string,unknown>`

8.5.117.2 Result:

asks the user for some values (not defined as parameters). Takes a string (optional) and a map as arguments. The string is used to specify the message of the dialog box. The map is to specify the parameters you want the user to change before the simulation starts, with the name of the parameter in string key, and the default value as value.

8.5.117.3 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 :

8.5.117.4 Examples:

```
map<string,unknown> values <- user_input(["Number" :: 100, "Location" :: {10, 10}]); create bug number: int(
```

8.5.118 using

8.5.118.1 Possible use:

- `any expression` **`using`** `topology` —> `unknown`
- **`using`** (`any expression` , `topology`) —> `unknown`

8.5.118.2 Result:

Allows to specify in which topology a spatial computation should take place.

8.5.118.3 Special cases:

- has no effect if the topology passed as a parameter is nil

8.5.118.4 Examples:

```
unknown var0 <- (agents closest_to self) using topology(world); // var0 equals the closest agent to self (the
```

8.5.119 variance

8.5.119.1 Possible use:

- **`variance`** (`container`) —> `float`

8.5.119.2 Result:

the variance of the elements of the operand. See Variance for more details.

8.5.119.3 Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

8.5.119.4 Examples:

```
float var0 <- variance ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.671875
```

8.5.119.5 See also:

mean, median,

8.5.120 variance

8.5.120.1 Possible use:

- `variance` (`float`) —> `float`
- `variance` (`int`, `float`, `float`) —> `float`

8.5.120.2 Result:

Returns the variance of a data sequence. That is (sumOfSquares - mean*sum) / size with mean = sum/size. Returns the variance from a standard deviation.

8.5.121 variance_of

8.5.121.1 Possible use:

- `container variance_of any expression` —> `unknown`
- `variance_of` (`container` , `any expression`) —> `unknown`

8.5.121.2 Result:

the variance of the right-hand expression evaluated on each of the elements of the left-hand operand

8.5.121.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

8.5.121.4 See also:

min_of, max_of, sum_of, product_of,

8.5.122 vertical

8.5.122.1 Possible use:

- `vertical` (`msi.gama.util.GamaMap<java.lang.Object,java.lang.Integer>`) —> `msi.gama.util.tree.GamaNode<`

8.5.123 voronoi

8.5.123.1 Possible use:

- `voronoi` (`list<point>`) —> `list<geometry>`
- `list<point>` `voronoi` `geometry` —> `list<geometry>`
- `voronoi` (`list<point>` , `geometry`) —> `list<geometry>`

8.5.123.2 Result:

A list of geometries corresponding to the Voronoi diagram built from the list of points A list of geometries corresponding to the Voronoi diagram built from the list of points according to the given clip

8.5.123.3 Examples:

```
list<geometry> var0 <- voronoi([{10,10},{50,50},{90,90},{10,90},{90,10}]); // var0 equals the list of geome
list<geometry> var1 <- voronoi([{10,10},{50,50},{90,90},{10,90},{90,10}], square(300)); // var1 equals the
```

8.5.124 weight_of

8.5.124.1 Possible use:

- `graph` `weight_of` `unknown` —> `float`
- `weight_of` (`graph` , `unknown`) —> `float`

8.5.124.2 Result:

returns the weight of the given edge (right-hand operand) contained in the graph given in right-hand operand.

8.5.124.3 Comment:

In a localized graph, an edge has a weight by default (the distance between both vertices).

8.5.124.4 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.

8.5.124.5 Examples:

```
graph graphFromMap <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
float var1 <- graphFromMap weight_of(link({1,5},{12,45})); // var1 equals 1.0
```

8.5.125 weighted_means_DM

8.5.125.1 Possible use:

- msi.gama.util.IList<java.util.List> **weighted_means_DM** msi.gama.util.IList<java.util.Map<java.lang.St
  —> int
- **weighted_means_DM** (msi.gama.util.IList<java.util.List> , msi.gama.util.IList<java.util.Map<java.lang.
  —> int

8.5.125.2 Result:

The index of the candidate that maximizes the weighted mean of its criterion values. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion (list of map)

8.5.125.3 Special cases:

- returns -1 is the list of candidates is nil or empty

8.5.125.4 Examples:

```
int var0 <- weighted_means_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"utility", "weight" :: 2.0],["na
```

8.5.125.5 See also:

promethee_DM, electre_DM, evidence_theory_DM,

8.5.126 where

8.5.126.1 Possible use:

- container **where** any expression —> list
- **where** (container , any expression) —> list

8.5.126.2 Result:

a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.

8.5.126.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

8.5.126.4 Special cases:

- if the left-hand operand is a list nil, where returns a new empty list
- if the left-operand is a map, the keyword each will contain each value

```
list var4 <- [1::2, 3::4, 5::6] where (each >= 4); // var4 equals [4, 6]
```

8.5.126.5 Examples:

```
list var0 <- [1,2,3,4,5,6,7,8] where (each > 3); // var0 equals [4, 5, 6, 7, 8]
list var2 <- g2 where (length(g2 out_edges_of each) = 0 ); // var2 equals [node9, node7, node10, node8, node1
list var3 <- (list(node) where (round(node(each).location.x) > 32); // var3 equals [node2, node3]
```

8.5.126.6 See also:

first_with, last_with, where,

8.5.127 with_lifetime

8.5.127.1 Possible use:

- predicate **with_lifetime** int —> predicate
- **with_lifetime** (predicate , int) —> predicate

8.5.127.2 Result:

change the parameters of the given predicate

8.5.127.3 Examples:

```
predicate with_lifetime 10
```

8.5.128 with_max_of

8.5.128.1 Possible use:

- container **with_max_of** any expression —> unknown
- **with_max_of** (container , any expression) —> unknown

