GAMA 1.8 - User Manual for Beginners

Srirama Bhamidipati

Updated: 2018-07-18 06:42:39 (Amsterdam)

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:

Report Errors

If you find any problems/typos/errors related to this book, you can report issues on this page. On this page you can click the big green button “New Issue” and fill in the details. Typically, you should fill

- name of the book
- a link to the book
- page or section number that has the error

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

Learn GAML (Beginner -I)

If you are a beginner, the next 4 chapters will introduce you to the GAML language. To learn the language, follow this recommended sequence:

- Literals
- Types or Data Types
- File Types
- Pseudo-variables

Chapter 4

Literals

(some literal expressions are also described in data types)

A literal is a way to specify an unnamed constant value corresponding to a given data type. GAML supports various types of literals for often — or less often — used data types.

4.1 Table of contents

4.2 Simple Types

Values of simple (i.e. not composed) types can all be expressed using literal expressions. Namely:

- **bool**: `true` and `false`.
- **int**: decimal value, such as `100`, or hexadecimal value if preceded by `'#'` (e.g. `#AAAAAA`, which returns the int `11184810`)
- **float**: the value in plain digits, using `'.'` for the decimal point (e.g. `123.297`)
- **string**: a sequence of characters enclosed between quotes (`'my string'`) or double quotes (`"my string"`)

4.3 Literal Constructors

Although they are not strictly literals in the sense given above, some special constructs (called *literal constructors*) allow the modeler to declare constants of other data types. They are actually operators but can be thought of literals when used with constant operands.

- **pair**: the key and the value separated by `::` (e.g. `12::'abc'`)
- **list**: the elements, separated by commas, enclosed inside square brackets (e.g. `[12,15,15]`)
- **map**: a list of pairs (e.g. `[12::'abc', 13::'def']`)
- **point**: 2 or 3 int or float ordinates enclosed inside curly brackets (e.g. `{10.0,10.0,10.0}`)

4.4 Universal Literal

Finally, a special literal, of type `unknown`, is shared between the data types and all the agent types (aka species). Only `bool`, `int` and `float`, which do not derive from `unknown`, do not accept this literal. All the others will accept it (e.g. `string s <- nil;` is ok).

- **unknown**: `nil`, which represents the non-initialized (or, literally, *unknown*) value.

Chapter 5

Types

A variable's or expression's *type* (or *data type*) determines the values it can take, plus the operations that can be performed on or with it. GAML is a statically-typed language, which means that the type of an expression is always known at compile time, and is even enforced with casting operations. There are 4 categories of types:

- primitive types, declared as keyword in the language,
- complex types, also declared as keyword in the language,
- parametric types, a refinement of complex types (mainly children of container) that is dynamically constructed using an enclosing type, a contents type and a key type,
- species types, dynamically constructed from the species declarations made by the modeler (and the built-in species present).

The hierarchy of types in GAML (only primitive and complex types are displayed here, of course, as the other ones are model-dependent) is the following:

5.1 Table of contents

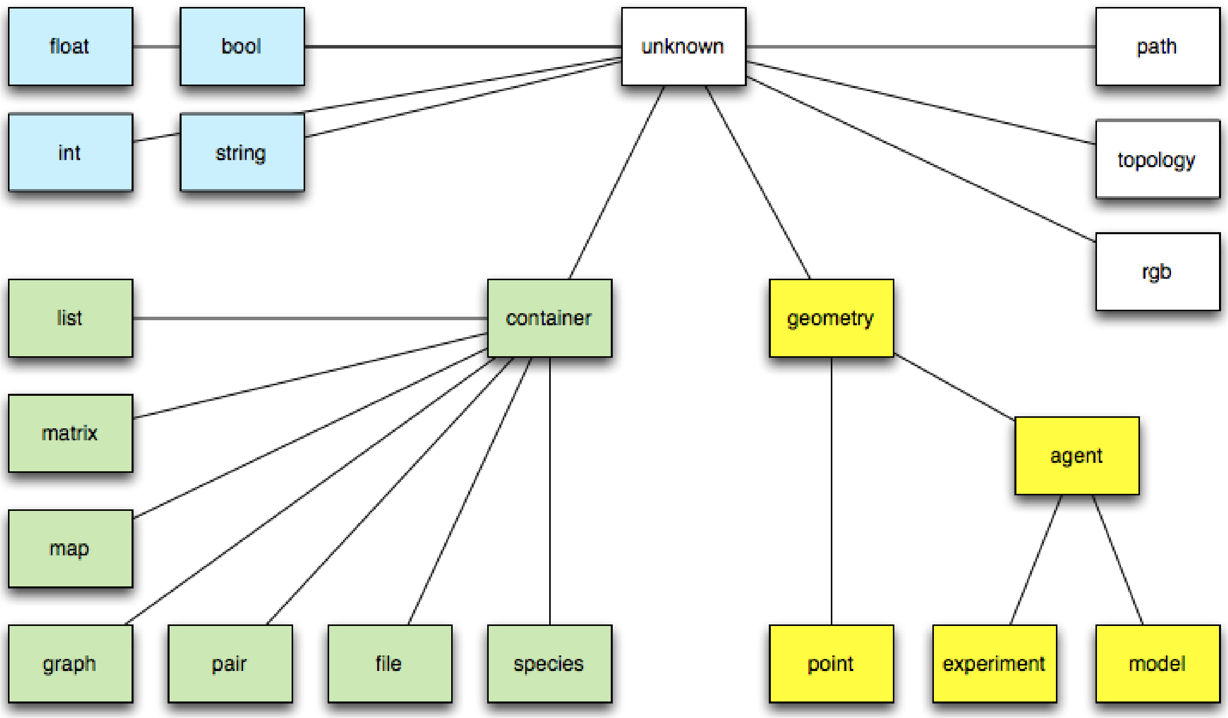

Figure 5.1: images/types_hierarchy.png

    * Species names as types
    * topology
  - Defining custom types

5.2 Primitive built-in types

5.2.1 bool

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

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

Top of the page

5.2.2 float

- **Definition:** primitive datatype holding floating point values, its absolute value is comprised between 4.9E-324 and 1.8E308.
- **Comments:** this datatype is internally backed up by the Java double datatype.
- **Litteral declaration:** decimal notation 123.45 or exponential notation 123e45 are supported.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to float to their operand. Using it is a way to obtain a float constant.

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

Top of the page

5.2.3 int

- **Definition:** primitive datatype holding integer values comprised between -2147483648 and 2147483647 (i.e. between `-2^31` and `2^31 - 1`.
- **Comments:** this datatype is internally backed up by the Java int datatype.
- **Litteral declaration:** decimal notation like 1, 256790 or hexadecimal notation like #1209FF are automatically interpreted.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to int to their operand. Using it is a way to obtain an integer constant.

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

Top of the page

5.2.4 string

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

Top of the page

5.3 Complex built-in types

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

```
complex_type nom_var <- init_var;
ltype_attr attr_var <- nom_var.attr_name;
```

For example:

```
file fileText <- file("../data/cell.Data");
bool fileTextReadable <- fileText.readable;
```

5.3.1 agent

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

Top of the page

5.3.2 container

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

```
container c  <- [1,2,3];
container c  <- matrix [[1,2,3],[4,5,6]];
container c  <- map ["x"::5, "y"::12];
container c  <- list species1;
```

Top of the page

5.3.3 file

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

```
folder(a_string)  // returns a file managing a existing folder
file(a_string) // returns any kind of file in read-only mode
read(text(a_string)) // returns a text file in read-only mode
read(image(a_string)) // does the same with an image file.
```

```
write(properties(a_string)) // returns a property file which is available for writing
                           // (if it exists, contents will be appended unless it is cleared
                             // using the standard container operations).
```

Top of the page

5.3.4 geometry

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

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

Top of the page

5.3.5 graph

- **Definition:** a datatype that represents a graph composed of vertices linked by edges.
- **Built-in attributes:**
  - edges(type = list of agent/geometry): the list of all edges
  - vertices(type = list of agent/geometry): the list of all vertices
  - circuit (type = path): an approximate minimal traveling salesman tour (hamiltonian cycle)
  - spanning_tree (type = list of agent/geometry): minimum spanning tree of the graph, i.e. a sub-graph such as every vertex lies in the tree, and as much edges lies in it but no cycles (or loops) are formed.
  - connected(type = bool): test whether the graph is connected
- **Remark:**
  - graphs are also a particular kind of container and can thus be manipulated using the container operators and commands.
  - This algorithm used to compute the circuit requires that the graph be complete and the triangle inequality exists (if x,y,z are vertices then d(x,y)+d(y,z)<d(x,z) for all x,y,z) then this algorithm will guarantee a hamiltonian cycle such that the total weight of the cycle is less than or equal to double the total weight of the optimal hamiltonian cycle.

- The computation of the spanning tree uses an implementation of the Kruskal's minimum spanning tree algorithm. If the given graph is connected it computes the minimum spanning tree, otherwise it computes the minimum spanning forest.

- **See also:** Graph operators
- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create road from: shape_file_road;
graph the_graph <- as_edge_graph(road);

graph([1,9,5])        --: ([1: in[] + out[], 5: in[] + out[], 9: in[] + out[]], [])
graph([node(0), node(1), node(2)]      // if node is a species
graph(['a'::345, 'b'::13])  --:  ([b: in[] + out[b::13], a: in[] + out[a::345], 13: in[b::13] + out[], 345: i
graph(a_graph)  --: a_graph
graph(node1)    --: null
```

Top of the page

5.3.6 list

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

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

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

Top of the page

5.3.7 map

- **Definition:** a composite datatype holding an ordered collection of pairs (a key, and its associated value).
- **Built-in attributes:**
  - keys (type = list): the list of all keys
  - values (type = list): the list of all values
  - pairs (type = list of pairs): the list of all pairs key::value
- **Comments:** maps are more or less equivalent to instances of Hashtable in Java (although they are backed up by a specific class).
- **Remark:** maps can contain values of any datatypes, including other maps or lists. Maps are also a particular kind of container and can thus be manipulated using the container operators and commands.

- **Litteral declaration:** a set of pair expressions separated by commas, enclosed in square brackets; each pair is represented by a key and a value sperarated by '::'. An example of map is [agentA::'big', agentB::'small', agentC::'big']. An empty map is noted .
- **Other declarations:** lists can be built litteraly from a point, or a string, or any other element by using the map casting operator.

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

Top of the page

5.3.8 matrix

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

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

Top of the page

5.3.9 pair

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

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

Top of the page

5.3.10 path

- **Definition:** a datatype representing a path linking two agents or geometries in a graph.
- **Built-in attributes:**
  - source (type = point): the source point, i.e. the first point of the path
  - target (type = point): the target point, i.e. the last point of the path
  - graph (type = graph): the current topology (in the case it is a spatial graph), null otherwise
  - edges (type = list of agents/geometries) : the edges of the graph composing the path
  - vertices (type = list of agents/geometries) : the vertices of the graph composing the path
  - segments (type = list of geometries): the list of the geometries composing the path
  - shape (type = geometry) : the global geometry of the path (polyline)
- **Comments:** the path created between two agents/geometries or locations will strongly depends on the topology in which it is created.
- **Remark:** a path is **immutable**, i.e. it can not be modified after it is created.
- **Declaration:** paths are very barely defined litterally. We can nevertheless use the `path` unary operator on a list of points to build a path. Operators dedicated to the computation of paths (such as path_to or path_between) are often used to build a path.

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

geometry rect <- rectangle(5);
geometry poly <- polygon([{10,20},{11,21},{10,21},{11,22}]);
path pa <- rect path_to poly;  // built a path between rect and poly, in the topolopy
                              // of the current agent (i.e. a line in a& continuous topology,
                                               // a path in a graph  in a graph topology )

a_topology path_between a_container_of_geometries // idem with an explicit topology and the possiblity
                                                   // to have more than 2 geometries
                                                  // (the path is then built incrementally)


path_between (a_graph, a_source, a_target) // idem with a the given graph as topology
```

Top of the page

5.3.11 point

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

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

Top of the page

5.3.12 rgb

- **Definition:** a datatype that represents a color in the RGB space.
- **Built-in attributes:**
  - red(type = int): the red component of the color
  - green(type = int): the green component of the color
  - blue(type = int): the blue component of the color
  - darker(type = rgb): a new color that is a darker version of this color
  - brighter(type = rgb): a new color that is a brighter version of this color
- **Remark:** rgbs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** there exist lot of ways to declare a color. We use the `rgb` casting operator applied to:
  - a string. The allowed color names are the constants defined in the Color Java class, i.e.: black, blue, cyan, darkGray, lightGray, gray, green, magenta, orange, pink, red, white, yellow.
  - a list. The integer value associated to the three first elements of the list are used to define the three red (element 0 of the list), green (element 1 of the list) and blue (element 2 of the list) components of the color.
  - a map. The red, green, blue compoenents take the value associated to the keys "r", "g", "b" in the map.
  - an integer <- the decimal integer is translated into a hexadecimal <- OxRRGGBB. The red (resp. green, blue) component of the color take the value RR (resp. GG, BB) translated in decimal.
  - Since GAMA 1.6.1, colors can be directly obtained like units, by using the ° or # symbol followed by the name in lowercase of one of the 147 CSS colors (see http://www.cssportal.com/css3-color-names/).
- **Declaration:**

```
rgb cssRed <- #red;   // Since 1.6.1
rgb testColor <- rgb('white');                 // rgb [255,255,255]
rgb test <- rgb(3,5,67);                      // rgb [3,5,67]
rgb te <- rgb(340);                            // rgb [0,1,84]
rgb tete <- rgb(["r"::34, "g"::56, "b"::345]); // rgb [34,56,255]
```

Top of the page

5.3.13 species

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

Top of the page

5.3.14 Species names as types

Once a species has been declared in a model, it automatically becomes a datatype. This means that : * It can be used to declare variables, parameters or constants, * It can be used as an operand to commands or operators that require species parameters, * It can be used as a casting operator (with the same capabilities as the built-in type agent)

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

```
global {
    init {
         create human number: 10;
         ask human {
               friend <- one_of (human - self);
         }
     }
}
entities {
    species human {
        human friend <- nil;
    }
}
```

Top of the page

5.3.15 topology

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

Top of the page

5.4 Defining custom types

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

```
species bottle {
    float volume <- 0.0 max:1 min:0.0;
    bool is_empty -> {volume = 0.0};
    action fill {
         volume <- 1.0;
    }
}
```

How to use this species to declare new bottles :

```
create bottle {
    volume <- 0.5;
}
```

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

```
species drinker {
     ...
    bottle my_bottle<- nil;
    float quantity <- rnd (100) / 100;
    bool thirsty <- false update: flip (0.1);
    ...
    action drink {
         if condition: ! bottle.is_empty {
              bottle.volume <-bottle.volume - quantity;
              thirsty <- false;
         }
    }
    ...
    init {
          create bottle return: created_bottle;
              volume <- 0.5;
          }
          my_bottle <- first(created_bottle);
    }
    ...
    reflex filling_bottle when: bottle.is_empty {
         ask  my_bottle {
              do fill;
         }
    }
    ...
    reflex drinking when: thirsty {
         do drink;
    }
}
```

Top of the page

Chapter 6

File Types

GAMA provides modelers with a generic type for files called **file**. It is possible to load a file using the *file* operator:

```
file my_file <- file("../includes/data.csv");
```

However, internally, GAMA makes the difference between the different types of files. Indeed, for instance:

```
global {
    init {
        file my_file <- file("../includes/data.csv");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
0.2
Iris-setosa
4.9
3.0
1.4
0.2
Iris-setosa
...
```

Indeed, the content of CSV file is a matrix: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default ",").

In contrary:

```
global {
    init {
```

```
        file my_file <- file("../includes/data.shp");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
```

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile.

In order to know how to load a file, GAMA analyzes its extension. For instance for a file with a ".csv" extension, GAMA knows that the file is a **csv** one and will try to split each line with the , separator. However, if the modeler wants to split each line with a different separator (for instance **;**) or load it as a text file, he/she will have to use a specific file operator.

Indeed, GAMA integrates specific operators corresponding to different types of files.

6.1 Table of contents

6.2 Text File

6.2.1 Extensions

Here the list of possible extensions for text file: * "txt" * "data" * "csv" * "text" * "tsv" * "xml"

Note that when trying to define the type of a file with the default file loading operator (**file**), GAMA will first try to test the other type of file. For example, for files with ".csv" extension, GAMA will cast them as csv file and not as text file.

6.2.2 Content

The content of a text file is a list of string corresponding to each line of the text file. For example:

```
global {
    init {
        file my_file <- text_file("../includes/data.txt");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength,sepalwidth,petallength,petalwidth,type
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
```

6.2.3 Operators

List of operators related to text files: * **text_file(string path)**: load a file (with an authorized extension) as a text file. * **text_file(string path, list content)**: load a file (with an authorized extension) as a text file and fill it with the given content. * **is_text(op)**: tests whether the operand is a text file

6.3 CSV File

6.3.1 Extensions

Here the list of possible extensions for csv file: * "csv" * "tsv"

6.3.2 Content

The content of a csv file is a matrix of string: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default ","). For example:

```
global {
    init {
        file my_file <- csv_file("../includes/data.csv");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
0.2
Iris-setosa
4.9
3.0
1.4
0.2
Iris-setosa
...
```

6.3.3 Operators

List of operators related to csv files: * **csv_file(string path)**: load a file (with an authorized extension) as a csv file with default separator (","). * **csv_file(string path, string separator)**: load a file (with an authorized extension) as a csv file with the given separator.

```
file my_file <- csv_file("../includes/data.csv", ";");
```

- **csv_file(string path, matrix content)**: load a file (with an authorized extension) as a csv file and fill it with the given content.
- **is_csv(op)**: tests whether the operand is a csv file

6.4 Shapefile

Shapefiles are classical GIS data files. A shapefile is not simple file, but a set of several files (source: wikipedia): * Mandatory files : * .shp — shape format; the feature geometry itself * .shx — shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly * .dbf — attribute format; columnar attributes for each shape, in dBase IV format

- Optional files :
  - .prj — projection format; the coordinate system and projection information, a plain text file describing the projection using well-known text format
  - .sbn and .sbx — a spatial index of the features
  - .fbn and .fbx — a spatial index of the features for shapefiles that are read-only
  - .ain and .aih — an attribute index of the active fields in a table
  - .ixs — a geocoding index for read-write shapefiles
  - .mxs — a geocoding index for read-write shapefiles (ODB format)
  - .atx — an attribute index for the .dbf file in the form of shapefile.columnname.atx (ArcGIS 8 and later)
  - .shp.xml — geospatial metadata in XML format, such as ISO 19115 or other XML schema
  - .cpg — used to specify the code page (only for .dbf) for identifying the character encoding to be used

More details about shapefiles can be found here.

6.4.1 Extensions

Here the list of possible extension for shapefile: * "shp"

6.4.2 Content

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile. For example:

```
global {
    init {
        file my_file <- shape_file("../includes/data.shp");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
...
```

Note that the attributes of each object of the shapefile is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of a corresponding attributes.

For example:

```
file my_file <- shape_file("../includes/data.shp");
write "my_file: " + my_file.contents;
loop el over: my_file {
    write (el get "TYPE");
}
```

6.4.3 Operators

List of operators related to shapefiles: * **shape_file(string path)**: load a file (with an authorized extension) as a shapefile with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). * **shape_file(string path, string code)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/) * **shape_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/)

```
file my_file <- shape_file("../includes/data.shp", "EPSG:32601");
```

- **shape_file(string path, list content)**: load a file (with an authorized extension) as a shapefile and fill it with the given content.
- **is_shape(op)**: tests whether the operand is a shapefile

6.5 OSM File

OSM (Open Street Map) is a collaborative project to create a free editable map of the world. The data produced in this project (OSM File) represent physical features on the ground (e.g., roads or buildings) using tags attached to its basic data structures (its nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way or relation (source: openstreetmap.org).

More details about OSM data can be found here.

6.5.1 Extensions

Here the list of possible extension for shapefile: * "osm" * "pbf" * "bz2" * "gz"

6.5.2 Content

The content of a OSM data is a list of geometries corresponding to the objects of the OSM file. For example:

```
global {
    init {
        file my_file <- osm_file("../includes/data.gz");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Point
Point
Point
```

```
Point
Point
LineString
LineString
Polygon
Polygon
Polygon
...
```

Note that like for shapefiles, the attributes of each object of the osm file is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of a corresponding attributes.

6.5.3 Operators

List of operators related to osm file: * **osm_file(string path)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, string code)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry.

```
file my_file <- osm_file("../includes/data.gz", "EPSG:32601");
```

- **osm_file(string path, map filter)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, only the elements with the defined values are loaded from the file. "' //map used to filter the object to build from the OSM file according to attributes. map filtering <- map(["highway"::["primary", "secondary", "tertiary", "motorway", "living_street","residential", "unclassified"], "building"::["yes"]]);

//OSM file to load file osmfile <- file

Chapter 7

Pseudo-variables

The expressions known as **pseudo-variables** are special read-only variables that are not declared anywhere (at least not in a species), and which represent a value that changes depending on the context of execution.

7.1 Table of contents

7.2 self

The pseudo-variable `self` always holds a reference to the agent executing the current statement.

- Example (sets the `friend` attribute of another random agent of the same species to `self` and conversely):

```
friend potential_friend <- one_of (species(self) - self);
if potential_friend != nil {
    potential_friend.friend <- self;
    friend <- potential_friend;
}
```

7.3 super

The pseudo-variable `super` behaves exactly in the same way as `self` except when calling an action, in which case it represents an indirection to the parent species. It is mainly used for allowing to call inherited actions within redefined ones. For instance:

```
species parent {

    int add(int a, int b) {
        return a + b;
```

```
    }

}

species child parent: parent {

    int add(int a, int b) {
        // Calls the action defined in 'parent' with modified arguments
        return super.add(a + 20, b + 20);
    }

}
```

7.4 myself

`myself` plays the same role as `self` but in remotely-executed code (`ask`, `create`, `capture` and `release` statements), where it represents the *calling* agent when the code is executed by the *remote* agent.

- Example (asks the first agent of my species to set its color to my color):

```
ask first (species (self)){
    color <- myself.color;
}
```

- Example (create 10 new agents of the species of my species, share the energy between them, turn them towards me, and make them move 4 times to get closer to me):

```
create species (self) number: 10 {
   energy <- myself.energy / 10.0;
   loop times: 4 {
       heading <- towards (myself);
       do move;
   }
}
```

7.5 each

`each` is available only in the right-hand argument of iterators. It is a pseudo-variable that represents, in turn, each of the elements of the left-hand container. It can then take any type depending on the context.

- Example:

```
list<string> names <- my_species collect each.name;  // each is of type my_species
int max <- max(['aa', 'bbb', 'cccc'] collect length(each)); // each is of type string
```

Chapter 8

Learn GAML (Beginner -II)

If you are a beginner, the next 7 chapters will introduce you to functions and statements in GAML language. Before you read these chapters it is important you know what are data types. To learn the language, follow this recommended sequence:

- Operators (9-15) : This includes 6 chapters introducing you to all the operators (A to Z) in GAMA. Go first to the chapter **opeators by categories** to get a feel of the scope of operators available. Operators are typically like functions or methods in other languages. They accept one or more arguments of the basic or complex data types and return a result in one of the data types.
- Statements (16) : Statement is a one-line sequence of keywords (commands) guided with controlling arguments (facets) that operate on one of the data types or a combination of operators and data types.

Chapter 9

Operators by categories

9.0.1 3D

box, cone3D, cube, cylinder, dem, hexagon, pyramid, rgb_to_xyz, set_z, sphere, teapot,

---

9.0.2 Arithmetic operators

-, /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,

---

9.0.3 BDI

and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, or, set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_lifetime, with_values,

---

9.0.4 Casting operators

as, as_int, as_matrix, font, is, is_skill, list_with, matrix_with, species, to_gaml, topology,

---

9.0.5 Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, grayscale, hsb, mean, median, rgb, rnd_color, sum,

---

9.0.6 Comparison operators

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

9.0.7 Containers-related operators

-, ::, +, accumulate, among, at, collect, contains, contains_all, contains_any, count, distinct, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_at, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, one_of, product_of, range, reverse, shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

---

9.0.8 Date-related operators

-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours, minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours, plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,

---

9.0.9 Dates

9.0.10 DescriptiveStatistics

auto_correlation, correlation, covariance, durbin_watson, kurtosis, moment, quantile, quantile_inverse, rank_interpolated, rms, skew, variance,

---

9.0.11 Displays

horizontal, stack, vertical,

9.0.12 Distributions

binomial_coeff, binomial_complemented, binomial_sum, chi_square, chi_square_complemented, gamma_distribution, gamma_distribution_complemented, normal_area, normal_density, normal_inverse, pValue_for_fStat, pValue_for_tStat, student_area, student_t_inverse,

---

9.0.13 Driving operators

as_driving_graph,

9.0.14 edge

edge_between, strahler,

9.0.15 EDP-related operators

diff, diff2, internal_zero_order_equation,

9.0.16 Files-related operators

crs, evaluate_sub_model, file, file_exists, folder, get, load_sub_model, new_folder, osm_file, read, step_sub_model, writable,

---

9.0.17 FIPA-related operators

conversation, message,

9.0.18 GamaMetaType

type_of,

9.0.19 GammaFunction

beta, gamma, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, log_gamma,

---

9.0.20 Graphs-related operators

add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_watts_strogatz, grid_cells_to_graph, in_degree_of, in_edges_of, layout, load_graph_from_file, load_shortest_paths, main_connected_component, max_flow_between, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_from, rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of, undirected, use_cache, weight_of, with_optimizer_type, with_weights,

---

9.0.21 Grid-related operators

as_4_grid, as_grid, as_hexagonal_grid, grid_at, path_between,

---

9.0.22 Iterator operators

accumulate, as_map, collect, count, create_map, distribution_of, distribution_of, distribution_of, distribution2d_of, distribution2d_of, distribution2d_of, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, product_of, sort_by, sum_of, variance_of, where, with_max_of, with_min_of,

---

9.0.23 List-related operators

copy_between, index_of, last_index_of,

9.0.24 Logical operators

:, !, ?, and, or, xor,

9.0.25 Map comparaison operators

fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,

---

9.0.26 Map-related operators

as_map, create_map, index_of, last_index_of,

9.0.27 Material

material,

9.0.28 Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

---

9.0.29 multicriteria operators

electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM, weighted_means_DM,

---

9.0.30 Path-related operators

agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, max_flow_between, path_between, path_to, paths_between, use_cache,

---

9.0.31 Points-related operators

-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid, closest_points_with, farthest_point_to, grid_at, norm, point, points_along, points_at, points_on,

---

9.0.32 Random operators

binomial, flip, gauss, improved_generator, open_simplex_generator, poisson, rnd, rnd_choice, sample, shuffle, simplex_generator, skew_gauss, truncated_gauss,

---

9.0.33 ReverseOperators

saveSimulation, serialize, serializeAgent, unSerializeSimulation, unSerializeSimulationFromFile,

---

9.0.34 Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

---

9.0.35 Spatial operators

-, *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, angle_between, any_location_in, arc, around, as_4_grid, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covers, cross, crosses, crs, CRS_transform, cube, curve, cylinder, dem, direction_between, disjoint_from, distance_between, distance_to, ellipse, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, line, link, masked_by, moran, neighbors_at, neighbors_of, overlapping, overlaps, partially_overlaps, path_between, path_to, plan, points_along, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rgb_to_xyz, rotated_by, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_squares, to_sub_geometries, touches, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

---

9.0.36 Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

9.0.37 Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, at_distance, closest_to, farthest_to, inside, neighbors_at, neighbors_of, overlapping,

---

9.0.38 Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

---

9.0.39 Spatial statistical operators

hierarchical_clustering, simple_clustering_by_distance,

9.0.40 Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, rotated_by, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

---

9.0.41 Species-related operators

index_of, last_index_of, of_generic_species, of_species,

9.0.42 Statistical operators

build, corR, dbscan, distribution_of, distribution2d_of, dtw, frequency_of, gamma_rnd, geometric_mean, gini, harmonic_mean, hierarchical_clustering, kmeans, kurtosis, max, mean, mean_deviation, meanR, median, min, moran, mul, predict, simple_clustering_by_distance, skewness, split, split_in, split_using, standard_deviation, sum, variance,

---

9.0.43 Strings-related operators

+, <, <=, >, >=, at, char, contains, contains_all, contains_any, copy_between, date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

---

9.0.44 System

., command, copy, dead, eval_gaml, every, is_error, is_warning, user_input,

---

9.0.45 Time-related operators

date, string,

9.0.46 Types-related operators

9.0.47 User control operators

user_input,

Chapter 10

Operators (A to A)

10.1 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.

10.2 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.

10.3 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).

10.4 Operators

10.4.1 -

10.4.1.1 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`

10.4.1.2 Result:

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

10.4.1.3 Comment:

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

10.4.1.4 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]])
```

10.4.1.5 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]
```

10.4.1.6 See also:

+, *, /, -, milliseconds_between,

10.4.2 :

10.4.2.1 Possible use:

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

10.4.2.2 See also:

?,

10.4.3 ::

10.4.3.1 Possible use:

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

10.4.3.2 Result:

produces a new pair combining the left and the right operands

10.4.3.3 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

---

10.4.4 !

10.4.4.1 Possible use:

- `!` (`bool`) —> `bool`

10.4.4.2 Result:

opposite boolean value.

10.4.4.3 Special cases:

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

10.4.4.4 Examples:

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

10.4.4.5 See also:

bool, and, or,

10.4.5 `!=`

10.4.5.1 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`

10.4.5.2 Result:

true if both operands are different, false otherwise

10.4.5.3 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
```

10.4.5.4 See also:

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

10.4.6 ?

10.4.6.1 Possible use:

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

10.4.6.2 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 :

10.4.6.3 Comment:

These functional tests can be combined together.

10.4.6.4 Examples:

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

10.4.6.5 See also:

:,

10.4.7 /

10.4.7.1 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`

10.4.7.2 Result:

Returns the division of the two operands.

10.4.7.3 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])
```

10.4.7.4 See also:

*, +, -,

10.4.8 .

10.4.8.1 Possible use:

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

10.4.8.2 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.

10.4.8.3 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]])
```

10.4.9 ^

10.4.9.1 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`

10.4.9.2 Result:

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

10.4.9.3 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
```

10.4.9.4 Examples:

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

10.4.9.5 See also:

*, sqrt,

10.4.10 `@`

Same signification as at

10.4.11 `*`

10.4.11.1 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`

10.4.11.2 Result:

Returns the product of the two operands.

10.4.11.3 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_prec
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
```

10.4.11.4 Examples:

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

10.4.11.5 See also:

/, +, -,

10.4.12 +

10.4.12.1 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`

10.4.12.2 Result:

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

10.4.12.3 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])
```

10.4.12.4 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]
```

10.4.12.5 See also:

-, *, /,

10.4.13 <

10.4.13.1 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

10.4.13.2 Result:

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

10.4.13.3 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
```

10.4.13.4 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
```

10.4.13.5 See also:

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

10.4.14 `<=`

10.4.14.1 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`

10.4.14.2 Result:

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

10.4.14.3 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
```

10.4.14.4 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
```

10.4.14.5 See also:

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

10.4.15 <>

Same signification as !=

10.4.16 =

10.4.16.1 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`

10.4.16.2 Result:

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

10.4.16.3 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
```

10.4.16.4 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
```

10.4.16.5 See also:

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

10.4.17 >

10.4.17.1 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`

10.4.17.2 Result:

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

10.4.17.3 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
```

10.4.17.4 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
```

10.4.17.5 See also:

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

10.4.18 >=

10.4.18.1 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`

10.4.18.2 Result:

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

10.4.18.3 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
```

10.4.18.4 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
```

10.4.18.5 See also:

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

10.4.19 `abs`

10.4.19.1 Possible use:

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

10.4.19.2 Result:

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

10.4.19.3 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
```

10.4.20 `accumulate`

10.4.20.1 Possible use:

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

10.4.20.2 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

10.4.20.3 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.

10.4.20.4 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]
```

10.4.20.5 See also:

collect,

10.4.21 `acos`

10.4.21.1 Possible use:

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

10.4.21.2 Result:

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

10.4.21.3 Special cases:

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

10.4.21.4 Examples:

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

10.4.21.5 See also:

asin, atan, cos,

10.4.22 `action`

10.4.22.1 Possible use:

- **`action`** (`any`) —> `action`

10.4.22.2 Result:

Casts the operand into the type action

10.4.23 `add_days`

Same signification as plus_days

10.4.24 `add_edge`

10.4.24.1 Possible use:

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

10.4.24.2 Result:

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

10.4.24.3 Comment:

if the edge already exists, the graph is unchanged

10.4.24.4 Examples:

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

10.4.24.5 See also:

add_node, graph,

10.4.25 `add_hours`

Same signification as plus_hours

10.4.26 add_minutes

Same signification as plus_minutes

10.4.27 add_months

Same signification as plus_months

10.4.28 add_ms

Same signification as plus_ms

10.4.29 add_node

10.4.29.1 Possible use:

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

10.4.29.2 Result:

adds a node in a graph.

10.4.29.3 Examples:

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

10.4.29.4 See also:

add_edge, graph,

10.4.30 add_point

10.4.30.1 Possible use:

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

10.4.30.2 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

10.4.30.3 Examples:

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

10.4.31 add_seconds

Same signification as +

10.4.32 add_weeks

Same signification as plus_weeks

10.4.33 add_years

Same signification as plus_years

10.4.34 adjacency

10.4.34.1 Possible use:

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

10.4.34.2 Result:

adjacency matrix of the given graph.

10.4.35 after

10.4.35.1 Possible use:

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

10.4.35.2 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

10.4.35.3 Examples:

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

10.4.36 agent

10.4.36.1 Possible use:

- **agent** (any) —> agent

10.4.36.2 Result:

Casts the operand into the type agent

10.4.37 agent_closest_to

10.4.37.1 Possible use:

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

10.4.37.2 Result:

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

10.4.37.3 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.

10.4.37.4 Examples:

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

10.4.37.5 See also:

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

---

10.4.38 agent_farthest_to

10.4.38.1 Possible use:

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

10.4.38.2 Result:

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

10.4.38.3 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.

10.4.38.4 Examples:

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

10.4.38.5 See also:

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

---

10.4.39 agent_from_geometry

10.4.39.1 Possible use:

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

10.4.39.2 Result:

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

10.4.39.3 Special cases:

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

10.4.39.4 Examples:

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

10.4.39.5 See also:

path,

10.4.40 agents_at_distance

10.4.40.1 Possible use:

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

10.4.40.2 Result:

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

10.4.40.3 Examples:

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

10.4.40.4 See also:

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

---

10.4.41 agents_inside

10.4.41.1 Possible use:

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

10.4.41.2 Result:

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

10.4.41.3 Examples:

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

10.4.41.4 See also:

agent_closest_to, agents_overlapping, closest_to, inside, overlapping,

---

10.4.42 agents_overlapping

10.4.42.1 Possible use:

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

10.4.42.2 Result:

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

10.4.42.3 Examples:

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

10.4.42.4 See also:

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

---

10.4.43 all_pairs_shortest_path

10.4.43.1 Possible use:

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

10.4.43.2 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.

10.4.43.3 Examples:

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

10.4.44 alpha_index

10.4.44.1 Possible use:

- **alpha_index** (graph) —> float

10.4.44.2 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)

10.4.44.3 Examples:

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

10.4.44.4 See also:

beta_index, gamma_index, nb_cycles, connectivity_index,

10.4.45 among

10.4.45.1 Possible use:

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

10.4.45.2 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

10.4.45.3 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

10.4.45.4 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
```

10.4.46 **and**

10.4.46.1 Possible use:

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

10.4.46.2 Result:

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

10.4.46.3 Comment:

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

10.4.46.4 See also:

bool, or, !,

10.4.47 **and**

10.4.47.1 Possible use:

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

10.4.47.2 Result:

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

10.4.47.3 Examples:

```
predicate1 and predicate2
```

10.4.48 **angle_between**

10.4.48.1 Possible use:

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

10.4.48.2 Result:

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

10.4.48.3 Examples:

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

10.4.49 any

Same signification as one_of

10.4.50 any_location_in

10.4.50.1 Possible use:

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

10.4.50.2 Result:

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

10.4.50.3 Examples:

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

10.4.50.4 See also:

closest_points_with, farthest_point_to, points_at,

10.4.51 any_point_in

Same signification as any_location_in

10.4.52 append_horizontally

10.4.52.1 Possible use:

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

10.4.52.2 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.

10.4.52.3 Examples:

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

10.4.53 append_vertically

10.4.53.1 Possible use:

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

10.4.53.2 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.

10.4.53.3 Examples:

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

10.4.54 arc

10.4.54.1 Possible use:

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

10.4.54.2 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

10.4.54.3 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.

10.4.54.4 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.

10.4.54.5 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
```

10.4.54.6 See also:

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

---

10.4.55 around

10.4.55.1 Possible use:

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

10.4.55.2 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.

10.4.55.3 Special cases:

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

10.4.55.4 Examples:

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

10.4.55.5 See also:

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

---

10.4.56 as

10.4.56.1 Possible use:

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

10.4.56.2 Result:

casting of the first argument into a given type

10.4.56.3 Comment:

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

10.4.56.4 Examples:

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

10.4.57 as_4_grid

10.4.57.1 Possible use:

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

10.4.57.2 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)

10.4.57.3 Examples:

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

10.4.57.4 See also:

as_grid, as_hexagonal_grid,

10.4.58 `as_distance_graph`

10.4.58.1 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`

10.4.58.2 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).

10.4.58.3 Comment:

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

10.4.58.4 Examples:

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

10.4.58.5 See also:

as_intersection_graph, as_edge_graph,

10.4.59 `as_driving_graph`

10.4.59.1 Possible use:

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

10.4.59.2 Result:

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

10.4.59.3 Examples:

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

10.4.59.4 See also:

as_intersection_graph, as_distance_graph, as_edge_graph,

10.4.60 as_edge_graph

10.4.60.1 Possible use:

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

10.4.60.2 Result:

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

10.4.60.3 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
```

10.4.60.4 See also:

as_intersection_graph, as_distance_graph,

10.4.61 as_grid

10.4.61.1 Possible use:

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

10.4.61.2 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)

10.4.61.3 Examples:

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

10.4.61.4 See also:

as_4_grid, as_hexagonal_grid,

10.4.62 `as_hexagonal_grid`

10.4.62.1 Possible use:

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

10.4.62.2 Result:

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

10.4.62.3 Examples:

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

10.4.62.4 See also:

as_4_grid, as_grid,

10.4.63 `as_int`

10.4.63.1 Possible use:

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

10.4.63.2 Result:

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

10.4.63.3 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

10.4.63.4 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
```

10.4.63.5 See also:

int,

10.4.64 `as_intersection_graph`

10.4.64.1 Possible use:

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

10.4.64.2 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).

10.4.64.3 Comment:

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

10.4.64.4 Examples:

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

10.4.64.5 See also:

as_distance_graph, as_edge_graph,

10.4.65 as_map

10.4.65.1 Possible use:

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

10.4.65.2 Result:

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

10.4.65.3 Comment:

the right-hand operand should be a pair

10.4.65.4 Special cases:

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

10.4.65.5 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]
```

10.4.66 as_matrix

10.4.66.1 Possible use:

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

10.4.66.2 Result:

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

10.4.66.3 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.

10.4.66.4 Special cases:

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

10.4.66.5 See also:

matrix,

10.4.67 as_path

10.4.67.1 Possible use:

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

10.4.67.2 Result:

create a graph path from the list of shape

10.4.67.3 Examples:

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

10.4.68 asin

10.4.68.1 Possible use:

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

10.4.68.2 Result:

the arcsin of the operand

10.4.68.3 Special cases:

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

10.4.68.4 Examples:

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

10.4.68.5 See also:

acos, atan, sin,

10.4.69 `at`

10.4.69.1 Possible use:

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

10.4.69.2 Result:

the element at the right operand index of the container

10.4.69.3 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

10.4.69.4 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}
```

10.4.69.5 Examples:

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

10.4.69.6 See also:

contains_all, contains_any,

10.4.70 at_distance

10.4.70.1 Possible use:

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

10.4.70.2 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)

10.4.70.3 Examples:

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

10.4.70.4 See also:

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

---

10.4.71 at_location

10.4.71.1 Possible use:

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

10.4.71.2 Result:

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

10.4.71.3 Examples:

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

10.4.72 atan

10.4.72.1 Possible use:

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

10.4.72.2 Result:

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

10.4.72.3 Examples:

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

10.4.72.4 See also:

acos, asin, tan,

10.4.73 `atan2`

10.4.73.1 Possible use:

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

10.4.73.2 Result:

the atan2 value of the two operands.

10.4.73.3 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.

10.4.73.4 Examples:

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

10.4.73.5 See also:

atan, acos, asin,

10.4.74 `attributes`

10.4.74.1 Possible use:

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

10.4.74.2 Result:

Casts the operand into the type attributes

10.4.75 auto_correlation

10.4.75.1 Possible use:

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

10.4.75.2 Result:

Returns the auto-correlation of a data sequence

Chapter 11

Operators (B to C)

11.1 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.

11.2 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.

11.3 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).

11.4 Operators

11.4.1 BDIPlan

11.4.1.1 Possible use:

- **BDIPlan** (`any`) —> `BDIPlan`

11.4.1.2 Result:

Casts the operand into the type BDIPlan

11.4.2 before

11.4.2.1 Possible use:

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

11.4.2.2 Result:

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

11.4.2.3 Examples:

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

11.4.3 beta

11.4.3.1 Possible use:

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

11.4.3.2 Result:

Returns the beta function with arguments a, b.

11.4.4 beta_index

11.4.4.1 Possible use:

- **beta_index** (`graph`) —> `float`

11.4.4.2 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.

11.4.4.3 Examples:

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

11.4.4.4 See also:

alpha_index, gamma_index, nb_cycles, connectivity_index,

11.4.5 between

11.4.5.1 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`

11.4.5.2 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

11.4.5.3 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
```

11.4.5.4 Examples:

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

11.4.6 betweenness_centrality

11.4.6.1 Possible use:

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

11.4.6.2 Result:

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

11.4.6.3 Examples:

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

11.4.7 biggest_cliques_of

11.4.7.1 Possible use:

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

11.4.7.2 Result:

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

11.4.7.3 Examples:

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

11.4.7.4 See also:

maximal_cliques_of,

11.4.8 binomial

11.4.8.1 Possible use:

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

11.4.8.2 Result:

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

11.4.8.3 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.

11.4.8.4 Examples:

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

11.4.8.5 See also:

poisson, gauss,

11.4.9 binomial_coeff

11.4.9.1 Possible use:

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

11.4.9.2 Result:

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

11.4.10 binomial_complemented

11.4.10.1 Possible use:

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

11.4.10.2 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.

11.4.11 `binomial_sum`

11.4.11.1 Possible use:

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

11.4.11.2 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.