8.5.128.2 Result:

one of elements of the left-hand operand that maximizes the value of the right-hand operand

8.5.128.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

8.5.128.4 Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

8.5.128.5 Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_max_of (each ); // var0 equals 8
unknown var2 <- g2 with_max_of (length(g2 out_edges_of each)  ) ; // var2 equals node4
unknown var3 <- (list(node) with_max_of (round(node(each).location.x)); // var3 equals node3
unknown var4 <- [1::2, 3::4, 5::6] with_max_of (each); // var4 equals 6
```

8.5.128.6 See also:

where, with_min_of,

8.5.129 with_min_of

8.5.129.1 Possible use:

- container **with_min_of** any expression —> unknown
- **with_min_of** (container , any expression) —> unknown

8.5.129.2 Result:

one of elements of the left-hand operand that minimizes the value of the right-hand operand

8.5.129.3 Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

8.5.129.4 Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

8.5.129.5 Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_min_of (each ); // var0 equals 1
unknown var2 <- g2 with_min_of (length(g2 out_edges_of each)  ); // var2 equals node11
unknown var3 <- (list(node) with_min_of (round(node(each).location.x)); // var3 equals node0
unknown var4 <- [1::2, 3::4, 5::6] with_min_of (each); // var4 equals 2
```

8.5.129.6 See also:

where, with_max_of,

8.5.130 `with_optimizer_type`

8.5.130.1 Possible use:

- `graph with_optimizer_type string` —> `graph`
- `with_optimizer_type` (`graph` , `string`) —> `graph`

8.5.130.2 Result:

changes the shortest path computation method of the given graph

8.5.130.3 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.

8.5.130.4 Examples:

```
graphEpidemio <- graphEpidemio with_optimizer_type "static";
```

8.5.130.5 See also:

set_verbose,

8.5.131 with_precision

8.5.131.1 Possible use:

- `float with_precision int —> float`
- `with_precision (float , int) —> float`
- `point with_precision int —> point`
- `with_precision (point , int) —> point`
- `geometry with_precision int —> geometry`
- `with_precision (geometry , int) —> geometry`

8.5.131.2 Result:

Rounds off the value of left-hand operand to the precision given by the value of right-hand operand Rounds off the ordinates of the left-hand point to the precision given by the value of right-hand operand A geometry corresponding to the rounding of points of the operand considering a given precison.

8.5.131.3 Examples:

```
float var0 <- 12345.78943 with_precision 2; // var0 equals 12345.79
float var1 <- 123 with_precision 2; // var1 equals 123.00
point var2 <- {12345.78943, 12345.78943, 12345.78943} with_precision 2 ; // var2 equals {12345.79, 12345.79,
geometry var3 <- self with_precision 2; // var3 equals the geometry resulting from the rounding of points of
```

8.5.131.4 See also:

round,

8.5.132 with_values

8.5.132.1 Possible use:

- `predicate with_values map —> predicate`
- `with_values (predicate , map) —> predicate`

8.5.132.2 Result:

change the parameters of the given predicate

8.5.132.3 Examples:

```
predicate with_values ["time"::10]
```

8.5.133 with_weights

8.5.133.1 Possible use:

- graph **with_weights** list —> graph
- **with_weights** (graph , list) —> graph
- graph **with_weights** map —> graph
- **with_weights** (graph , map) —> graph

8.5.133.2 Result:

returns the graph (left-hand operand) with weight given in the map (right-hand operand).

8.5.133.3 Comment:

this operand re-initializes the path finder

8.5.133.4 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.foo
```

8.5.134 without_holes

8.5.134.1 Possible use:

- **without_holes** (geometry) —> geometry

8.5.134.2 Result:

A geometry corresponding to the operand geometry (geometry, agent, point) without its holes

8.5.134.3 Examples:

```
geometry var0 <- solid(self); // var0 equals the geometry corresponding to the geometry of the agent applying
```

8.5.135 writable

8.5.135.1 Possible use:

- `file` **`writable`** `bool` —> `file`
- **`writable`** (`file` , `bool`) —> `file`

8.5.135.2 Result:

Marks the file as read-only or not, depending on the second boolean argument, and returns the first argument

8.5.135.3 Comment:

A file is created using its native flags. This operator can change them. Beware that this change is system-wide (and not only restrained to GAMA): changing a file to read-only mode (e.g. “writable(f, false)”)

8.5.135.4 Examples:

```
file var0 <- shape_file("../images/point_eau.shp") writable false; // var0 equals returns a file in read-onl
```

8.5.135.5 See also:

file,

8.5.136 xml_file

8.5.136.1 Possible use:

- **`xml_file`** (`string`) —> `file`

8.5.136.2 Result:

Constructs a file of type xml. Allowed extensions are limited to xml

---

8.5.137 xor

8.5.137.1 Possible use:

- `bool` **`xor`** `bool` —> `bool`
- **`xor`** (`bool` , `bool`) —> `bool`

8.5.137.2 Result:

a bool value, equal to the logical xor between the left-hand operand and the right-hand operand. False when they are equal

8.5.137.3 Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like 1 xor 0 is accepted and returns true.

8.5.137.4 See also:

or, and, !,

8.5.138 `years_between`

8.5.138.1 Possible use:

- `date` **`years_between`** `date` —> `int`
- **`years_between`** (`date` , `date`) —> `int`

8.5.138.2 Result:

Provide the exact number of years between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

8.5.138.3 Examples:

```
int var0 <- years_between(date('2000-01-01'), date('2010-01-01')); // var0 equals 10
```