11.4.12 `blend`

11.4.12.1 Possible use:

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

11.4.12.2 Result:

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

11.4.12.3 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
```

11.4.12.4 Examples:

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

11.4.12.5 See also:

rgb, hsb,

11.4.13 `bool`

11.4.13.1 Possible use:

- **`bool`** (`any`) —> `bool`

11.4.13.2 Result:

Casts the operand into the type bool

11.4.14 `box`

11.4.14.1 Possible use:

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

11.4.14.2 Result:

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

11.4.14.3 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.

11.4.14.4 Special cases:

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

11.4.14.5 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=
```

11.4.14.6 See also:

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

---

11.4.15 brewer_colors

11.4.15.1 Possible use:

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

11.4.15.2 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/)

11.4.15.3 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
```

11.4.15.4 See also:

brewer_palettes,

11.4.16 brewer_palettes

11.4.16.1 Possible use:

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

11.4.16.2 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)

11.4.16.3 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
```

11.4.16.4 See also:

brewer_colors,

11.4.17 `buffer`

Same signification as +

11.4.18 `build`

11.4.18.1 Possible use:

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

11.4.18.2 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)

11.4.18.3 Examples:

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

11.4.19 `ceil`

11.4.19.1 Possible use:

- **`ceil`** (`float`) —> `float`

11.4.19.2 Result:

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

11.4.19.3 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
```

11.4.19.4 See also:

floor, round,

11.4.20 centroid

11.4.20.1 Possible use:

- **centroid** (`geometry`) —> `point`

11.4.20.2 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

11.4.20.3 Examples:

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

11.4.20.4 See also:

any_location_in, closest_points_with, farthest_point_to, points_at,

---

11.4.21 char

11.4.21.1 Possible use:

- **char** (`int`) —> `string`

11.4.21.2 Special cases:

- converts ACSII integer value to character

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

11.4.22 chi_square

11.4.22.1 Possible use:

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

11.4.22.2 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.

11.4.23 chi_square_complemented

11.4.23.1 Possible use:

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

11.4.23.2 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.

11.4.24 circle

11.4.24.1 Possible use:

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

11.4.24.2 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.

11.4.24.3 Comment:

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

11.4.24.4 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.

11.4.24.5 Examples:

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

11.4.24.6 See also:

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

---

11.4.25 clean

11.4.25.1 Possible use:

- **clean** (geometry) —> geometry

11.4.25.2 Result:

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

11.4.25.3 Comment:

The cleaning corresponds to a buffer with a distance of 0.0

11.4.25.4 Examples:

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

11.4.26 clean_network

11.4.26.1 Possible use:

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

11.4.26.2 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)

11.4.26.3 Comment:

The cleaned set of polylines

11.4.26.4 Examples:

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

11.4.27 closest_points_with

11.4.27.1 Possible use:

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

11.4.27.2 Result:

A list of two closest points between the two geometries.

11.4.27.3 Examples:

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

11.4.27.4 See also:

any_location_in, any_point_in, farthest_point_to, points_at,

11.4.28 closest_to

11.4.28.1 Possible use:

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

11.4.28.2 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).

11.4.28.3 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.

11.4.28.4 Examples:

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

11.4.28.5 See also:

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

---

11.4.29 collect

11.4.29.1 Possible use:

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

11.4.29.2 Result:

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

11.4.29.3 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.

11.4.29.4 Special cases:

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

11.4.29.5 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
```

11.4.29.6 See also:

accumulate,

11.4.30 column_at

11.4.30.1 Possible use:

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

11.4.30.2 Result:

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

11.4.30.3 Examples:

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

11.4.30.4 See also:

row_at, rows_list,

11.4.31 `columns_list`

11.4.31.1 Possible use:

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

11.4.31.2 Result:

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

11.4.31.3 Examples:

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

11.4.31.4 See also:

rows_list,

11.4.32 `command`

11.4.32.1 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`

11.4.32.2 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.

11.4.33 `cone`

11.4.33.1 Possible use:

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

11.4.33.2 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.

11.4.33.3 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.

11.4.33.4 Special cases:

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

11.4.33.5 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.
```

11.4.33.6 See also:

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

---

11.4.34 `cone3D`

11.4.34.1 Possible use:

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

11.4.34.2 Result:

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

11.4.34.3 Comment:

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

11.4.34.4 Special cases:

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

11.4.34.5 Examples:

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

11.4.34.6 See also:

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

---

11.4.35 connected_components_of

11.4.35.1 Possible use:

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

11.4.35.2 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.

11.4.35.3 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 a
```

11.4.35.4 See also:

alpha_index, connectivity_index, nb_cycles,

11.4.36 `connectivity_index`

11.4.36.1 Possible use:

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

11.4.36.2 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).

11.4.36.3 Examples:

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

11.4.36.4 See also:

alpha_index, beta_index, gamma_index, nb_cycles,

11.4.37 `container`

11.4.37.1 Possible use:

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

11.4.37.2 Result:

Casts the operand into the type container

11.4.38 `contains`

11.4.38.1 Possible use:

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

11.4.38.2 Result:

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

11.4.38.3 Comment:

the contains operator behavior depends on the nature of the operand

11.4.38.4 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
```

11.4.38.5 Examples:

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

11.4.38.6 See also:

contains_all, contains_any,

11.4.39 `contains_all`

11.4.39.1 Possible use:

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

11.4.39.2 Result:

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

11.4.39.3 Comment:

the definition of contains depends on the container

11.4.39.4 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
```

11.4.39.5 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
```

11.4.39.6 See also:

contains, contains_any,

11.4.40 `contains_any`

11.4.40.1 Possible use:

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

11.4.40.2 Result:

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

11.4.40.3 Comment:

the definition of contains depends on the container

11.4.40.4 Special cases:

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

11.4.40.5 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
```

11.4.40.6 See also:

contains, contains_all,

11.4.41 `contains_edge`

11.4.41.1 Possible use:

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

11.4.41.2 Result:

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

11.4.41.3 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
```

11.4.41.4 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
```

11.4.41.5 See also:

contains_vertex,

11.4.42 contains_vertex

11.4.42.1 Possible use:

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

11.4.42.2 Result:

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

11.4.42.3 Special cases:

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

11.4.42.4 Examples:

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

11.4.42.5 See also:

contains_edge,

11.4.43 conversation

11.4.43.1 Possible use:

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

11.4.44 convex_hull

11.4.44.1 Possible use:

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

11.4.44.2 Result:

A geometry corresponding to the convex hull of the operand.

11.4.44.3 Examples:

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

11.4.45 `copy`

11.4.45.1 Possible use:

- **`copy`** (`unknown`) —> `unknown`

11.4.45.2 Result:

returns a copy of the operand.

11.4.46 `copy_between`

11.4.46.1 Possible use:

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

11.4.46.2 Result:

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

11.4.46.3 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

11.4.46.4 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"
```

11.4.47 `corR`

11.4.47.1 Possible use:

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

11.4.47.2 Result:

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

11.4.47.3 Special cases:

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

11.4.47.4 Examples:

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

11.4.48 `correlation`

11.4.48.1 Possible use:

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

11.4.48.2 Result:

Returns the correlation of two data sequences

11.4.49 `cos`

11.4.49.1 Possible use:

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

11.4.49.2 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.

11.4.49.3 Special cases:

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

11.4.49.4 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
```

11.4.49.5 See also:

sin, tan,

11.4.50 `cos_rad`

11.4.50.1 Possible use:

- **`cos_rad`** (`float`) —> `float`

11.4.50.2 Result:

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

11.4.50.3 Special cases:

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

11.4.50.4 See also:

sin, tan,

11.4.51 `count`

11.4.51.1 Possible use:

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

11.4.51.2 Result:

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

11.4.51.3 Comment:

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

11.4.51.4 Special cases:

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

11.4.51.5 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
```

11.4.51.6 See also:

group_by,

11.4.52 `covariance`

11.4.52.1 Possible use:

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

11.4.52.2 Result:

Returns the covariance of two data sequences

11.4.53 `covers`

11.4.53.1 Possible use:

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

11.4.53.2 Result:

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

11.4.53.3 Special cases:

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

11.4.53.4 Examples:

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

11.4.53.5 See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

11.4.54 create_map

11.4.54.1 Possible use:

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

11.4.54.2 Result:

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

11.4.54.3 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

11.4.54.4 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,'
```

11.4.55 cross

11.4.55.1 Possible use:

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

11.4.55.2 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

11.4.55.3 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
```

11.4.55.4 See also:

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

---

11.4.56 `crosses`

11.4.56.1 Possible use:

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

11.4.56.2 Result:

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

11.4.56.3 Special cases:

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

11.4.56.4 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 t
```

11.4.56.5 See also:

disjoint_from, intersects, overlaps, partially_overlaps, touches,

---

11.4.57 `crs`

11.4.57.1 Possible use:

- **`crs`** (`file`) —> `string`

11.4.57.2 Result:

the Coordinate Reference System (CRS) of the GIS file

11.4.57.3 Examples:

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

11.4.58 `CRS_transform`

11.4.58.1 Possible use:

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

11.4.58.2 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
```

11.4.59 `csv_file`

11.4.59.1 Possible use:

- **`csv_file`** (`string`) —> `file`

11.4.59.2 Result:

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

---

11.4.60 `cube`

11.4.60.1 Possible use:

- **`cube`** (`float`) —> `geometry`

11.4.60.2 Result:

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

11.4.60.3 Comment:

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

11.4.60.4 Special cases:

- returns nil if the operand is nil.

11.4.60.5 Examples:

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

11.4.60.6 See also:

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

---

11.4.61 `curve`

11.4.61.1 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`

11.4.61.2 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.

11.4.61.3 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

11.4.61.4 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 compose
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
```

11.4.61.5 See also:

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

---

11.4.62 `cylinder`

11.4.62.1 Possible use:

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

11.4.62.2 Result:

A cylinder geometry which radius is equal to the operand.

11.4.62.3 Comment:

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

11.4.62.4 Special cases:

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

11.4.62.5 Examples:

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

11.4.62.6 See also:

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

Chapter 12

Operators (D to H)

12.1 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.

12.2 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.

12.3 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).

12.4 Operators

12.4.1 `date`

12.4.1.1 Possible use:

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

12.4.1.2 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

12.4.1.3 Examples:

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

12.4.2 `dbscan`

12.4.2.1 Possible use:

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

12.4.2.2 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)

12.4.2.3 Special cases:

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

12.4.2.4 Examples:

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

12.4.3 dead

12.4.3.1 Possible use:

- **dead** (agent) —> bool

12.4.3.2 Result:

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

12.4.3.3 Examples:

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

12.4.4 degree_of

12.4.4.1 Possible use:

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

12.4.4.2 Result:

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

12.4.4.3 Examples:

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

12.4.4.4 See also:

in_degree_of, out_degree_of,

12.4.5 dem

12.4.5.1 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

12.4.5.2 Result:

A polygon that is equivalent to the surface of the texture

12.4.5.3 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 the
```

12.4.6 det

Same signification as determinant

12.4.7 determinant

12.4.7.1 Possible use:

- **determinant** (matrix) —> float

12.4.7.2 Result:

The determinant of the given matrix

12.4.7.3 Examples:

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

12.4.8 `diff`

12.4.8.1 Possible use:

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

12.4.8.2 Result:

A placeholder function for expressing equations

12.4.9 `diff2`

12.4.9.1 Possible use:

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

12.4.9.2 Result:

A placeholder function for expressing equations

12.4.10 `directed`

12.4.10.1 Possible use:

- **`directed`** (`graph`) —> `graph`

12.4.10.2 Result:

the operand graph becomes a directed graph.

12.4.10.3 Comment:

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

12.4.10.4 See also:

undirected,

12.4.11 direction_between

12.4.11.1 Possible use:

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

12.4.11.2 Result:

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

12.4.11.3 Examples:

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

12.4.11.4 See also:

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

---

12.4.12 direction_to

Same signification as towards

12.4.13 disjoint_from

12.4.13.1 Possible use:

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

12.4.13.2 Result:

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

12.4.13.3 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.

12.4.13.4 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
```

12.4.13.5 See also:

intersects, crosses, overlaps, partially_overlaps, touches,

12.4.14 `distance_between`

12.4.14.1 Possible use:

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

12.4.14.2 Result:

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

12.4.14.3 Examples:

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

12.4.14.4 See also:

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

---

12.4.15 `distance_to`

12.4.15.1 Possible use:

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

12.4.15.2 Result:

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

12.4.15.3 Examples:

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

12.4.15.4 See also:

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

---

12.4.16 distinct

12.4.16.1 Possible use:

- **distinct** (container) —> list

12.4.16.2 Result:

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

12.4.16.3 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]
```

12.4.16.4 Examples:

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

12.4.17 distribution_of

12.4.17.1 Possible use:

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

12.4.17.2 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)

12.4.17.3 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'
```

12.4.17.4 See also:

as_map,

12.4.18 distribution2d_of

12.4.18.1 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

12.4.18.2 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)

12.4.18.3 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'
```

12.4.18.4 See also:

as_map,

12.4.19 `div`

12.4.19.1 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`

12.4.19.2 Result:

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

12.4.19.3 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.

12.4.19.4 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
```

12.4.19.5 See also:

mod,

12.4.20 `dnorm`

Same signification as normal_density

12.4.21 `dtw`

12.4.21.1 Possible use:

- `list` **`dtw`** `list` —> `float`

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

12.4.21.2 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

12.4.21.3 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
```

12.4.22 durbin_watson

12.4.22.1 Possible use:

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

12.4.22.2 Result:

Durbin-Watson computation

12.4.23 dxf_file

12.4.23.1 Possible use:

- **dxf_file** (string) —> file

12.4.23.2 Result:

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

---

12.4.24 edge

12.4.24.1 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

12.4.25 edge_between

12.4.25.1 Possible use:

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

12.4.25.2 Result:

returns the edge linking two nodes

12.4.25.3 Examples:

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

12.4.25.4 See also:

out_edges_of, in_edges_of,

12.4.26 edge_betweenness

12.4.26.1 Possible use:

- **edge_betweenness** (graph) —> map

12.4.26.2 Result:

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

12.4.26.3 Examples:

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

12.4.27 `edges`

12.4.27.1 Possible use:

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

12.4.28 `eigenvalues`

12.4.28.1 Possible use:

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

12.4.28.2 Result:

The eigen values (matrix) of the given matrix

12.4.28.3 Examples:

```
list<float> var0 <- eigenvalues(matrix([[5,-3],[6,-4]])); // var0 equals [2.0000000000000004,-0.9999999999
```

12.4.29 `electre_DM`

12.4.29.1 Possible use:

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

12.4.29.2 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.

12.4.29.3 Special cases:

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

12.4.29.4 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, "q
```

12.4.29.5 See also:

weighted_means_DM, promethee_DM, evidence_theory_DM,

12.4.30 ellipse

12.4.30.1 Possible use:

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

12.4.30.2 Result:

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

12.4.30.3 Comment:

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

12.4.30.4 Special cases:

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

12.4.30.5 Examples:

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

12.4.30.6 See also:

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

---

12.4.31 `emotion`

12.4.31.1 Possible use:

- **`emotion`** (`any`) —> `emotion`

12.4.31.2 Result:

Casts the operand into the type emotion

12.4.32 `empty`

12.4.32.1 Possible use:

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

12.4.32.2 Result:

true if the operand is empty, false otherwise.

12.4.32.3 Comment:

the empty operator behavior depends on the nature of the operand

12.4.32.4 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
```

12.4.33 enlarged_by

Same signification as +

12.4.34 envelope

12.4.34.1 Possible use:

- **envelope** (unknown) —> geometry

12.4.34.2 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

---

12.4.35 eval_gaml

12.4.35.1 Possible use:

- **eval_gaml** (string) —> unknown

12.4.35.2 Result:

evaluates the given GAML string.

12.4.35.3 Examples:

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

12.4.36 eval_when

12.4.36.1 Possible use:

- **eval_when** (BDIPlan) —> bool

12.4.36.2 Result:

evaluate the facet when of a given plan

12.4.36.3 Examples:

```
eval_when(plan1)
```

12.4.37 evaluate_sub_model

12.4.37.1 Possible use:

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

12.4.37.2 Result:

Load a submodel

12.4.37.3 Comment:

loaded submodel

12.4.38 even

12.4.38.1 Possible use:

- **`even`** (`int`) —> `bool`

12.4.38.2 Result:

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

12.4.38.3 Special cases:

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

12.4.38.4 Examples:

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

12.4.39 every

12.4.39.1 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

12.4.39.2 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'

12.4.39.3 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

12.4.39.4 Examples:

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

12.4.39.5 See also:

since, after, to,

12.4.40 every_cycle

Same signification as every

12.4.41 evidence_theory_DM

12.4.41.1 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

12.4.41.2 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)

12.4.41.3 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

12.4.41.4 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
```

12.4.41.5 See also:

weighted_means_DM, electre_DM,

12.4.42 exp

12.4.42.1 Possible use:

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

12.4.42.2 Result:

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

12.4.42.3 Special cases:

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

12.4.42.4 Examples:

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

12.4.42.5 See also:

ln,

12.4.43 `fact`

12.4.43.1 Possible use:

- **`fact`** (`int`) —> `float`

12.4.43.2 Result:

Returns the factorial of the operand.

12.4.43.3 Special cases:

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

12.4.43.4 Examples:

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

12.4.44 `farthest_point_to`

12.4.44.1 Possible use:

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

12.4.44.2 Result:

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

12.4.44.3 Examples:

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

12.4.44.4 See also:

any_location_in, any_point_in, closest_points_with, points_at,

---

12.4.45 farthest_to

12.4.45.1 Possible use:

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

12.4.45.2 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).

12.4.45.3 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.

12.4.45.4 Examples:

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

12.4.45.5 See also:

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

---

12.4.46 file

12.4.46.1 Possible use:

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

12.4.46.2 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.

12.4.46.3 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.

12.4.46.4 Special cases:

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

12.4.46.5 Examples:

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

12.4.46.6 See also:

folder, new_folder,

12.4.47 `file`

12.4.47.1 Possible use:

- **`file`** (`any`) —> `file`

12.4.47.2 Result:

Casts the operand into the type file

12.4.48 `file_exists`

12.4.48.1 Possible use:

- **`file_exists`** (`string`) —> `bool`

12.4.48.2 Result:

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

12.4.49 `first`

12.4.49.1 Possible use:

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

12.4.49.2 Result:

the first value of the operand

12.4.49.3 Comment:

the first operator behavior depends on the nature of the operand

12.4.49.4 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'
```

12.4.49.5 See also:

last,

12.4.50 `first_of`

Same signification as first

12.4.51 `first_with`

12.4.51.1 Possible use:

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

12.4.51.2 Result:

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

12.4.51.3 Comment:

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

12.4.51.4 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)
```

12.4.51.5 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
```

12.4.51.6 See also:

group_by, last_with, where,

12.4.52 `flip`

12.4.52.1 Possible use:

- **`flip`** (`float`) —> `bool`

12.4.52.2 Result:

true or false given the probability represented by the operand

12.4.52.3 Special cases:

- flip 0 always returns false, flip 1 true

12.4.52.4 Examples:

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

12.4.52.5 See also:

rnd,

12.4.53 `float`

12.4.53.1 Possible use:

- **`float`** (`any`) —> `float`

12.4.53.2 Result:

Casts the operand into the type float

12.4.54 `floor`

12.4.54.1 Possible use:

- **`floor`** (`float`) —> `float`

12.4.54.2 Result:

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

12.4.54.3 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
```

12.4.54.4 See also:

ceil, round,

12.4.55 folder

12.4.55.1 Possible use:

- **folder** (string) —> file

12.4.55.2 Result:

opens an existing repository

12.4.55.3 Special cases:

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

12.4.55.4 Examples:

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

12.4.55.5 See also:

file, new_folder,

12.4.56 font

12.4.56.1 Possible use:

- **font** (string, int, int) —> font

12.4.56.2 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.

12.4.56.3 Examples:

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

12.4.57 frequency_of

12.4.57.1 Possible use:

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

12.4.57.2 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)

12.4.57.3 Examples:

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

12.4.57.4 See also:

as_map,

12.4.58 from

Same signification as since

12.4.59 fuzzy_choquet_DM

12.4.59.1 Possible use:

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

12.4.59.2 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)

12.4.59.3 Special cases:

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

12.4.59.4 Examples:

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

12.4.59.5 See also:

promethee_DM, electre_DM, evidence_theory_DM,

12.4.60 `fuzzy_kappa`

12.4.60.1 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`

12.4.60.2 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

12.4.60.3 Examples:

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

12.4.61 `fuzzy_kappa_sim`

12.4.61.1 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`

12.4.61.2 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,

12.4.61.3 Examples:

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

12.4.62 gaml_file

12.4.62.1 Possible use:

- **gaml_file** (string) —> file

12.4.62.2 Result:

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

---

12.4.63 gaml_type

12.4.63.1 Possible use:

- **gaml_type** (any) —> gaml_type

12.4.63.2 Result:

Casts the operand into the type gaml_type

12.4.64 gamma

12.4.64.1 Possible use:

- **gamma** (float) —> float

12.4.64.2 Result:

Returns the value of the Gamma function at x.

12.4.65 gamma_distribution

12.4.65.1 Possible use:

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

12.4.65.2 Result:

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

12.4.65.3 Comment:

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

12.4.66 gamma_distribution_complemented

12.4.66.1 Possible use:

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

12.4.66.2 Result:

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

12.4.67 gamma_index

12.4.67.1 Possible use:

- **gamma_index** (graph) —> float

12.4.67.2 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.

12.4.67.3 Examples:

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

12.4.67.4 See also:

alpha_index, beta_index, nb_cycles, connectivity_index,

12.4.68 `gamma_rnd`

12.4.68.1 Possible use:

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

12.4.68.2 Result:

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

12.4.68.3 Examples:

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

12.4.69 `gauss`

12.4.69.1 Possible use:

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

12.4.69.2 Result:

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

12.4.69.3 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

12.4.69.4 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
```

12.4.69.5 See also:

truncated_gauss, poisson, skew_gauss,

12.4.70 `generate_barabasi_albert`

12.4.70.1 Possible use:

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

12.4.70.2 Result:

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

12.4.70.3 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:

12.4.70.4 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?

12.4.70.5 Examples:

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

12.4.70.6 See also:

generate_watts_strogatz,

12.4.71 generate_complete_graph

12.4.71.1 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

12.4.71.2 Result:

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

12.4.71.3 Comment:

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

12.4.71.4 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?

12.4.71.5 Examples:

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

12.4.71.6 See also:

generate_barabasi_albert, generate_watts_strogatz,

12.4.72 `generate_watts_strogatz`

12.4.72.1 Possible use:

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

12.4.72.2 Result:

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

12.4.72.3 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:

12.4.72.4 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?

12.4.72.5 Examples:

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

12.4.72.6 See also:

generate_barabasi_albert,

12.4.73 geojson_file

12.4.73.1 Possible use:

- **geojson_file** (string) —> file

12.4.73.2 Result:

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

---

12.4.74 geometric_mean

12.4.74.1 Possible use:

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

12.4.74.2 Result:

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

12.4.74.3 Comment:

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

12.4.74.4 Examples:

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

12.4.74.5 See also:

mean, median, harmonic_mean,

12.4.75 geometry

12.4.75.1 Possible use:

- **geometry** (any) —> geometry

12.4.75.2 Result:

Casts the operand into the type geometry

12.4.76 geometry_collection

12.4.76.1 Possible use:

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

12.4.76.2 Result:

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

12.4.76.3 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.

12.4.76.4 Examples:

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

12.4.76.5 See also:

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

---

12.4.77 get

Same signification as read

12.4.77.1 Possible use:

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

12.4.77.2 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.

12.4.77.3 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
```

12.4.78 `get_about`

12.4.78.1 Possible use:

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

12.4.78.2 Result:

get the about value of the given emotion

12.4.78.3 Examples:

```
get_about(emotion)
```

12.4.79 `get_agent`

12.4.79.1 Possible use:

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

12.4.79.2 Result:

get the agent value of the given social link

12.4.79.3 Examples:

```
get_agent(social_link1)
```

12.4.80 `get_agent_cause`

12.4.80.1 Possible use:

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

12.4.80.2 Result:

get the agent cause value of the given emotion

12.4.80.3 Examples:

```
get_agent_cause(emotion)
```

12.4.81 get_belief_op

12.4.81.1 Possible use:

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

12.4.81.2 Result:

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

12.4.81.3 Examples:

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

12.4.82 get_belief_with_name_op

12.4.82.1 Possible use:

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

12.4.82.2 Result:

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

12.4.82.3 Examples:

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

12.4.83 get_beliefs_op

12.4.83.1 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

12.4.83.2 Result:

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

12.4.83.3 Examples:

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

12.4.84 get_beliefs_with_name_op

12.4.84.1 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

12.4.84.2 Result:

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

12.4.84.3 Examples:

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

12.4.85 get_current_intention_op

12.4.85.1 Possible use:

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

12.4.85.2 Result:

get the current intention.

12.4.85.3 Examples:

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

12.4.86 get_decay

12.4.86.1 Possible use:

- **get_decay** (emotion) —> float

12.4.86.2 Result:

get the decay value of the given emotion

12.4.86.3 Examples:

```
get_decay(emotion)
```

12.4.87 get_desire_op

12.4.87.1 Possible use:

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

12.4.87.2 Result:

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

12.4.87.3 Examples:

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

12.4.88 get_desire_with_name_op

12.4.88.1 Possible use:

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

12.4.88.2 Result:

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

12.4.88.3 Examples:

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

12.4.89 get_desires_op

12.4.89.1 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

12.4.89.2 Result:

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

12.4.89.3 Examples:

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

12.4.90 get_desires_with_name_op

12.4.90.1 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

12.4.90.2 Result:

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

12.4.90.3 Examples:

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

12.4.91 get_dominance

12.4.91.1 Possible use:

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

12.4.91.2 Result:

get the dominance value of the given social link

12.4.91.3 Examples:

```
get_dominance(social_link1)
```

12.4.92 `get_familiarity`

12.4.92.1 Possible use:

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

12.4.92.2 Result:

get the familiarity value of the given social link

12.4.92.3 Examples:

```
get_familiarity(social_link1)
```

12.4.93 `get_ideal_op`

12.4.93.1 Possible use:

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

12.4.93.2 Result:

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

12.4.93.3 Examples:

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

12.4.94 `get_ideal_with_name_op`

12.4.94.1 Possible use:

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

12.4.94.2 Result:

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

12.4.94.3 Examples:

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

12.4.95 get_ideals_op

12.4.95.1 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>

12.4.95.2 Result:

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

12.4.95.3 Examples:

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

12.4.96 get_ideals_with_name_op

12.4.96.1 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

12.4.96.2 Result:

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

12.4.96.3 Examples:

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

12.4.97 get_intensity

12.4.97.1 Possible use:

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

12.4.97.2 Result:

get the intensity value of the given emotion

12.4.97.3 Examples:

```
emotion set_intensity 12
```

12.4.98 get_intention_op

12.4.98.1 Possible use:

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

12.4.98.2 Result:

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

12.4.98.3 Examples:

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

12.4.99 get_intention_with_name_op

12.4.99.1 Possible use:

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

12.4.99.2 Result:

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

12.4.99.3 Examples:

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

12.4.100 get_intentions_op

12.4.100.1 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

12.4.100.2 Result:

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

12.4.100.3 Examples:

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

12.4.101 get_intentions_with_name_op

12.4.101.1 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

12.4.101.2 Result:

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

12.4.101.3 Examples:

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

12.4.102 get_lifetime

12.4.102.1 Possible use:

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

12.4.102.2 Result:

get the lifetime value of the given mental state

12.4.102.3 Examples:

```
get_lifetime(mental_state1)
```

12.4.103 get_liking

12.4.103.1 Possible use:

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

12.4.103.2 Result:

get the liking value of the given social link

12.4.103.3 Examples:

```
get_liking(social_link1)
```

12.4.104 `get_modality`

12.4.104.1 Possible use:

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

12.4.104.2 Result:

get the modality value of the given mental state

12.4.104.3 Examples:

```
get_modality(mental_state1)
```

12.4.105 `get_obligation_op`

12.4.105.1 Possible use:

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

12.4.105.2 Result:

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

12.4.105.3 Examples:

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

12.4.106 `get_obligation_with_name_op`

12.4.106.1 Possible use:

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

12.4.106.2 Result:

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

12.4.106.3 Examples:

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

12.4.107 **get_obligations_op**

12.4.107.1 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

12.4.107.2 Result:

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

12.4.107.3 Examples:

```
get_obligations_op(self,has_water)
```

12.4.108 **get_obligations_with_name_op**

12.4.108.1 Possible use:

- agent **get_obligations_with_name_op** string —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.Men
- **get_obligations_with_name_op** (agent , string) —> msi.gama.util.IList<msi.gaml.architecture.simplebdi.

12.4.108.2 Result:

get the list of obligations in the obligation base which predicate has the given name.

12.4.108.3 Examples:

```
get_obligations_with_name_op(self,"has_water")
```

12.4.109 **get_plan_name**

12.4.109.1 Possible use:

- **get_plan_name** (BDIPlan) —> string

12.4.109.2 Result:

get the name of a given plan

12.4.109.3 Examples:

```
get_plan_name(agent.current_plan)
```

12.4.110 `get_predicate`

12.4.110.1 Possible use:

- **`get_predicate`** (`mental_state`) —> `predicate`

12.4.110.2 Result:

get the predicate value of the given mental state

12.4.110.3 Examples:

```
get_predicate(mental_state1)
```

12.4.111 `get_solidarity`

12.4.111.1 Possible use:

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

12.4.111.2 Result:

get the solidarity value of the given social link

12.4.111.3 Examples:

```
get_solidarity(social_link1)
```

12.4.112 `get_strength`

12.4.112.1 Possible use:

- **`get_strength`** (`mental_state`) —> `float`

12.4.112.2 Result:

get the strength value of the given mental state

12.4.112.3 Examples:

```
get_strength(mental_state1)
```

12.4.113 get_super_intention

12.4.113.1 Possible use:

- **get_super_intention** (predicate) —> mental_state

12.4.114 get_trust

12.4.114.1 Possible use:

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

12.4.114.2 Result:

get the familiarity value of the given social link

12.4.114.3 Examples:

```
get_familiarity(social_link1)
```

12.4.115 get_truth

12.4.115.1 Possible use:

- **get_truth** (predicate) —> bool

12.4.116 get_uncertainties_op

12.4.116.1 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

12.4.116.2 Result:

get the uncertainties in the uncertainty base with the given predicate.

12.4.116.3 Examples:

```
get_uncertinties_op(self,has_water)
```

12.4.117 get_uncertainties_with_name_op

12.4.117.1 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

12.4.117.2 Result:

get the list of uncertainties in the uncertainty base which predicate has the given name.

12.4.117.3 Examples:

```
get_uncertainties_with_name_op(self,"has_water")
```

12.4.118 get_uncertainty_op

12.4.118.1 Possible use:

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

12.4.118.2 Result:

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

12.4.118.3 Examples:

```
get_uncertainty_op(self,has_water)
```

12.4.119 get_uncertainty_with_name_op

12.4.119.1 Possible use:

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

12.4.119.2 Result:

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

12.4.119.3 Examples:

```
get_uncertainty_with_name_op(self,"has_water")
```

12.4.120 gif_file

12.4.120.1 Possible use:

- **gif_file** (string) —> file

12.4.120.2 Result:

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

---

12.4.121 gini

12.4.121.1 Possible use:

- **gini** (list<float>) —> float

12.4.121.2 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
```

12.4.122 gml_file

12.4.122.1 Possible use:

- **gml_file** (string) —> file

12.4.122.2 Result:

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

---

12.4.123 graph

12.4.123.1 Possible use:

- **graph** (any) —> graph

12.4.123.2 Result:

Casts the operand into the type graph

12.4.124 grayscale

12.4.124.1 Possible use:

- **grayscale** (rgb) —> rgb

12.4.124.2 Result:

Converts rgb color to grayscale value

12.4.124.3 Comment:

r=red, g=green, b=blue. Between 0 and 255 and gray = 0.299 * red + 0.587 * green + 0.114 * blue (Photoshop value)

12.4.124.4 Examples:

```
rgb var0 <- grayscale (rgb(255,0,0)); // var0 equals to a dark grey
```

12.4.124.5 See also:

rgb, hsb,

12.4.125 grid_at

12.4.125.1 Possible use:

- species **grid_at** point —> agent
- **grid_at** (species , point) —> agent

12.4.125.2 Result:

returns the cell of the grid (right-hand operand) at the position given by the right-hand operand

12.4.125.3 Comment:

If the left-hand operand is a point of floats, it is used as a point of ints.

12.4.125.4 Special cases:

- if the left-hand operand is not a grid cell species, returns nil

12.4.125.5 Examples:

```
agent var0 <- grid_cell grid_at {1,2}; // var0 equals the agent grid_cell with grid_x=1 and grid_y = 2
```

12.4.126 grid_cells_to_graph

12.4.126.1 Possible use:

- **grid_cells_to_graph** (container) —> graph

12.4.126.2 Result:

creates a graph from a list of cells (operand). An edge is created between neighbors.

12.4.126.3 Examples:

```
my_cell_graph<-grid_cells_to_graph(cells_list)
```

12.4.127 grid_file

12.4.127.1 Possible use:

- **grid_file** (string) —> file

12.4.127.2 Result:

Constructs a file of type grid. Allowed extensions are limited to asc, tif

---

12.4.128 group_by

12.4.128.1 Possible use:

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

12.4.128.2 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

12.4.128.3 Comment:

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

12.4.128.4 Special cases:

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

12.4.128.5 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]]
```

12.4.128.6 See also:

first_with, last_with, where,

12.4.129 harmonic_mean

12.4.129.1 Possible use:

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

12.4.129.2 Result:

the harmonic mean of the elements of the operand. See Harmonic_mean for more details.

12.4.129.3 Comment:

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

12.4.129.4 Examples:

```
float var0 <- harmonic_mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 4.804159445407279
```

12.4.129.5 See also:

mean, median, geometric_mean,

12.4.130 has_belief_op

12.4.130.1 Possible use:

- agent **has_belief_op** predicate —> bool
- **has_belief_op** (agent , predicate) —> bool

12.4.130.2 Result:

indicates if there already is a belief about the given predicate.

12.4.130.3 Examples:

```
has_belief_op(self,has_water)
```

12.4.131 has_belief_with_name_op

12.4.131.1 Possible use:

- agent **has_belief_with_name_op** string —> bool
- **has_belief_with_name_op** (agent , string) —> bool

12.4.131.2 Result:

indicates if there already is a belief about the given name.

12.4.131.3 Examples:

```
has_belief_with_name_op(self,"has_water")
```

12.4.132 has_desire_op

12.4.132.1 Possible use:

- agent **has_desire_op** predicate —> bool
- **has_desire_op** (agent , predicate) —> bool

12.4.132.2 Result:

indicates if there already is a desire about the given predicate.

12.4.132.3 Examples:

```
has_desire_op(self,has_water)
```

12.4.133 has_desire_with_name_op

12.4.133.1 Possible use:

- agent **has_desire_with_name_op** string —> bool
- **has_desire_with_name_op** (agent , string) —> bool

12.4.133.2 Result:

indicates if there already is a desire about the given name.

12.4.133.3 Examples:

```
has_desire_with_name_op(self,"has_water")
```

12.4.134 has_ideal_op

12.4.134.1 Possible use:

- agent **has_ideal_op** predicate —> bool
- **has_ideal_op** (agent , predicate) —> bool

12.4.134.2 Result:

indicates if there already is an ideal about the given predicate.

12.4.134.3 Examples:

```
has_ideal_op(self,has_water)
```

12.4.135 has_ideal_with_name_op

12.4.135.1 Possible use:

- agent **has_ideal_with_name_op** string —> bool
- **has_ideal_with_name_op** (agent , string) —> bool

12.4.135.2 Result:

indicates if there already is an ideal about the given name.

12.4.135.3 Examples:

```
has_ideal_with_name_op(self,"has_water")
```

12.4.136 `has_intention_op`

12.4.136.1 Possible use:

- `agent` **`has_intention_op`** `predicate` —> `bool`
- **`has_intention_op`** (`agent` , `predicate`) —> `bool`

12.4.136.2 Result:

indicates if there already is an intention about the given predicate.

12.4.136.3 Examples:

```
has_intention_op(self,has_water)
```

12.4.137 `has_intention_with_name_op`

12.4.137.1 Possible use:

- `agent` **`has_intention_with_name_op`** `string` —> `bool`
- **`has_intention_with_name_op`** (`agent` , `string`) —> `bool`

12.4.137.2 Result:

indicates if there already is an intention about the given name.

12.4.137.3 Examples:

```
has_intention_with_name_op(self,"has_water")
```

12.4.138 `has_obligation_op`

12.4.138.1 Possible use:

- `agent` **`has_obligation_op`** `predicate` —> `bool`
- **`has_obligation_op`** (`agent` , `predicate`) —> `bool`

12.4.138.2 Result:

indicates if there already is an obligation about the given predicate.

12.4.138.3 Examples:

```
has_obligation_op(self,has_water)
```

12.4.139 has_obligation_with_name_op

12.4.139.1 Possible use:

- agent **has_obligation_with_name_op** string —> bool
- **has_obligation_with_name_op** (agent , string) —> bool

12.4.139.2 Result:

indicates if there already is an obligation about the given name.

12.4.139.3 Examples:

```
has_obligation_with_name_op(self,"has_water")
```

12.4.140 has_uncertainty_op

12.4.140.1 Possible use:

- agent **has_uncertainty_op** predicate —> bool
- **has_uncertainty_op** (agent , predicate) —> bool

12.4.140.2 Result:

indicates if there already is an uncertainty about the given predicate.

12.4.140.3 Examples:

```
has_uncertainty_op(self,has_water)
```

12.4.141 has_uncertainty_with_name_op

12.4.141.1 Possible use:

- agent **has_uncertainty_with_name_op** string —> bool
- **has_uncertainty_with_name_op** (agent , string) —> bool

12.4.141.2 Result:

indicates if there already is an uncertainty about the given name.

12.4.141.3 Examples:

```
has_uncertainty_with_name_op(self,"has_water")
```

12.4.142 `hexagon`

12.4.142.1 Possible use:

- **`hexagon`** (`point`) —> `geometry`
- **`hexagon`** (`float`) —> `geometry`
- `float` **`hexagon`** `float` —> `geometry`
- **`hexagon`** (`float` , `float`) —> `geometry`

12.4.142.2 Result:

A hexagon geometry which the given with and height

12.4.142.3 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.

12.4.142.4 Special cases:

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

12.4.142.5 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.
```

12.4.142.6 See also:

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

---

12.4.143 `hierarchical_clustering`

12.4.143.1 Possible use:

- `container<agent>` **`hierarchical_clustering`** `float` —> `list`
- **`hierarchical_clustering`** (`container<agent>` , `float`) —> `list`

12.4.143.2 Result:

A tree (list of list) contained groups of agents clustered by distance considering a distance min between two groups.

12.4.143.3 Comment:

use of hierarchical clustering with Minimum for linkage criterion between two groups of agents.

12.4.143.4 Examples:

```
list var0 <- [ag1, ag2, ag3, ag4, ag5] hierarchical_clustering 20.0; // var0 equals for example, can return [
```

12.4.143.5 See also:

simple_clustering_by_distance,

12.4.144 `horizontal`

12.4.144.1 Possible use:

- **`horizontal`** (`msi.gama.util.GamaMap<java.lang.Object,java.lang.Integer>`) —> `msi.gama.util.tree.GamaNoc`

12.4.145 `hsb`

12.4.145.1 Possible use:

- **`hsb`** (`float`, `float`, `float`) —> `rgb`
- **`hsb`** (`float`, `float`, `float`, `int`) —> `rgb`
- **`hsb`** (`float`, `float`, `float`, `float`) —> `rgb`

12.4.145.2 Result:

Converts hsb (h=hue, s=saturation, b=brightness) value to Gama color

12.4.145.3 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)

12.4.145.4 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")
```

12.4.145.5 See also:

rgb,

12.4.146 `hypot`

12.4.146.1 Possible use:

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

12.4.146.2 Result:

Returns sqrt(x2 +y2) without intermediate overflow or underflow.

12.4.146.3 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.

12.4.146.4 Examples:

```
float var0 <- hypot(0,1,0,1); // var0 equals sqrt(2)
```

Chapter 13

Operators (I to M)

13.1 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.

13.2 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.

13.3 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).

13.4 Operators

13.4.1 IDW

13.4.1.1 Possible use:

- **IDW** (container<agent>, map<point,float>, int) —> map<agent,float>

13.4.1.2 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)

13.4.1.3 Examples:

```
map<agent,float> var0 <- IDW([ag1, ag2, ag3, ag4, ag5],[{10,10}::25.0, {10,80}::10.0, {100,10}::15.0], 2);
```

13.4.2 image_file

13.4.2.1 Possible use:

- **image_file** (string) —> file

13.4.2.2 Result:

Constructs a file of type image. Allowed extensions are limited to tiff, jpg, jpeg, png, pict, bmp

---

13.4.3 improved_generator

13.4.3.1 Possible use:

- **improved_generator** (float, float, float, float) —> float

13.4.3.2 Result:

take a x, y, z and a bias parameters and gives a value

13.4.3.3 Examples:

```
float var0 <- improved_generator(2,3,4,253); // var0 equals 10.2
```

13.4.4 `in`

13.4.4.1 Possible use:

- `string` **`in`** `string` —> `bool`
- **`in`** (`string` , `string`) —> `bool`
- `unknown` **`in`** `container` —> `bool`
- **`in`** (`unknown` , `container`) —> `bool`

13.4.4.2 Result:

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

13.4.4.3 Comment:

the definition of in depends on the container

13.4.4.4 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

13.4.4.5 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
```

13.4.4.6 See also:

contains,

13.4.5 `in_degree_of`

13.4.5.1 Possible use:

- `graph` **`in_degree_of`** `unknown` —> `int`
- **`in_degree_of`** (`graph` , `unknown`) —> `int`

13.4.5.2 Result:

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

13.4.5.3 Examples:

```
int var1 <- graphFromMap in_degree_of (node(3)); // var1 equals 2
```

13.4.5.4 See also:

out_degree_of, degree_of,

13.4.6 in_edges_of

13.4.6.1 Possible use:

- graph **in_edges_of** unknown —> list
- **in_edges_of** (graph , unknown) —> list

13.4.6.2 Result:

returns the list of the in-edges of a vertex (right-hand operand) in the graph given as left-hand operand.

13.4.6.3 Examples:

```
list var1 <- graphFromMap in_edges_of node({12,45}); // var1 equals [LineString]
```

13.4.6.4 See also:

out_edges_of,

13.4.7 incomplete_beta

13.4.7.1 Possible use:

- **incomplete_beta** (float, float, float) —> float

13.4.7.2 Result:

Returns the regularized integral of the beta function with arguments a and b, from zero to x.

13.4.8 incomplete_gamma

13.4.8.1 Possible use:

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

13.4.8.2 Result:

Returns the regularized integral of the Gamma function with argument a to the integration end point x.

13.4.9 incomplete_gamma_complement

13.4.9.1 Possible use:

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

13.4.9.2 Result:

Returns the complemented regularized incomplete Gamma function of the argument a and integration start point x.

13.4.10 indented_by

13.4.10.1 Possible use:

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

13.4.10.2 Result:

Converts a (possibly multiline) string by indenting it by a number – specified by the second operand – of tabulations to the right

---

13.4.11 index_by

13.4.11.1 Possible use:

- `container` **`index_by`** `any expression` —> `map`
- **`index_by`** (`container` , `any expression`) —> `map`

13.4.11.2 Result:

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

13.4.11.3 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

13.4.11.4 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,
```

13.4.12 index_of

13.4.12.1 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`

13.4.12.2 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

13.4.12.3 Comment:

The definition of index_of and the type of the index depend on the container

13.4.12.4 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
```

13.4.12.5 Examples:

```
unknown var0 <- [1::2, 3::4, 5::6] index_of 4; // var0 equals 3
```

13.4.12.6 See also:

at, last_index_of,

13.4.13 inside

13.4.13.1 Possible use:

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

13.4.13.2 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).

13.4.13.3 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
```

13.4.13.4 See also:

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

---

13.4.14 `int`

13.4.14.1 Possible use:

- **`int`** (`any`) —> `int`

13.4.14.2 Result:

Casts the operand into the type int

13.4.15 `inter`

13.4.15.1 Possible use:

- `container` **`inter`** `container` —> `list`
- **`inter`** (`container` , `container`) —> `list`
- `geometry` **`inter`** `geometry` —> `geometry`
- **`inter`** (`geometry` , `geometry`) —> `geometry`

13.4.15.2 Result:

the intersection of the two operands A geometry resulting from the intersection between the two geometries

13.4.15.3 Comment:

both containers are transformed into sets (so without duplicated element, cf. remove_deplicates operator) before the set intersection is computed.

13.4.15.4 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]
```

13.4.15.5 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)
```

13.4.15.6 See also:

remove_duplicates, union, +, -,

13.4.16 `interleave`

13.4.16.1 Possible use:

- **`interleave`** (`container`) —> `list`

13.4.16.2 Result:

a new list containing the interleaved elements of the containers contained in the operand

13.4.16.3 Comment:

the operand should be a list of lists of elements. The result is a list of elements.

13.4.16.4 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
```

13.4.17 `internal_at`

13.4.17.1 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`

13.4.17.2 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]

13.4.17.3 See also:

at,

13.4.18 `internal_integrated_value`

13.4.18.1 Possible use:

- `any expression` **`internal_integrated_value`** `any expression` —> `list`
- **`internal_integrated_value`** (`any expression` , `any expression`) —> `list`

13.4.18.2 Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with [index]

13.4.19 `internal_zero_order_equation`

13.4.19.1 Possible use:

- **`internal_zero_order_equation`** (`any expression`) —> `float`

13.4.19.2 Result:

An internal placeholder function

13.4.20 `intersection`

Same signification as inter

13.4.21 `intersects`

13.4.21.1 Possible use:

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

13.4.21.2 Result:

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

13.4.21.3 Special cases:

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

13.4.21.4 Examples:

```
bool var0 <- square(5) intersects {10,10}; // var0 equals false
```

13.4.21.5 See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

13.4.22 `inverse`

13.4.22.1 Possible use:

- **`inverse`** (`matrix`) —> `matrix<float>`

13.4.22.2 Result:

The inverse matrix of the given matrix. If no inverse exists, returns a matrix that has properties that resemble that of an inverse.

13.4.22.3 Examples:

```
matrix<float> var0 <- inverse(matrix([[4,3],[3,2]])); // var0 equals matrix([[-2.0,3.0],[3.0,-4.0]])
```

13.4.23 `inverse_distance_weighting`

Same signification as IDW

13.4.24 `is`

13.4.24.1 Possible use:

- `unknown` **`is`** `any expression` —> `bool`
- **`is`** (`unknown` , `any expression`) —> `bool`

13.4.24.2 Result:

returns true if the left operand is of the right operand type, false otherwise

13.4.24.3 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
```

13.4.25 is_csv

13.4.25.1 Possible use:

- **is_csv** (any) —> bool

13.4.25.2 Result:

Tests whether the operand is a csv file.

13.4.26 is_dxf

13.4.26.1 Possible use:

- **is_dxf** (any) —> bool

13.4.26.2 Result:

Tests whether the operand is a dxf file.

13.4.27 is_error

13.4.27.1 Possible use:

- **is_error** (any expression) —> bool

13.4.27.2 Result:

Returns whether or not the argument raises an error when evaluated

---

13.4.28 is_finite

13.4.28.1 Possible use:

- **is_finite** (float) —> bool

13.4.28.2 Result:

Returns whether the argument is a finite number or not

13.4.28.3 Examples:

```
bool var0 <- is_finite(4.66); // var0 equals true
bool var1 <- is_finite(#infinity); // var1 equals false
```

13.4.29 is_gaml

13.4.29.1 Possible use:

- **is_gaml** (any) —> bool

13.4.29.2 Result:

Tests whether the operand is a gaml file.

13.4.30 is_geojson

13.4.30.1 Possible use:

- **is_geojson** (any) —> bool

13.4.30.2 Result:

Tests whether the operand is a geojson file.

13.4.31 is_gif

13.4.31.1 Possible use:

- **is_gif** (any) —> bool

13.4.31.2 Result:

Tests whether the operand is a gif file.

13.4.32 is_gml

13.4.32.1 Possible use:

- **is_gml** (any) —> bool

13.4.32.2 Result:

Tests whether the operand is a gml file.

13.4.33 is_grid

13.4.33.1 Possible use:

- **is_grid** (any) —> bool

13.4.33.2 Result:

Tests whether the operand is a grid file.

13.4.34 is_image

13.4.34.1 Possible use:

- **is_image** (any) —> bool

13.4.34.2 Result:

Tests whether the operand is a image file.

13.4.35 is_json

13.4.35.1 Possible use:

- **is_json** (any) —> bool

13.4.35.2 Result:

Tests whether the operand is a json file.

13.4.36 is_number

13.4.36.1 Possible use:

- **is_number** (string) —> bool
- **is_number** (float) —> bool

13.4.36.2 Result:

tests whether the operand represents a numerical value Returns whether the argument is a real number or not

13.4.36.3 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 #.

13.4.36.4 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
```

13.4.37 is_obj

13.4.37.1 Possible use:

- **is_obj** (any) —> bool

13.4.37.2 Result:

Tests whether the operand is a obj file.

13.4.38 is_osm

13.4.38.1 Possible use:

- `is_osm` (`any`) —> `bool`

13.4.38.2 Result:

Tests whether the operand is a osm file.

13.4.39 is_pgm

13.4.39.1 Possible use:

- `is_pgm` (`any`) —> `bool`

13.4.39.2 Result:

Tests whether the operand is a pgm file.

13.4.40 is_property

13.4.40.1 Possible use:

- `is_property` (`any`) —> `bool`

13.4.40.2 Result:

Tests whether the operand is a property file.

13.4.41 is_R

13.4.41.1 Possible use:

- `is_R` (`any`) —> `bool`

13.4.41.2 Result:

Tests whether the operand is a R file.

13.4.42 is_shape

13.4.42.1 Possible use:

- `is_shape` (`any`) —> `bool`

13.4.42.2 Result:

Tests whether the operand is a shape file.

13.4.43 is_skill

13.4.43.1 Possible use:

- `unknown` `is_skill` `string` —> `bool`
- `is_skill` (`unknown` , `string`) —> `bool`

13.4.43.2 Result:

returns true if the left operand is an agent whose species implements the right-hand skill name

13.4.43.3 Examples:

```
bool var0 <- agentA is_skill 'moving'; // var0 equals true
```

13.4.44 is_svg

13.4.44.1 Possible use:

- `is_svg` (`any`) —> `bool`

13.4.44.2 Result:

Tests whether the operand is a svg file.

13.4.45 is_text

13.4.45.1 Possible use:

- `is_text` (`any`) —> `bool`

13.4.45.2 Result:

Tests whether the operand is a text file.

13.4.46 `is_threeds`

13.4.46.1 Possible use:

- **`is_threeds`** (`any`) —> `bool`

13.4.46.2 Result:

Tests whether the operand is a threeds file.

13.4.47 `is_URL`

13.4.47.1 Possible use:

- **`is_URL`** (`any`) —> `bool`

13.4.47.2 Result:

Tests whether the operand is a URL file.

13.4.48 `is_warning`

13.4.48.1 Possible use:

- **`is_warning`** (`any expression`) —> `bool`

13.4.48.2 Result:

Returns whether or not the argument raises a warning when evaluated

---

13.4.49 `is_xml`

13.4.49.1 Possible use:

- **`is_xml`** (`any`) —> `bool`

13.4.49.2 Result:

Tests whether the operand is a xml file.

13.4.50 json_file

13.4.50.1 Possible use:

- **json_file** (string) —> file

13.4.50.2 Result:

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

---

13.4.51 kappa

13.4.51.1 Possible use:

- **kappa** (list, list, list) —> float
- **kappa** (list, list, list, list) —> float

13.4.51.2 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.

13.4.51.3 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],[
```

13.4.52 kappa_sim

13.4.52.1 Possible use:

- **kappa_sim** (list, list, list, list) —> float
- **kappa_sim** (list, list, list, list, list) —> float

13.4.52.2 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).

13.4.52.3 Examples:

```
kappa([cat1,cat1,cat2,cat2,cat2],[cat2,cat1,cat2,cat1,cat3],[cat2,cat1,cat2,cat3,cat3], [cat1,cat2,cat3]
```

13.4.53 kmeans

13.4.53.1 Possible use:

- list **kmeans** int —> list<list>
- **kmeans** (list , int) —> list<list>
- **kmeans** (list, int, int) —> list<list>

13.4.53.2 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)

13.4.53.3 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

13.4.53.4 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 []
```

13.4.54 kurtosis

13.4.54.1 Possible use:

- **kurtosis** (list) —> float

13.4.54.2 Result:

returns kurtosis value computed from the operand list of values

13.4.54.3 Special cases:

- if the length of the list is lower than 3, returns NaN

13.4.54.4 Examples:

```
float var0 <- kurtosis ([1,2,3,4,5]); // var0 equals 1.0
```

13.4.55 kurtosis

13.4.55.1 Possible use:

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

13.4.55.2 Result:

Returns the kurtosis (aka excess) of a data sequence Returns the kurtosis (aka excess) of a data sequence

---

13.4.56 last

13.4.56.1 Possible use:

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

13.4.56.2 Result:

the last element of the operand

13.4.56.3 Comment:

the last operator behavior depends on the nature of the operand

13.4.56.4 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
```

13.4.56.5 See also:

first,

13.4.57 `last_index_of`

13.4.57.1 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`

13.4.57.2 Result:

the index of the last occurence of the right operand in the left operand container

13.4.57.3 Comment:

The definition of last_index_of and the type of the index depend on the container

13.4.57.4 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
```

13.4.57.5 See also:

at, index_of, last_index_of,

13.4.58 `last_of`

Same signification as last

13.4.59 `last_with`

13.4.59.1 Possible use:

- `container` **`last_with`** `any expression` —> `unknown`
- **`last_with`** (`container` , `any expression`) —> `unknown`

13.4.59.2 Result:

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

13.4.59.3 Comment:

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

13.4.59.4 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)
```

13.4.59.5 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
```

13.4.59.6 See also:

group_by, first_with, where,

13.4.60 `layout`

13.4.60.1 Possible use:

- `graph` **`layout`** `string` —> `graph`
- **`layout`** (`graph` , `string`) —> `graph`
- **`layout`** (`graph`, `string`, `int`) —> `graph`
- **`layout`** (`graph`, `string`, `int`, `map<string,unknown>`) —> `graph`

13.4.60.2 Result:

layouts a GAMA graph.

13.4.61 `length`

13.4.61.1 Possible use:

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

13.4.61.2 Result:

the number of elements contained in the operand

13.4.61.3 Comment:

the length operator behavior depends on the nature of the operand

13.4.61.4 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
```

13.4.62 `lgamma`

Same signification as log_gamma

13.4.63 `line`

13.4.63.1 Possible use:

- **`line`** (`container<geometry>`) —> `geometry`
- `container<geometry>` **`line`** `float` —> `geometry`
- **`line`** (`container<geometry>` , `float`) —> `geometry`

13.4.63.2 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.

13.4.63.3 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
```

13.4.63.4 Examples:

```
geometry var1 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}]); // var1 equals a polyline geometry composed of t
```

13.4.63.5 See also:

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

---

13.4.64 `link`

13.4.64.1 Possible use:

- `geometry` **`link`** `geometry` —> `geometry`
- **`link`** (`geometry` , `geometry`) —> `geometry`

13.4.64.2 Result:

A dynamic line geometry between the location of the two operands

13.4.64.3 Comment:

The geometry of the link is a line between the locations of the two operands, which is built and maintained dynamically

13.4.64.4 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}

13.4.64.5 Examples:

```
geometry var0 <- link (geom1,geom2); // var0 equals a link geometry between geom1 and geom2.
```

13.4.64.6 See also:

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

---

13.4.65 `list`

13.4.65.1 Possible use:

- **`list`** (`any`) —> `list`

13.4.65.2 Result:

Casts the operand into the type list

13.4.66 `list_with`

13.4.66.1 Possible use:

- `int` **`list_with`** `any expression` —> `list`
- **`list_with`** (`int` , `any expression`) —> `list`

13.4.66.2 Result:

creates a list with a size provided by the first operand, and filled with the second operand

13.4.66.3 Comment:

Note that the right operand should be positive, and that the second one is evaluated for each position in the list.

13.4.66.4 See also:

list,

13.4.67 `ln`

13.4.67.1 Possible use:

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

13.4.67.2 Result:

Returns the natural logarithm (base e) of the operand.

13.4.67.3 Special cases:

- an exception is raised if the operand is less than zero.

13.4.67.4 Examples:

```
float var0 <- ln(exp(1)); // var0 equals 1.0
float var1 <- ln(1); // var1 equals 0.0
```

13.4.67.5 See also:

exp,

13.4.68 `load_graph_from_file`

13.4.68.1 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`

13.4.68.2 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

13.4.68.3 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:

13.4.68.4 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
```

13.4.68.5 Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(       "pajek",        "./example_of_Pa
```

13.4.69 `load_shortest_paths`

13.4.69.1 Possible use:

- `graph` **`load_shortest_paths`** `matrix` —> `graph`
- **`load_shortest_paths`** (`graph` , `matrix`) —> `graph`

13.4.69.2 Result:

put in the graph cache the computed shortest paths contained in the matrix (rows: source, columns: target)

13.4.69.3 Examples:

```
graph var0 <- load_shortest_paths(shortest_paths_matrix); // var0 equals return my_graph with all the short
```

13.4.70 `load_sub_model`

13.4.70.1 Possible use:

- `string` **`load_sub_model`** `string` —> `msi.gama.kernel.experiment.IExperimentAgent`
- **`load_sub_model`** (`string` , `string`) —> `msi.gama.kernel.experiment.IExperimentAgent`

13.4.70.2 Result:

Load a submodel

13.4.70.3 Comment:

loaded submodel

13.4.71 `log`

13.4.71.1 Possible use:

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

13.4.71.2 Result:

Returns the logarithm (base 10) of the operand.

13.4.71.3 Special cases:

- an exception is raised if the operand is equals or less than zero.

13.4.71.4 Examples:

```
float var0 <- log(10); // var0 equals 1.0
float var1 <- log(1); // var1 equals 0.0
```

13.4.71.5 See also:

ln,

13.4.72 `log_gamma`

13.4.72.1 Possible use:

- `log_gamma` (`float`) —> `float`

13.4.72.2 Result:

Returns the log of the value of the Gamma function at x.

13.4.73 lower_case

13.4.73.1 Possible use:

- `lower_case` (`string`) —> `string`

13.4.73.2 Result:

Converts all of the characters in the string operand to lower case

13.4.73.3 Examples:

```
string var0 <- lower_case("Abc"); // var0 equals 'abc'
```

13.4.73.4 See also:

upper_case,

13.4.74 main_connected_component

13.4.74.1 Possible use:

- `main_connected_component` (`graph`) —> `graph`

13.4.74.2 Result:

returns the sub-graph corresponding to the main connected components of the graph

13.4.74.3 Examples:

```
graph var0 <- main_connected_component(my_graph); // var0 equals the sub-graph corresponding to the main con
```

13.4.74.4 See also:

connected_components_of,

13.4.75 map

13.4.75.1 Possible use:

- `map` (`any`) —> `map`

13.4.75.2 Result:

Casts the operand into the type map

13.4.76 `masked_by`

13.4.76.1 Possible use:

- `geometry` **`masked_by`** `container<geometry>` —> `geometry`
- **`masked_by`** (`geometry` , `container<geometry>`) —> `geometry`
- **`masked_by`** (`geometry`, `container<geometry>`, `int`) —> `geometry`

13.4.76.2 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
```

13.4.77 `material`

13.4.77.1 Possible use:

- `float` **`material`** `float` —> `msi.gama.util.GamaMaterial`
- **`material`** (`float` , `float`) —> `msi.gama.util.GamaMaterial`

13.4.77.2 Result:

Returns

13.4.77.3 Examples:

13.4.77.4 See also:

,

13.4.78 `material`

13.4.78.1 Possible use:

- **`material`** (any) —> `material`

13.4.78.2 Result:

Casts the operand into the type material

13.4.79 matrix

13.4.79.1 Possible use:

- **matrix** (any) —> matrix

13.4.79.2 Result:

Casts the operand into the type matrix

13.4.80 matrix_with

13.4.80.1 Possible use:

- point **matrix_with** any expression —> matrix
- **matrix_with** (point , any expression) —> matrix

13.4.80.2 Result:

creates a matrix with a size provided by the first operand, and filled with the second operand

13.4.80.3 Comment:

Note that both components of the right operand point should be positive, otherwise an exception is raised.

13.4.80.4 See also:

matrix, as_matrix,

13.4.81 max

13.4.81.1 Possible use:

- **max** (container) —> unknown

13.4.81.2 Result:

the maximum element found in the operand

13.4.81.3 Comment:

the max operator behavior depends on the nature of the operand

13.4.81.4 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}
```

13.4.81.5 See also:

min,

13.4.82 `max_flow_between`

13.4.82.1 Possible use:

- **`max_flow_between`** (`graph`, `unknown`, `unknown`) —> `msi.gama.util.GamaMap<java.lang.Object,java.lang.Double>`

13.4.82.2 Result:

The max flow (map in a graph between the source and the sink using Edmonds-Karp algorithm

13.4.82.3 Examples:

```
max_flow_between(my_graph, vertice1, vertice2)
```

13.4.83 max_of

13.4.83.1 Possible use:

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

13.4.83.2 Result:

the maximum value of the right-hand expression evaluated on each of the elements of the left-hand operand

13.4.83.3 Comment:

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

13.4.83.4 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
```

13.4.83.5 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
```

13.4.83.6 See also:

min_of,

13.4.84 maximal_cliques_of

13.4.84.1 Possible use:

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

13.4.84.2 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.

13.4.84.3 Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- maximal_cliques_of (my_graph); // var1 equals the list of all the maximal cliques as list
```

13.4.84.4 See also:

biggest_cliques_of,

13.4.85 `mean`

13.4.85.1 Possible use:

- **`mean`** (`container`) —> `unknown`

13.4.85.2 Result:

the mean of all the elements of the operand

13.4.85.3 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.

13.4.85.4 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

13.4.85.5 Examples:

```
unknown var0 <- mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 5.125
```

13.4.85.6 See also:

sum,

13.4.86 mean_deviation

13.4.86.1 Possible use:

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

13.4.86.2 Result:

the deviation from the mean of all the elements of the operand. See Mean_deviation for more details.

13.4.86.3 Comment:

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

13.4.86.4 Examples:

```
float var0 <- mean_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.125
```

13.4.86.5 See also:

mean, standard_deviation,

13.4.87 mean_of

13.4.87.1 Possible use:

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

13.4.87.2 Result:

the mean of the right-hand expression evaluated on each of the elements of the left-hand operand

13.4.87.3 Comment:

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

13.4.87.4 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
```

13.4.87.5 Examples:

```
unknown var0 <- [1,2] mean_of (each * 10 ); // var0 equals 15
```

13.4.87.6 See also:

min_of, max_of, sum_of, product_of,

13.4.88 meanR

13.4.88.1 Possible use:

- **meanR** (container) —> unknown

13.4.88.2 Result:

returns the mean value of given vector (right-hand operand) in given variable (left-hand operand).

13.4.88.3 Examples:

```
list<int> X <- [2, 3, 1];
int var1 <- meanR(X); // var1 equals 2
```

13.4.89 median

13.4.89.1 Possible use:

- **median** (container) —> unknown

13.4.89.2 Result:

the median of all the elements of the operand.

13.4.89.3 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

13.4.89.4 Examples:

```
unknown var0 <- median ([4.5, 3.5, 5.5, 3.4, 7.0]); // var0 equals 4.5
```

13.4.89.5 See also:

mean,

13.4.90 `mental_state`

13.4.90.1 Possible use:

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

13.4.90.2 Result:

Casts the operand into the type mental_state

13.4.91 `message`

13.4.91.1 Possible use:

- **`message`** (`unknown`) —> `msi.gama.extensions.messaging.GamaMessage`

13.4.91.2 Result:

to be added

13.4.92 `milliseconds_between`

13.4.92.1 Possible use:

- `date` **`milliseconds_between`** `date` —> `float`
- **`milliseconds_between`** (`date` , `date`) —> `float`

13.4.92.2 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)

13.4.92.3 Examples:

```
float var0 <- milliseconds_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 2.6784E9
```

13.4.93 `min`

13.4.93.1 Possible use:

- **`min`** (`container`) —> `unknown`

13.4.93.2 Result:

the minimum element found in the operand.

13.4.93.3 Comment:

the min operator behavior depends on the nature of the operand

13.4.93.4 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
```

13.4.93.5 See also:

max,

13.4.94 min_of

13.4.94.1 Possible use:

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

13.4.94.2 Result:

the minimum value of the right-hand expression evaluated on each of the elements of the left-hand operand

13.4.94.3 Comment:

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

13.4.94.4 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
```

13.4.94.5 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
```

13.4.94.6 See also:

max_of,

13.4.95 minus_days

13.4.95.1 Possible use:

- date **minus_days** int —> date
- **minus_days** (date , int) —> date

13.4.95.2 Result:

Subtract a given number of days from a date

13.4.95.3 Examples:

```
date var0 <- date('2000-01-01') minus_days 20; // var0 equals date('1999-12-12')
```

13.4.96 minus_hours

13.4.96.1 Possible use:

- date **minus_hours** int —> date
- **minus_hours** (date , int) —> date

13.4.96.2 Result:

Remove a given number of hours from a date

13.4.96.3 Examples:

```
// equivalent to date1 - 15 #h
date var1 <- date('2000-01-01') minus_hours 15 ; // var1 equals date('1999-12-31 09:00:00')
```

13.4.97 minus_minutes

13.4.97.1 Possible use:

- date **minus_minutes** int —> date
- **minus_minutes** (date , int) —> date

13.4.97.2 Result:

Subtract a given number of minutes from a date

13.4.97.3 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')
```

13.4.98 minus_months

13.4.98.1 Possible use:

- date **minus_months** int —> date
- **minus_months** (date , int) —> date

13.4.98.2 Result:

Subtract a given number of months from a date

13.4.98.3 Examples:

```
date var0 <- date('2000-01-01') minus_months 5; // var0 equals date('1999-08-01')
```

13.4.99 minus_ms

13.4.99.1 Possible use:

- date **minus_ms** int —> date
- **minus_ms** (date , int) —> date

13.4.99.2 Result:

Remove a given number of milliseconds from a date

13.4.99.3 Examples:

```
// equivalent to date1 - 15 #ms
date var1 <- date('2000-01-01') minus_ms 1000 ; // var1 equals date('1999-12-31 23:59:59')
```

13.4.100 minus_seconds

Same signification as -

13.4.101 minus_weeks

13.4.101.1 Possible use:

- date **minus_weeks** int —> date
- **minus_weeks** (date , int) —> date

13.4.101.2 Result:

Subtract a given number of weeks from a date

13.4.101.3 Examples:

```
date var0 <- date('2000-01-01') minus_weeks 15; // var0 equals date('1999-09-18')
```

13.4.102 minus_years

13.4.102.1 Possible use:

- date **minus_years** int —> date
- **minus_years** (date , int) —> date

13.4.102.2 Result:

Subtract a given number of year from a date

13.4.102.3 Examples:

```
date var0 <- date('2000-01-01') minus_years 3; // var0 equals date('1997-01-01')
```

13.4.103 mod

13.4.103.1 Possible use:

- int **mod** int —> int
- **mod** (int , int) —> int

13.4.103.2 Result:

Returns the remainder of the integer division of the left-hand operand by the right-hand operand.

13.4.103.3 Special cases:

- if operands are float, they are truncated
- if the right-hand operand is equal to zero, raises an exception.

13.4.103.4 Examples:

```
int var0 <- 40 mod 3; // var0 equals 1
```

13.4.103.5 See also:

div,

13.4.104 moment

13.4.104.1 Possible use:

- **moment** (container, int, float) —> float

13.4.104.2 Result:

Returns the moment of k-th order with constant c of a data sequence

---

13.4.105 months_between

13.4.105.1 Possible use:

- date **months_between** date —> int
- **months_between** (date , date) —> int

13.4.105.2 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)

13.4.105.3 Examples:

```
int var0 <- months_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 1
```

13.4.106 moran

13.4.106.1 Possible use:

- list<float> **moran** matrix<float> —> float
- **moran** (list<float> , matrix<float>) —> float

13.4.106.2 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
```

13.4.107 mul

13.4.107.1 Possible use:

- `mul` (`container`) —> `unknown`

13.4.107.2 Result:

the product of all the elements of the operand

13.4.107.3 Comment:

the mul operator behavior depends on the nature of the operand

13.4.107.4 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
```

13.4.107.5 See also:

sum,

Chapter 14

Operators (N to R)

14.1 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.

14.2 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.

14.3 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).

14.4 Operators

14.4.1 nb_cycles

14.4.1.1 Possible use:

- **nb_cycles** (graph) —> int

14.4.1.2 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.

14.4.1.3 Examples:

```
graph graphEpidemio <- graph([]);
int var1 <- nb_cycles(graphEpidemio); // var1 equals the number of cycles in the graph
```

14.4.1.4 See also:

alpha_index, beta_index, gamma_index, connectivity_index,

14.4.2 neighbors_at

14.4.2.1 Possible use:

- geometry **neighbors_at** float —> list
- **neighbors_at** (geometry , float) —> list

14.4.2.2 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).

14.4.2.3 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.

14.4.2.4 Examples:

```
list var0 <- (self neighbors_at (10)); // var0 equals all the agents located at a distance lower or equal to 1
```

14.4.2.5 See also:

neighbors_of, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to, at_distance,

---

14.4.3 `neighbors_of`

14.4.3.1 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`

14.4.3.2 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.

14.4.3.3 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
```

14.4.3.4 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
```

14.4.3.5 See also:

predecessors_of, successors_of, neighbors_at, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

14.4.4 `new_emotion`

14.4.4.1 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

14.4.4.2 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)

14.4.4.3 Examples:

```
emotion("joy",12.3,eatFood,4) emotion("joy",12.3,4) emotion("joy",eatFood) emotion("joy",12.3,eatFood,4)
```

14.4.5 `new_folder`

14.4.5.1 Possible use:

- **new_folder** (string) —> file

14.4.5.2 Result:

opens an existing repository or create a new folder if it does not exist.

14.4.5.3 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.

14.4.5.4 Examples:

```
file dirNewT <- new_folder("incl/");   // dirNewT represents the repository "../incl/"
```

14.4.5.5 See also:

folder, file,

14.4.6 new_mental_state

14.4.6.1 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

14.4.6.2 Result:

a new mental state a new mental state

14.4.6.3 Examples:

```
new_mental-state(belief) new_mental-state(belief) new_mental-state(belief) new_mental-state(belief) new_m
```

14.4.7 new_predicate

14.4.7.1 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

14.4.7.2 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)

14.4.7.3 Examples:

```
predicate("people to meet") predicate("hasWater", 10  predicate("people to meet", ["time"::10], true) predi
```

14.4.8 new_social_link

14.4.8.1 Possible use:

- **new_social_link** (agent) —> msi.gaml.architecture.simplebdi.SocialLink
- **new_social_link** (agent, float, float, float, float) —> msi.gaml.architecture.simplebdi.SocialLink

14.4.8.2 Result:

a new social link a new social link

14.4.8.3 Examples:

```
new_social_link(agentA) new_social_link(agentA,0.0,-0.1,0.2,0.1)
```

14.4.9 `node`

14.4.9.1 Possible use:

- **`node`** (`unknown`) —> `unknown`
- `unknown` **`node`** `float` —> `unknown`
- **`node`** (`unknown` , `float`) —> `unknown`

14.4.10 `nodes`

14.4.10.1 Possible use:

- **`nodes`** (`container`) —> `container`

14.4.11 `norm`

14.4.11.1 Possible use:

- **`norm`** (`point`) —> `float`

14.4.11.2 Result:

the norm of the vector with the coordinates of the point operand.

14.4.11.3 Examples:

```
float var0 <- norm({3,4}); // var0 equals 5.0
```

14.4.12 `Norm`

14.4.12.1 Possible use:

- **`Norm`** (`any`) —> `Norm`

14.4.12.2 Result:

Casts the operand into the type Norm

14.4.13 `normal_area`

14.4.13.1 Possible use:

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

14.4.13.2 Result:

Returns the area to the left of x in the normal distribution with the given mean and standard deviation.

14.4.14 `normal_density`

14.4.14.1 Possible use:

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

14.4.14.2 Result:

Returns the probability of x in the normal distribution with the given mean and standard deviation.

14.4.15 `normal_inverse`

14.4.15.1 Possible use:

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

14.4.15.2 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.

14.4.16 `not`

Same signification as !

14.4.17 `obj_file`

14.4.17.1 Possible use:

- **`obj_file`** (`string`) —> `file`

14.4.17.2 Result:

Constructs a file of type obj. Allowed extensions are limited to obj, OBJ

14.4.18 of

Same signification as .

14.4.19 of_generic_species

14.4.19.1 Possible use:

- container **of_generic_species** species —> list
- **of_generic_species** (container , species) —> list

14.4.19.2 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

14.4.19.3 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 [sous
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,s
```

14.4.19.4 See also:

of_species,

14.4.20 of_species

14.4.20.1 Possible use:

- container **of_species** species —> list
- **of_species** (container , species) —> list

14.4.20.2 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).

14.4.20.3 Special cases:

- if the right operand is nil, of_species returns the right operand

14.4.20.4 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]
```

14.4.20.5 See also:

of_generic_species,

14.4.21 `one_of`

14.4.21.1 Possible use:

- **`one_of`** (`container<KeyType,ValueType>`) —> `ValueType`

14.4.21.2 Result:

one of the values stored in this container at a random key

14.4.21.3 Comment:

the one_of operator behavior depends on the nature of the operand

14.4.21.4 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
```

14.4.21.5 See also:

contains,

14.4.22 `open_simplex_generator`

14.4.22.1 Possible use:

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

14.4.22.2 Result:

take a x, y and a bias parameters and gives a value

14.4.22.3 Examples:

```
float var0 <- open_simplex_generator(2,3,253); // var0 equals 10.2
```

14.4.23 `or`

14.4.23.1 Possible use:

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

14.4.23.2 Result:

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

14.4.23.3 Comment:

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

14.4.23.4 See also:

bool, and, !,

14.4.24 `or`

14.4.24.1 Possible use:

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

14.4.24.2 Result:

create a new predicate from two others by including them as subintentions. It's an exclusive "or"

14.4.24.3 Examples:

```
predicate1 or predicate2
```

14.4.25 `osm_file`

14.4.25.1 Possible use:

- `string` **`osm_file`** `map<string,list>` —> `file`
- **`osm_file`** (`string` , `map<string,list>`) —> `file`
- **`osm_file`** (`string`, `map<string,list>`, `int`) —> `file`

14.4.25.2 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.

14.4.25.3 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.

14.4.25.4 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.

14.4.25.5 Examples:

```
file myOSMfile2 <- osm_file("../includes/rouen.osm",["highway"::["primary","motorway"]], 0); file myOSMfi
```

14.4.25.6 See also:

file,

14.4.26 out_degree_of

14.4.26.1 Possible use:

- `graph` **`out_degree_of`** `unknown` —> `int`
- **`out_degree_of`** (`graph` , `unknown`) —> `int`

14.4.26.2 Result:

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

14.4.26.3 Examples:

```
int var1 <- graphFromMap out_degree_of (node(3)); // var1 equals 4
```

14.4.26.4 See also:

in_degree_of, degree_of,

14.4.27 out_edges_of

14.4.27.1 Possible use:

- `graph` **`out_edges_of`** `unknown` —> `list`
- **`out_edges_of`** (`graph` , `unknown`) —> `list`

14.4.27.2 Result:

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

14.4.27.3 Examples:

```
list var1 <- graphFromMap out_edges_of (node(3)); // var1 equals 3
```

14.4.27.4 See also:

in_edges_of,

14.4.28 `overlapping`

14.4.28.1 Possible use:

- `container<agent>` **`overlapping`** `geometry` —> `list<geometry>`
- **`overlapping`** (`container<agent>` , `geometry`) —> `list<geometry>`

14.4.28.2 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).

14.4.28.3 Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] overlapping(self); // var0 equals return the agents among ag1, ag2 and
```

14.4.28.4 See also:

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

---

14.4.29 `overlaps`

14.4.29.1 Possible use:

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

14.4.29.2 Result:

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

14.4.29.3 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

14.4.29.4 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}
```

14.4.29.5 See also:

disjoint_from, crosses, intersects, partially_overlaps, touches,

---

14.4.30 pair

14.4.30.1 Possible use:

- **pair** (any) —> pair

14.4.30.2 Result:

Casts the operand into the type pair

14.4.31 partially_overlaps

14.4.31.1 Possible use:

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

14.4.31.2 Result:

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

14.4.31.3 Comment:

if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).

14.4.31.4 Special cases:

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

14.4.31.5 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
```

14.4.31.6 See also:

disjoint_from, crosses, overlaps, intersects, touches,

14.4.32 path

14.4.32.1 Possible use:

- **path** (any) —> path

14.4.32.2 Result:

Casts the operand into the type path

14.4.33 path_between

14.4.33.1 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

14.4.33.2 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

14.4.33.3 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
```

14.4.33.4 See also:

towards, direction_to, distance_between, direction_between, path_to, distance_to,

---

14.4.34 path_to

14.4.34.1 Possible use:

- geometry **path_to** geometry —> path
- **path_to** (geometry , geometry) —> path
- point **path_to** point —> path
- **path_to** (point , point) —> path

14.4.34.2 Result:

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

14.4.34.3 Examples:

```
path var0 <- ag1 path_to ag2; // var0 equals the path between ag1 and ag2 considering the topology of the agen
```

14.4.34.4 See also:

towards, direction_to, distance_between, direction_between, path_between, distance_to,

---

14.4.35 paths_between

14.4.35.1 Possible use:

- **paths_between** (graph, pair, int) —> msi.gama.util.IList<msi.gama.util.path.GamaSpatialPath>

14.4.35.2 Result:

The K shortest paths between a list of two objects in a graph

14.4.35.3 Examples:

```
msi.gama.util.IList<msi.gama.util.path.GamaSpatialPath> var0 <- paths_between(my_graph, ag1:: ag2, 2); //
```

14.4.36 pbinom

Same signification as binomial_sum

14.4.37 pchisq

Same signification as chi_square

14.4.38 percent_absolute_deviation

14.4.38.1 Possible use:

- list<float> **percent_absolute_deviation** list<float> —> float
- **percent_absolute_deviation** (list<float> , list<float>) —> float

14.4.38.2 Result:

percent absolute deviation indicator for 2 series of values: percent_absolute_deviation(list_vals_observe,list_vals_sim)

14.4.38.3 Examples:

```
percent_absolute_deviation([200,300,150,150,200],[250,250,100,200,200])
```

14.4.39 percentile

Same signification as quantile_inverse

14.4.40 pgamma

Same signification as gamma_distribution

14.4.41 pgm_file

14.4.41.1 Possible use:

- **pgm_file** (string) —> file

14.4.41.2 Result:

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

---

14.4.42 plan

14.4.42.1 Possible use:

- container<geometry> **plan** float —> geometry
- **plan** (container<geometry> , float) —> geometry

14.4.42.2 Result:

A polyline geometry from the given list of points.

14.4.42.3 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.

14.4.42.4 Examples:

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

14.4.42.5 See also:

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

---

14.4.43 plus_days

14.4.43.1 Possible use:

- date **plus_days** int —> date
- **plus_days** (date , int) —> date

14.4.43.2 Result:

Add a given number of days to a date

14.4.43.3 Examples:

```
date var0 <- date('2000-01-01') plus_days 12; // var0 equals date('2000-01-13')
```

14.4.44 plus_hours

14.4.44.1 Possible use:

- date **plus_hours** int —> date
- **plus_hours** (date , int) —> date

14.4.44.2 Result:

Add a given number of hours to a date

14.4.44.3 Examples:

```
// equivalent to date1 + 15 #h
date var1 <- date('2000-01-01') plus_hours 24; // var1 equals date('2000-01-02')
```

14.4.45 plus_minutes

14.4.45.1 Possible use:

- date **plus_minutes** int —> date
- **plus_minutes** (date , int) —> date

14.4.45.2 Result:

Add a given number of minutes to a date

14.4.45.3 Examples:

```
// equivalent to date1 + 5 #mn
date var1 <- date('2000-01-01') plus_minutes 5 ; // var1 equals date('2000-01-01 00:05:00')
```

14.4.46 plus_months

14.4.46.1 Possible use:

- date **plus_months** int —> date
- **plus_months** (date , int) —> date

14.4.46.2 Result:

Add a given number of months to a date

14.4.46.3 Examples:

```
date var0 <- date('2000-01-01') plus_months 5; // var0 equals date('2000-06-01')
```

14.4.47 plus_ms

14.4.47.1 Possible use:

- date **plus_ms** int —> date
- **plus_ms** (date , int) —> date

14.4.47.2 Result:

Add a given number of milliseconds to a date

14.4.47.3 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')
```

14.4.48 plus_seconds

Same signification as +

14.4.49 plus_weeks

14.4.49.1 Possible use:

- date **plus_weeks** int —> date
- **plus_weeks** (date , int) —> date

14.4.49.2 Result:

Add a given number of weeks to a date

14.4.49.3 Examples:

```
date var0 <- date('2000-01-01') plus_weeks 15; // var0 equals date('2000-04-15')
```

14.4.50 plus_years

14.4.50.1 Possible use:

- date **plus_years** int —> date
- **plus_years** (date , int) —> date

14.4.50.2 Result:

Add a given number of years to a date

14.4.50.3 Examples:

```
date var0 <- date('2000-01-01') plus_years 15; // var0 equals date('2015-01-01')
```

14.4.51 pnorm

Same signification as normal_area

14.4.52 point

14.4.52.1 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

14.4.52.2 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.

14.4.53 points_along

14.4.53.1 Possible use:

- geometry **points_along** list<float> —> list
- **points_along** (geometry , list<float>) —> list

14.4.53.2 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.

14.4.53.3 Examples:

```
list var0 <- line([{10,10},{80,80}]) points_along ([0.3, 0.5, 0.9]); // var0 equals the list of following po
```

14.4.53.4 See also:

closest_points_with, farthest_point_to, points_at, points_on,

---

14.4.54 points_at

14.4.54.1 Possible use:

- int **points_at** float —> list<point>
- **points_at** (int , float) —> list<point>

14.4.54.2 Result:

A list of left-operand number of points located at a the right-operand distance to the agent location.

14.4.54.3 Examples:

```
list<point> var0 <- 3 points_at(20.0); // var0 equals returns [pt1, pt2, pt3] with pt1, pt2 and pt3 located a
```

14.4.54.4 See also:

any_location_in, any_point_in, closest_points_with, farthest_point_to,

---

14.4.55 points_on

14.4.55.1 Possible use:

- geometry **points_on** float —> list
- **points_on** (geometry , float) —> list

14.4.55.2 Result:

A list of points of the operand-geometry distant from each other to the float right-operand .

14.4.55.3 Examples:

```
list var0 <-  square(5) points_on(2); // var0 equals a list of points belonging to the exterior ring of the sq
```

14.4.55.4 See also:

closest_points_with, farthest_point_to, points_at,

14.4.56 `poisson`

14.4.56.1 Possible use:

- **`poisson`** (`float`) —> `int`

14.4.56.2 Result:

A value from a random variable following a Poisson distribution (with the positive expected number of occurence lambda as operand).

14.4.56.3 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.

14.4.56.4 Examples:

```
int var0 <- poisson(3.5); // var0 equals a random positive integer
```

14.4.56.5 See also:

binomial, gauss,

14.4.57 `polygon`

14.4.57.1 Possible use:

- **`polygon`** (`container<agent>`) —> `geometry`

14.4.57.2 Result:

A polygon geometry from the given list of points.

14.4.57.3 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.

14.4.57.4 Examples:

```
geometry var0 <- polygon([{0,0}, {0,10}, {10,10}, {10,0}]); // var0 equals a polygon geometry composed of the
```

14.4.57.5 See also:

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

---

14.4.58 `polyhedron`

14.4.58.1 Possible use:

- `container<geometry>` **`polyhedron`** `float` —> `geometry`
- **`polyhedron`** (`container<geometry>` , `float`) —> `geometry`

14.4.58.2 Result:

A polyhedron geometry from the given list of points.

14.4.58.3 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.

14.4.58.4 Examples:

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

14.4.58.5 See also:

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

---

14.4.59 `polyline`

Same signification as line

14.4.60 polyplan

Same signification as plan

14.4.61 predecessors_of

14.4.61.1 Possible use:

- graph **predecessors_of** unknown —> list
- **predecessors_of** (graph , unknown) —> list

14.4.61.2 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)

14.4.61.3 Examples:

```
list var1 <- graphEpidemio predecessors_of ({1,5}); // var1 equals []
list var2 <- graphEpidemio predecessors_of node({34,56}); // var2 equals [{12;45}]
```

14.4.61.4 See also:

neighbors_of, successors_of,

14.4.62 predicate

14.4.62.1 Possible use:

- **predicate** (any) —> predicate

14.4.62.2 Result:

Casts the operand into the type predicate

14.4.63 predict

14.4.63.1 Possible use:

- regression **predict** list<float> —> float
- **predict** (regression , list<float>) —> float

14.4.63.2 Result:

returns the value predict by the regression parameters for a given instance. Usage: predict(regression, instance)

14.4.63.3 Examples:

```
predict(my_regression, [1,2,3])
```

14.4.64 `product`

Same signification as mul

14.4.65 `product_of`

14.4.65.1 Possible use:

- `container` **`product_of`** `any expression` —> `unknown`
- **`product_of`** (`container` , `any expression`) —> `unknown`

14.4.65.2 Result:

the product of the right-hand expression evaluated on each of the elements of the left-hand operand

14.4.65.3 Comment:

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

14.4.65.4 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
```

14.4.65.5 Examples:

```
unknown var0 <- [1,2] product_of (each * 10 ); // var0 equals 200
```

14.4.65.6 See also:

min_of, max_of, sum_of, mean_of,

14.4.66 promethee_DM

14.4.66.1 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

14.4.66.2 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.

14.4.66.3 Special cases:

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

14.4.66.4 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,
```

14.4.66.5 See also:

weighted_means_DM, electre_DM, evidence_theory_DM,

14.4.67 property_file

14.4.67.1 Possible use:

- **property_file** (string) —> file

14.4.67.2 Result:

Constructs a file of type property. Allowed extensions are limited to properties

---

14.4.68 pValue_for_fStat

14.4.68.1 Possible use:

- **pValue_for_fStat** (float, int, int) —> float

14.4.68.2 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.

14.4.69 pValue_for_tStat

14.4.69.1 Possible use:

- float **pValue_for_tStat** int —> float
- **pValue_for_tStat** (float , int) —> float

14.4.69.2 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|.

14.4.70 pyramid

14.4.70.1 Possible use:

- **pyramid** (float) —> geometry

14.4.70.2 Result:

A square geometry which side size is given by the operand.

14.4.70.3 Comment:

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

14.4.70.4 Special cases:

- returns nil if the operand is nil.

14.4.70.5 Examples:

```
geometry var0 <- pyramid(5); // var0 equals a geometry as a square with side_size = 5.
```

14.4.70.6 See also:

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

---

14.4.71 quantile

14.4.71.1 Possible use:

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

14.4.71.2 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.

14.4.72 quantile_inverse

14.4.72.1 Possible use:

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

14.4.72.2 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.

14.4.73 R_correlation

Same signification as corR

14.4.74 R_file

14.4.74.1 Possible use:

- **`R_file`** (`string`) —> `file`

14.4.74.2 Result:

Constructs a file of type R. Allowed extensions are limited to r

---

14.4.75 `R_mean`

Same signification as meanR

14.4.76 `range`

14.4.76.1 Possible use:

- **`range`** (`int`) —> `list`
- `int` **`range`** `int` —> `list`
- **`range`** (`int` , `int`) —> `list`
- **`range`** (`int`, `int`, `int`) —> `list`

14.4.76.2 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

---

14.4.77 `rank_interpolated`

14.4.77.1 Possible use:

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

14.4.77.2 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.

14.4.78 `read`

14.4.78.1 Possible use:

- **`read`** (`string`) —> `unknown`

14.4.78.2 Result:

Reads an attribute of the agent. The attribute's name is specified by the operand.

14.4.78.3 Examples:

```
unknown
agent_name <- read ('name'); //agent_name equals reads the 'name' variable of agent then assigns the returne
```

14.4.79 `rectangle`

14.4.79.1 Possible use:

- **`rectangle`** (`point`) —> `geometry`
- `float` **`rectangle`** `float` —> `geometry`
- **`rectangle`** (`float` , `float`) —> `geometry`
- `point` **`rectangle`** `point` —> `geometry`
- **`rectangle`** (`point` , `point`) —> `geometry`

14.4.79.2 Result:

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

14.4.79.3 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.

14.4.79.4 Special cases:

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

14.4.79.5 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} a
```

14.4.79.6 See also:

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

---

14.4.80 reduced_by

Same signification as -

14.4.81 regression

14.4.81.1 Possible use:

- **regression** (any) —> regression

14.4.81.2 Result:

Casts the operand into the type regression

14.4.82 remove_duplicates

Same signification as distinct

14.4.83 remove_node_from

14.4.83.1 Possible use:

- geometry **remove_node_from** graph —> graph
- **remove_node_from** (geometry , graph) —> graph

14.4.83.2 Result:

removes a node from a graph.

14.4.83.3 Comment:

all the edges containing this node are also removed.

14.4.83.4 Examples:

```
graph var0 <- node(0) remove_node_from graphEpidemio; // var0 equals the graph without node(0)
```

14.4.84 replace

14.4.84.1 Possible use:

- **replace** (string, string, string) —> string

14.4.84.2 Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding the second operand by the third operand

14.4.84.3 Examples:

```
string var0 <- replace('to be or not to be,that is the question','to', 'do'); // var0 equals 'do be or not do 
```

14.4.84.4 See also:

replace_regex,

14.4.85 replace_regex

14.4.85.1 Possible use:

- **replace_regex** (string, string, string) —> string

14.4.85.2 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

14.4.85.3 Examples:

```
string var0 <- replace_regex("colour, color", "colou?r", "col"); // var0 equals 'col, col'
```

14.4.85.4 See also:

replace,

14.4.86 `reverse`

14.4.86.1 Possible use:

- **`reverse`** (`msi.gama.util.GamaMap<K,V>`) —> `container`
- **`reverse`** (`container<KeyType,ValueType>`) —> `container`
- **`reverse`** (`string`) —> `string`

14.4.86.2 Result:

the operand elements in the reversed order in a copy of the operand.

14.4.86.3 Comment:

the reverse operator behavior depends on the nature of the operand

14.4.86.4 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([["c1
```

- if it is a string, reverse returns a new string with characters in the reversed order

```
string var3 <- reverse ('abcd'); // var3 equals 'dcba'
```

14.4.87 `rewire_n`

14.4.87.1 Possible use:

- `graph` **`rewire_n`** `int` —> `graph`
- **`rewire_n`** (`graph` , `int`) —> `graph`

14.4.87.2 Result:

rewires the given count of edges.

14.4.87.3 Comment:

If there are too many edges, all the edges will be rewired.

14.4.87.4 Examples:

```
graph var1 <- graphEpidemio rewire_n 10; // var1 equals the graph with 3 edges rewired
```

14.4.88 `rgb`

14.4.88.1 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`

14.4.88.2 Result:

Returns a color defined by red, green, blue components and an alpha blending value.

14.4.88.3 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)

14.4.88.4 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)
```

14.4.88.5 See also:

hsb,

14.4.89 `rgb`

14.4.89.1 Possible use:

- **`rgb`** (`any`) —> `rgb`

14.4.89.2 Result:

Casts the operand into the type rgb

14.4.90 `rgb_to_xyz`

14.4.90.1 Possible use:

- **`rgb_to_xyz`** (`file`) —> `list<point>`

14.4.90.2 Result:

A list of point corresponding to RGB value of an image (r:x , g:y, b:z)

14.4.90.3 Examples:

```
list<point> var0 <- rgb_to_xyz(texture); // var0 equals a list of points
```

14.4.91 `rms`

14.4.91.1 Possible use:

- `int` **`rms`** `float` —> `float`
- **`rms`** (`int` , `float`) —> `float`

14.4.91.2 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.

14.4.92 `rnd`

14.4.92.1 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`

14.4.92.2 Result:

a random integer in the interval [0, operand]

14.4.92.3 Comment:

to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision

14.4.92.4 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]

14.4.92.5 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
```

14.4.92.6 See also:

flip,

14.4.93 `rnd_choice`

14.4.93.1 Possible use:

- **`rnd_choice`** (`list`) —> `int`

14.4.93.2 Result:

returns an index of the given list with a probability following the (normalized) distribution described in the list (a form of lottery)

14.4.93.3 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
```

14.4.93.4 See also:

rnd,

14.4.94 `rnd_color`

14.4.94.1 Possible use:

- **`rnd_color`** (`int`) —> `rgb`
- `int` **`rnd_color`** `int` —> `rgb`
- **`rnd_color`** (`int` , `int`) —> `rgb`

14.4.94.2 Result:

rgb color rgb color

14.4.94.3 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))

14.4.94.4 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))
```

14.4.94.5 See also:

rgb, hsb,

14.4.95 `rotated_by`

14.4.95.1 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`

14.4.95.2 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)

14.4.95.3 Comment:

the right-hand operand can be a float or a int

14.4.95.4 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
```

14.4.95.5 See also:

transformed_by, translated_by,

14.4.96 `round`

14.4.96.1 Possible use:

- **`round`** (`int`) —> `int`
- **`round`** (`point`) —> `point`
- **`round`** (`float`) —> `int`

14.4.96.2 Result:

Returns the rounded value of the operand.

14.4.96.3 Special cases:

- if the operand is an int, round returns it

14.4.96.4 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
```

14.4.96.5 See also:

round, int, with_precision,

14.4.97 `row_at`

14.4.97.1 Possible use:

- `matrix` **`row_at`** `int` —> `list`
- **`row_at`** (`matrix` , `int`) —> `list`

14.4.97.2 Result:

returns the row at a num_line (right-hand operand)

14.4.97.3 Examples:

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

14.4.97.4 See also:

column_at, columns_list,

14.4.98 `rows_list`

14.4.98.1 Possible use:

- **`rows_list`** (`matrix`) —> `list<list>`

14.4.98.2 Result:

returns a list of the rows of the matrix, with each row as a list of elements

14.4.98.3 Examples:

```
list<list> var0 <- rows_list(matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]
```

14.4.98.4 See also:

columns_list,

Chapter 15

Operators (S to Z)

15.1 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.

15.2 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.

15.3 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).

15.4 Operators

15.4.1 `sample`

15.4.1.1 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`

15.4.1.2 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

15.4.1.3 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]
```

15.4.2 `Sanction`

15.4.2.1 Possible use:

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

15.4.2.2 Result:

Casts the operand into the type Sanction

15.4.3 `saveSimulation`

15.4.3.1 Possible use:

- **`saveSimulation`** (`string`) —> `int`

15.4.4 `scaled_by`

Same signification as *

15.4.5 scaled_to

15.4.5.1 Possible use:

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

15.4.5.2 Result:

allows to restrict the size of a geometry so that it fits in the envelope {width, height, depth} defined by the second operand

15.4.5.3 Examples:

```
geometry var0 <- shape scaled_to {10,10}; // var0 equals a geometry corresponding to the geometry of the agen
```

15.4.6 select

Same signification as where

15.4.7 serialize

15.4.7.1 Possible use:

- **serialize** (unknown) —> string

15.4.7.2 Result:

It serializes any object, i.e. transform it into a string.

15.4.8 serializeAgent

15.4.8.1 Possible use:

- **serializeAgent** (agent) —> string

15.4.9 set_about

15.4.9.1 Possible use:

- emotion **set_about** predicate —> emotion
- **set_about** (emotion , predicate) —> emotion

15.4.9.2 Result:

change the about value of the given emotion

15.4.9.3 Examples:

```
emotion set_about predicate1
```

15.4.10 set_agent

15.4.10.1 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

15.4.10.2 Result:

change the agent value of the given social link

15.4.10.3 Examples:

```
social_link set_agent agentA
```

15.4.11 set_agent_cause

15.4.11.1 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

15.4.11.2 Result:

change the agentCause value of the given predicate change the agentCause value of the given emotion

15.4.11.3 Examples:

```
predicate set_agent_cause agentA emotion set_agent_cause agentA
```

15.4.12 `set_decay`

15.4.12.1 Possible use:

- `emotion` **`set_decay`** `float` —> `emotion`
- **`set_decay`** (`emotion` , `float`) —> `emotion`

15.4.12.2 Result:

change the decay value of the given emotion

15.4.12.3 Examples:

```
emotion set_decay 12
```

15.4.13 `set_dominance`

15.4.13.1 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`

15.4.13.2 Result:

change the dominance value of the given social link

15.4.13.3 Examples:

```
social_link set_dominance 0.4
```

15.4.14 `set_familiarity`

15.4.14.1 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`

15.4.14.2 Result:

change the familiarity value of the given social link

15.4.14.3 Examples:

```
social_link set_familiarity 0.4
```

15.4.15 set_intensity

15.4.15.1 Possible use:

- `emotion` **`set_intensity`** `float` —> `emotion`
- **`set_intensity`** (`emotion` , `float`) —> `emotion`

15.4.15.2 Result:

change the intensity value of the given emotion

15.4.15.3 Examples:

```
emotion set_intensity 12
```

15.4.16 set_lifetime

15.4.16.1 Possible use:

- `mental_state` **`set_lifetime`** `int` —> `mental_state`
- **`set_lifetime`** (`mental_state` , `int`) —> `mental_state`

15.4.16.2 Result:

change the lifetime value of the given mental state

15.4.16.3 Examples:

```
mental state set_lifetime 1
```

15.4.17 set_liking

15.4.17.1 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

15.4.17.2 Result:

change the liking value of the given social link

15.4.17.3 Examples:

```
social_link set_liking 0.4
```

15.4.18 set_modality

15.4.18.1 Possible use:

- mental_state **set_modality** string —> mental_state
- **set_modality** (mental_state , string) —> mental_state

15.4.18.2 Result:

change the modality value of the given mental state

15.4.18.3 Examples:

```
mental state set_modality belief
```

15.4.19 set_predicate

15.4.19.1 Possible use:

- mental_state **set_predicate** predicate —> mental_state
- **set_predicate** (mental_state , predicate) —> mental_state

15.4.19.2 Result:

change the predicate value of the given mental state

15.4.19.3 Examples:

```
mental state set_predicate pred1
```

15.4.20 `set_solidarity`

15.4.20.1 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`

15.4.20.2 Result:

change the solidarity value of the given social link

15.4.20.3 Examples:

```
social_link set_solidarity 0.4
```

15.4.21 `set_strength`

15.4.21.1 Possible use:

- `mental_state` **`set_strength`** `float` —> `mental_state`
- **`set_strength`** (`mental_state` , `float`) —> `mental_state`

15.4.21.2 Result:

change the strength value of the given mental state

15.4.21.3 Examples:

```
mental state set_strength 1.0
```

15.4.22 `set_trust`

15.4.22.1 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`

15.4.22.2 Result:

change the trust value of the given social link

15.4.22.3 Examples:

```
social_link set_familiarity 0.4
```

15.4.23 `set_truth`

15.4.23.1 Possible use:

- `predicate` **`set_truth`** `bool` —> `predicate`
- **`set_truth`** (`predicate` , `bool`) —> `predicate`

15.4.23.2 Result:

change the is_true value of the given predicate

15.4.23.3 Examples:

```
predicate set_truth false
```

15.4.24 `set_z`

15.4.24.1 Possible use:

- `geometry` **`set_z`** `container<float>` —> `geometry`
- **`set_z`** (`geometry` , `container<float>`) —> `geometry`
- **`set_z`** (`geometry`, `int`, `float`) —> `geometry`

15.4.24.2 Result:

Sets the z ordinate of the n-th point of a geometry to the value provided by the third argument

15.4.24.3 Examples:

```
loop i from: 0 to: length(shape.points) - 1{set shape <-  set_z (shape, i, 3.0);} shape <- triangle(3) set_z [
```

15.4.25 `shape_file`

15.4.25.1 Possible use:

- **`shape_file`** (`string`) —> `file`

15.4.25.2 Result:

Constructs a file of type shape. Allowed extensions are limited to shp

---

15.4.26 `shuffle`

15.4.26.1 Possible use:

- **`shuffle`** (`container`) —> `list`
- **`shuffle`** (`matrix`) —> `matrix`
- **`shuffle`** (`string`) —> `string`

15.4.26.2 Result:

The elements of the operand in random order.

15.4.26.3 Special cases:

- if the operand is empty, returns an empty list (or string, matrix)

15.4.26.4 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)
```

15.4.26.5 See also:

reverse,

15.4.27 `signum`

15.4.27.1 Possible use:

- **`signum`** (`float`) —> `int`

15.4.27.2 Result:

Returns -1 if the argument is negative, +1 if it is positive, 0 if it is equal to zero or not a number

15.4.27.3 Examples:

```
int var0 <- signum(-12); // var0 equals -1
int var1 <- signum(14); // var1 equals 1
int var2 <- signum(0); // var2 equals 0
```

15.4.28 simple_clustering_by_distance

15.4.28.1 Possible use:

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

15.4.28.2 Result:

A list of agent groups clustered by distance considering a distance min between two groups.

15.4.28.3 Examples:

```
list<list<agent>> var0 <- [ag1, ag2, ag3, ag4, ag5] simpleClusteringByDistance 20.0; // var0 equals for exam
```

15.4.28.4 See also:

hierarchical_clustering,

15.4.29 simple_clustering_by_envelope_distance

Same signification as simple_clustering_by_distance

15.4.30 simplex_generator

15.4.30.1 Possible use:

- **simplex_generator** (float, float, float) —> float

15.4.30.2 Result:

take a x, y and a bias parameters and gives a value

15.4.30.3 Examples:

```
float var0 <- simplex_generator(2,3,253); // var0 equals 10.2
```

15.4.31 `simplification`

15.4.31.1 Possible use:

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

15.4.31.2 Result:

A geometry corresponding to the simplification of the operand (geometry, agent, point) considering a tolerance distance.

15.4.31.3 Comment:

The algorithm used for the simplification is Douglas-Peucker

15.4.31.4 Examples:

```
geometry var0 <- self simplification 0.1; // var0 equals the geometry resulting from the application of the D
```

15.4.32 `sin`

15.4.32.1 Possible use:

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

15.4.32.2 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.

15.4.32.3 Special cases:

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

15.4.32.4 Examples:

```
float var0 <- sin (0); // var0 equals 0.0
float var1 <- sin(360) with_precision 10 with_precision 10; // var1 equals 0.0
```

15.4.32.5 See also:

cos, tan,

15.4.33 sin_rad

15.4.33.1 Possible use:

- **sin_rad** (float) —> float

15.4.33.2 Result:

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

15.4.33.3 Examples:

```
float var0 <- sin_rad(#pi); // var0 equals 0.0
```

15.4.33.4 See also:

cos_rad, tan_rad,

15.4.34 since

15.4.34.1 Possible use:

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

15.4.34.2 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

15.4.34.3 Examples:

```
reflex when: since(starting_date) {}   // this reflex will always be run every(2#days) since (starting_date
```

15.4.35 skeletonize

15.4.35.1 Possible use:

- **skeletonize** (geometry) —> list<geometry>
- geometry **skeletonize** float —> list<geometry>
- **skeletonize** (geometry , float) —> list<geometry>
- **skeletonize** (geometry, float, float) —> list<geometry>

15.4.35.2 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)

15.4.35.3 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
```

15.4.36 skew

15.4.36.1 Possible use:

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

15.4.36.2 Result:

Returns the skew of a data sequence. Returns the skew of a data sequence, which is moment(data,3,mean) / standardDeviation3

15.4.37 skew_gauss

15.4.37.1 Possible use:

- **skew_gauss** (float, float, float, float) —> float

15.4.37.2 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

15.4.37.3 Examples:

```
float var0 <- skew_gauss(0.0, 1.0, 0.7,0.1); // var0 equals 0.1729218460343077
```

15.4.37.4 See also:

gauss, truncated_gauss, poisson,

15.4.38 skewness

15.4.38.1 Possible use:

- `skewness` (`list`) —> `float`

15.4.38.2 Result:

returns skewness value computed from the operand list of values

15.4.38.3 Special cases:

- if the length of the list is lower than 3, returns NaN

15.4.38.4 Examples:

```
float var0 <- skewness ([1,2,3,4,5]); // var0 equals 0.0
```

15.4.39 skill

15.4.39.1 Possible use:

- `skill` (`any`) —> `skill`

15.4.39.2 Result:

Casts the operand into the type skill

15.4.40 smooth

15.4.40.1 Possible use:

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

15.4.40.2 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).

15.4.40.3 Examples:

```
geometry var0 <- smooth(square(10), 0.0); // var0 equals a 'rounded' square
```

15.4.41 social_link

15.4.41.1 Possible use:

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

15.4.41.2 Result:

Casts the operand into the type social_link

15.4.42 solid

Same signification as without_holes

15.4.43 sort

Same signification as sort_by

15.4.44 sort_by

15.4.44.1 Possible use:

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

15.4.44.2 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.

15.4.44.3 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.

15.4.44.4 Special cases:

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

15.4.44.5 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]
```

15.4.44.6 See also:

group_by,

15.4.45 source_of

15.4.45.1 Possible use:

- graph **source_of** unknown —> unknown
- **source_of** (graph , unknown) —> unknown

15.4.45.2 Result:

returns the source of the edge (right-hand operand) contained in the graph given in left-hand operand.

15.4.45.3 Special cases:

- if the lef-hand operand (the graph) is nil, throws an Exception

15.4.45.4 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}
```

15.4.45.5 See also:

target_of,

15.4.46 `spatial_graph`

15.4.46.1 Possible use:

- **`spatial_graph`** (`container`) —> `graph`

15.4.46.2 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

15.4.46.3 See also:

graph,

15.4.47 `species`

15.4.47.1 Possible use:

- **`species`** (`unknown`) —> `species`

15.4.47.2 Result:

casting of the operand to a species.

15.4.47.3 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

15.4.47.4 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
```

15.4.48 `species_of`

Same signification as species

15.4.49 `sphere`

15.4.49.1 Possible use:

- **`sphere`** (`float`) —> `geometry`

15.4.49.2 Result:

A sphere geometry which radius is equal to the operand.

15.4.49.3 Comment:

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

15.4.49.4 Special cases:

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

15.4.49.5 Examples:

```
geometry var0 <- sphere(10); // var0 equals a geometry as a circle of radius 10 but displays a sphere.
```

15.4.49.6 See also:

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

---

15.4.50 split

15.4.50.1 Possible use:

- **split** (list) —> list<list>

15.4.50.2 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

15.4.50.3 See also:

split_in, split_using,

15.4.51 split_at

15.4.51.1 Possible use:

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

15.4.51.2 Result:

The two part of the left-operand lines split at the given right-operand point

15.4.51.3 Special cases:

- if the left-operand is a point or a polygon, returns an empty list

15.4.51.4 Examples:

```
list<geometry> var0 <- polyline([{1,2},{4,6}]) split_at {7,6}; // var0 equals [polyline([{1.0,2.0},{7.0,6.0
```

15.4.52 split_geometry

15.4.52.1 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>

15.4.52.2 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})

15.4.52.3 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
```

15.4.53 split_in

15.4.53.1 Possible use:

- `list` **`split_in`** `int` —> `list<list>`
- **`split_in`** (`list` , `int`) —> `list<list>`
- **`split_in`** (`list`, `int`, `bool`) —> `list<list>`

15.4.53.2 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

15.4.53.3 See also:

split, split_using,

15.4.54 split_lines

15.4.54.1 Possible use:

- **`split_lines`** (`container<geometry>`) —> `list<geometry>`
- `container<geometry>` **`split_lines`** `bool` —> `list<geometry>`
- **`split_lines`** (`container<geometry>` , `bool`) —> `list<geometry>`

15.4.54.2 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

15.4.54.3 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
```

15.4.55 `split_using`

15.4.55.1 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>`

15.4.55.2 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

15.4.55.3 See also:

split, split_in,

15.4.56 `split_with`

15.4.56.1 Possible use:

- `string` **`split_with`** `string` —> `list`
- **`split_with`** (`string` , `string`) —> `list`
- **`split_with`** (`string`, `string`, `bool`) —> `list`

15.4.56.2 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).

15.4.56.3 Comment:

Delimiters themselves are excluded from the resulting list.Delimiters themselves are excluded from the resulting list.

15.4.56.4 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']
```

15.4.57 sqrt

15.4.57.1 Possible use:

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

15.4.57.2 Result:

Returns the square root of the operand.

15.4.57.3 Special cases:

- if the operand is negative, an exception is raised

15.4.57.4 Examples:

```
float var0 <- sqrt(4); // var0 equals 2.0
float var1 <- sqrt(4); // var1 equals 2.0
```

15.4.58 square

15.4.58.1 Possible use:

- **square** (`float`) —> `geometry`

15.4.58.2 Result:

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

15.4.58.3 Comment:

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

15.4.58.4 Special cases:

- returns nil if the operand is nil.

15.4.58.5 Examples:

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

15.4.58.6 See also:

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

---

15.4.59 `squircle`

15.4.59.1 Possible use:

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

15.4.59.2 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

15.4.59.3 Comment:

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

15.4.59.4 Special cases:

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

15.4.59.5 Examples:

```
geometry var0 <- squircle(4,4); // var0 equals a geometry as a squircle of side 4 with a power of 4.
```

15.4.59.6 See also:

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

---

15.4.60 `stack`

15.4.60.1 Possible use:

- **`stack`** (`msi.gama.util.IList<java.lang.Integer>`) —> `msi.gama.util.tree.GamaNode<java.lang.String>`

15.4.61 `standard_deviation`

15.4.61.1 Possible use:

- **`standard_deviation`** (`container`) —> `float`

15.4.61.2 Result:

the standard deviation on the elements of the operand. See Standard_deviation for more details.

15.4.61.3 Comment:

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

15.4.61.4 Examples:

```
float var0 <- standard_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.2930100540985752
```

15.4.61.5 See also:

mean, mean_deviation,

15.4.62 `step_sub_model`

15.4.62.1 Possible use:

- **`step_sub_model`** (`msi.gama.kernel.experiment.IExperimentAgent`) —> `int`

15.4.62.2 Result:

Load a submodel

15.4.62.3 Comment:

loaded submodel

15.4.63 strahler

15.4.63.1 Possible use:

- **strahler** (graph) —> map

15.4.63.2 Result:

retur for each edge, its strahler number

15.4.64 string

15.4.64.1 Possible use:

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

15.4.64.2 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

15.4.64.3 Examples:

```
format(#now, 'yyyy-MM-dd') format(#now, 'yyyy-MM-dd')
```

15.4.65 student_area

15.4.65.1 Possible use:

- float **student_area** int —> float
- **student_area** (float , int) —> float

15.4.65.2 Result:

Returns the area to the left of x in the Student T distribution with the given degrees of freedom.

15.4.66 `student_t_inverse`

15.4.66.1 Possible use:

- `float` **`student_t_inverse`** `int` —> `float`
- **`student_t_inverse`** (`float` , `int`) —> `float`

15.4.66.2 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.

15.4.67 `subtract_days`

Same signification as minus_days

15.4.68 `subtract_hours`

Same signification as minus_hours

15.4.69 `subtract_minutes`

Same signification as minus_minutes

15.4.70 `subtract_months`

Same signification as minus_months

15.4.71 `subtract_ms`

Same signification as minus_ms

15.4.72 `subtract_seconds`

Same signification as -

15.4.73 `subtract_weeks`

Same signification as minus_weeks

15.4.74 `subtract_years`

Same signification as minus_years

15.4.75 `successors_of`

15.4.75.1 Possible use:

- `graph` **`successors_of`** `unknown` —> `list`
- **`successors_of`** (`graph` , `unknown`) —> `list`

15.4.75.2 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)

15.4.75.3 Examples:

```
list var1 <- graphEpidemio successors_of ({1,5}); // var1 equals [{12,45}]
list var2 <- graphEpidemio successors_of node({34,56}); // var2 equals []
```

15.4.75.4 See also:

predecessors_of, neighbors_of,

15.4.76 `sum`

15.4.76.1 Possible use:

- **`sum`** (`container`) —> `unknown`
- **`sum`** (`graph`) —> `float`

15.4.76.2 Result:

the sum of all the elements of the operand

15.4.76.3 Comment:

the behavior depends on the nature of the operand

15.4.76.4 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}
```

15.4.76.5 See also:

mul,

15.4.77 `sum_of`

15.4.77.1 Possible use:

- `container` **`sum_of`** `any expression` —> `unknown`
- **`sum_of`** (`container` , `any expression`) —> `unknown`

15.4.77.2 Result:

the sum of the right-hand expression evaluated on each of the elements of the left-hand operand

15.4.77.3 Comment:

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

15.4.77.4 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
```

15.4.77.5 Examples:

```
unknown var0 <- [1,2] sum_of (each * 100 ); // var0 equals 300
```

15.4.77.6 See also:

min_of, max_of, product_of, mean_of,

15.4.78 `svg_file`

15.4.78.1 Possible use:

- **`svg_file`** (`string`) —> `file`

15.4.78.2 Result:

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

---

15.4.79 `tan`

15.4.79.1 Possible use:

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

15.4.79.2 Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in decimal degrees).

15.4.79.3 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).

15.4.79.4 Examples:

```
float var0 <- tan (0); // var0 equals 0.0
float var1 <- tan(90); // var1 equals 1.633123935319537E16
```

15.4.79.5 See also:

cos, sin,

15.4.80 `tan_rad`

15.4.80.1 Possible use:

- **`tan_rad`** (`float`) —> `float`

15.4.80.2 Result:

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

15.4.80.3 See also:

cos_rad, sin_rad,

15.4.81 `tanh`

15.4.81.1 Possible use:

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

15.4.81.2 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).

15.4.81.3 Examples:

```
float var0 <- tanh(0); // var0 equals 0.0
float var1 <- tanh(100); // var1 equals 1.0
```

15.4.82 target_of

15.4.82.1 Possible use:

- graph **target_of** unknown —> unknown
- **target_of** (graph , unknown) —> unknown

15.4.82.2 Result:

returns the target of the edge (right-hand operand) contained in the graph given in left-hand operand.

15.4.82.3 Special cases:

- if the lef-hand operand (the graph) is nil, returns nil

15.4.82.4 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}
```

15.4.82.5 See also:

source_of,

15.4.83 teapot

15.4.83.1 Possible use:

- **teapot** (float) —> geometry

15.4.83.2 Result:

A teapot geometry which radius is equal to the operand.

15.4.83.3 Comment:

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

15.4.83.4 Special cases:

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

15.4.83.5 Examples:

```
geometry var0 <- teapot(10); // var0 equals a geometry as a circle of radius 10 but displays a teapot.
```

15.4.83.6 See also:

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

---

15.4.84 `text_file`

15.4.84.1 Possible use:

- **`text_file`** (`string`) —> `file`

15.4.84.2 Result:

Constructs a file of type text. Allowed extensions are limited to txt, data, text

---

15.4.85 `TGauss`

Same signification as truncated_gauss

15.4.86 `threeds_file`

15.4.86.1 Possible use:

- **`threeds_file`** (`string`) —> `file`

15.4.86.2 Result:

Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

---

15.4.87 `to`

Same signification as until

15.4.87.1 Possible use:

- date **to** date —> msi.gama.util.IList<msi.gama.util.GamaDate>
- **to** (date , date) —> msi.gama.util.IList<msi.gama.util.GamaDate>

15.4.87.2 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

15.4.87.3 Comment:

The default step can be overruled by using the every operator applied to this interval

15.4.87.4 Examples:

```
date('2000-01-01') to date('2010-01-01') // builds an interval between these two dates (date('2000-01-01')
```

15.4.87.5 See also:

every,

15.4.88 to_GAMA_CRS

15.4.88.1 Possible use:

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

15.4.88.2 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
```

15.4.89 to_gaml

15.4.89.1 Possible use:

- **to_gaml** (unknown) —> string

15.4.89.2 Result:

returns the literal description of an expression or description – action, behavior, species, aspect, even model – in gaml

15.4.89.3 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
```

15.4.90 to_rectangles

15.4.90.1 Possible use:

- **to_rectangles** (geometry, point, bool) —> list<geometry>
- **to_rectangles** (geometry, int, int, bool) —> list<geometry>

15.4.90.2 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

15.4.90.3 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
```

15.4.91 `to_squares`

15.4.91.1 Possible use:

- **`to_squares`** (`geometry`, `int`, `bool`) —> `list<geometry>`
- **`to_squares`** (`geometry`, `float`, `bool`) —> `list<geometry>`
- **`to_squares`** (`geometry`, `int`, `bool`, `float`) —> `list<geometry>`

15.4.91.2 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

15.4.91.3 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
```

15.4.92 `to_sub_geometries`

15.4.92.1 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>`

15.4.92.2 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.

15.4.92.3 Examples:

```
list<geometry> var0 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4], 1.0); // var0 equals a list of t
list<geometry> var1 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4]); // var1 equals a list of three
```

15.4.93 `to_triangles`

Same signification as triangulate

15.4.94 `tokenize`

Same signification as split_with

15.4.95 `topology`

15.4.95.1 Possible use:

- **`topology`** (`unknown`) —> `topology`

15.4.95.2 Result:

casting of the operand to a topology.

15.4.95.3 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.

15.4.95.4 Examples:

```
topology var0 <- topology(0); // var0 equals niltopology(a_graph)  --: Multiple topology in POLYGON ((24.71
```

15.4.95.5 See also:

geometry,

15.4.96 topology

15.4.96.1 Possible use:

- **topology** (any) —> topology

15.4.96.2 Result:

Casts the operand into the type topology

15.4.97 touches

15.4.97.1 Possible use:

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

15.4.97.2 Result:

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

15.4.97.3 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.

15.4.97.4 Special cases:

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

15.4.97.5 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
```

15.4.97.6 See also:

disjoint_from, crosses, overlaps, partially_overlaps, intersects,

---

15.4.98 towards

15.4.98.1 Possible use:

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

15.4.98.2 Result:

The direction (in degree) between the two geometries (geometries, agents, points) considering the topology of the agent applying the operator.

15.4.98.3 Examples:

```
float var0 <- ag1 towards ag2; // var0 equals the direction between ag1 and ag2 and ag3 considering the topolo
```

15.4.98.4 See also:

distance_between, distance_to, direction_between, path_between, path_to,

---

15.4.99 trace

15.4.99.1 Possible use:

- **`trace`** (`matrix`) —> `float`

15.4.99.2 Result:

The trace of the given matrix (the sum of the elements on the main diagonal).

15.4.99.3 Examples:

```
float var0 <- trace(matrix([[1,2],[3,4]])); // var0 equals 5
```

15.4.100 transformed_by

15.4.100.1 Possible use:

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

15.4.100.2 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)

15.4.100.3 Examples:

```
geometry var0 <- self transformed_by {45, 0.5}; // var0 equals the geometry resulting from 45 degrees rotatio
```

15.4.100.4 See also:

rotated_by, translated_by,

15.4.101 translated_by

15.4.101.1 Possible use:

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

15.4.101.2 Result:

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

15.4.101.3 Examples:

```
geometry var0 <- self translated_by {10,10,10}; // var0 equals the geometry resulting from applying the trans
```

15.4.101.4 See also:

rotated_by, transformed_by,

15.4.102 `translated_to`

Same signification as at_location

15.4.103 `transpose`

15.4.103.1 Possible use:

- **`transpose`** (`matrix`) —> `matrix`

15.4.103.2 Result:

The transposition of the given matrix

15.4.103.3 Examples:

```
matrix var0 <- transpose(matrix([[5,-3],[6,-4]])); // var0 equals matrix([[5,6],[-3,-4]])
```

15.4.104 `triangle`

15.4.104.1 Possible use:

- **`triangle`** (`float`) —> `geometry`

15.4.104.2 Result:

A triangle geometry which side size is given by the operand.

15.4.104.3 Comment:

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

15.4.104.4 Special cases:

- returns nil if the operand is nil.

15.4.104.5 Examples:

```
geometry var0 <- triangle(5); // var0 equals a geometry as a triangle with side_size = 5.
```

15.4.104.6 See also:

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

---

15.4.105 triangulate

15.4.105.1 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>

15.4.105.2 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)

15.4.105.3 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
```

15.4.106 truncated_gauss

15.4.106.1 Possible use:

- **truncated_gauss** (point) —> float
- **truncated_gauss** (list) —> float

15.4.106.2 Result:

A random value from a normally distributed random variable in the interval]mean - standardDeviation; mean + standardDeviation[.

15.4.106.3 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

15.4.106.4 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
```

15.4.106.5 See also:

gauss,

15.4.107 `type_of`

15.4.107.1 Possible use:

- **`type_of`** (`unknown`) —> `msi.gaml.types.IType<?>`

15.4.108 `undirected`

15.4.108.1 Possible use:

- **`undirected`** (`graph`) —> `graph`

15.4.108.2 Result:

the operand graph becomes an undirected graph.

15.4.108.3 Comment:

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

15.4.108.4 See also:

directed,

15.4.109 union

15.4.109.1 Possible use:

- **union** (container<geometry>) —> geometry
- container **union** container —> list
- **union** (container , container) —> list

15.4.109.2 Result:

returns a new list containing all the elements of both containers without duplicated elements.

15.4.109.3 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

15.4.109.4 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]
```

15.4.109.5 See also:

inter, +,

15.4.110 unknown

15.4.110.1 Possible use:

- **unknown** (any) —> unknown

15.4.110.2 Result:

Casts the operand into the type unknown

15.4.111 unSerializeSimulation

15.4.111.1 Possible use:

- **unSerializeSimulation** (string) —> int

15.4.111.2 Result:

unSerializeSimulation

15.4.112 unSerializeSimulationFromFile

15.4.112.1 Possible use:

- **`unSerializeSimulationFromFile`** (`string`) —> `int`

15.4.113 until

15.4.113.1 Possible use:

- **`until`** (`date`) —> `bool`
- `any expression` **`until`** `date` —> `bool`
- **`until`** (`any expression` , `date`) —> `bool`

15.4.113.2 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'

15.4.113.3 Examples:

```
reflex when: until(starting_date) {}   // This reflex will be run only once at the beginning of the simulatio
```

15.4.114 upper_case

15.4.114.1 Possible use:

- **`upper_case`** (`string`) —> `string`

15.4.114.2 Result:

Converts all of the characters in the string operand to upper case

15.4.114.3 Examples:

```
string var0 <- upper_case("Abc"); // var0 equals 'ABC'
```

15.4.114.4 See also:

lower_case,

15.4.115 `URL_file`

15.4.115.1 Possible use:

- **`URL_file`** (`string`) —> `file`

15.4.115.2 Result:

Constructs a file of type URL. Allowed extensions are limited to url

---

15.4.116 `use_cache`

15.4.116.1 Possible use:

- `graph` **`use_cache`** `bool` —> `graph`
- **`use_cache`** (`graph` , `bool`) —> `graph`

15.4.116.2 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).

15.4.116.3 Comment:

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

15.4.116.4 See also:

path_between,

15.4.117 `user_input`

15.4.117.1 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>`

15.4.117.2 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.

15.4.117.3 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 :

15.4.117.4 Examples:

```
map<string,unknown> values <- user_input(["Number" :: 100, "Location" :: {10, 10}]); create bug number: int(
```

15.4.118 using

15.4.118.1 Possible use:

- `any expression` **`using`** `topology` —> `unknown`
- **`using`** (`any expression` , `topology`) —> `unknown`

15.4.118.2 Result:

Allows to specify in which topology a spatial computation should take place.

15.4.118.3 Special cases:

- has no effect if the topology passed as a parameter is nil

15.4.118.4 Examples:

```
unknown var0 <- (agents closest_to self) using topology(world); // var0 equals the closest agent to self (the
```

15.4.119 variance

15.4.119.1 Possible use:

- **`variance`** (`container`) —> `float`

15.4.119.2 Result:

the variance of the elements of the operand. See Variance for more details.

15.4.119.3 Comment:

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

15.4.119.4 Examples:

```
float var0 <- variance ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.671875
```

15.4.119.5 See also:

mean, median,

15.4.120 `variance`

15.4.120.1 Possible use:

- **`variance`** (`float`) —> `float`
- **`variance`** (`int`, `float`, `float`) —> `float`

15.4.120.2 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.

15.4.121 `variance_of`

15.4.121.1 Possible use:

- `container` **`variance_of`** `any expression` —> `unknown`
- **`variance_of`** (`container` , `any expression`) —> `unknown`

15.4.121.2 Result:

the variance of the right-hand expression evaluated on each of the elements of the left-hand operand

15.4.121.3 Comment:

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

15.4.121.4 See also:

min_of, max_of, sum_of, product_of,

15.4.122 vertical

15.4.122.1 Possible use:

- **vertical** (msi.gama.util.GamaMap<java.lang.Object,java.lang.Integer>) —> msi.gama.util.tree.GamaNode

---

15.4.123 voronoi

15.4.123.1 Possible use:

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

15.4.123.2 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

15.4.123.3 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
```

15.4.124 weight_of

15.4.124.1 Possible use:

- graph **weight_of** unknown —> float
- **weight_of** (graph , unknown) —> float

15.4.124.2 Result:

returns the weight of the given edge (right-hand operand) contained in the graph given in right-hand operand.

15.4.124.3 Comment:

In a localized graph, an edge has a weight by default (the distance between both vertices).

15.4.124.4 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.

15.4.124.5 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
```

15.4.125 `weighted_means_DM`

15.4.125.1 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`

15.4.125.2 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)

15.4.125.3 Special cases:

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

15.4.125.4 Examples:

```
int var0 <- weighted_means_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"utility", "weight" :: 2.0],["na
```

15.4.125.5 See also:

promethee_DM, electre_DM, evidence_theory_DM,

15.4.126 `where`

15.4.126.1 Possible use:

- `container` **`where`** `any expression` —> `list`
- **`where`** (`container` , `any expression`) —> `list`

15.4.126.2 Result:

a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.

15.4.126.3 Comment:

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

15.4.126.4 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]
```

15.4.126.5 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]
```

15.4.126.6 See also:

first_with, last_with, where,

15.4.127 `with_lifetime`

15.4.127.1 Possible use:

- `predicate` **`with_lifetime`** `int` —> `predicate`
- **`with_lifetime`** (`predicate` , `int`) —> `predicate`

15.4.127.2 Result:

change the parameters of the given predicate

15.4.127.3 Examples:

```
predicate with_lifetime 10
```

15.4.128 with_max_of

15.4.128.1 Possible use:

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

15.4.128.2 Result:

one of elements of the left-hand operand that maximizes the value of the right-hand operand

15.4.128.3 Comment:

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

15.4.128.4 Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

15.4.128.5 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
```

15.4.128.6 See also:

where, with_min_of,

15.4.129 with_min_of

15.4.129.1 Possible use:

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

15.4.129.2 Result:

one of elements of the left-hand operand that minimizes the value of the right-hand operand

15.4.129.3 Comment:

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

15.4.129.4 Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

15.4.129.5 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
```

15.4.129.6 See also:

where, with_max_of,

15.4.130 with_optimizer_type

15.4.130.1 Possible use:

- graph **with_optimizer_type** string —> graph
- **with_optimizer_type** (graph , string) —> graph

15.4.130.2 Result:

changes the shortest path computation method of the given graph

15.4.130.3 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.

15.4.130.4 Examples:

```
graphEpidemio <- graphEpidemio with_optimizer_type "static";
```

15.4.130.5 See also:

set_verbose,

15.4.131 `with_precision`

15.4.131.1 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`

15.4.131.2 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.

15.4.131.3 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
```

15.4.131.4 See also:

round,

15.4.132 `with_values`

15.4.132.1 Possible use:

- `predicate` **`with_values`** `map` —> `predicate`
- **`with_values`** (`predicate` , `map`) —> `predicate`

15.4.132.2 Result:

change the parameters of the given predicate

15.4.132.3 Examples:

```
predicate with_values ["time"::10]
```

15.4.133 with_weights

15.4.133.1 Possible use:

- graph **with_weights** list —> graph
- **with_weights** (graph , list) —> graph
- graph **with_weights** map —> graph
- **with_weights** (graph , map) —> graph

15.4.133.2 Result:

returns the graph (left-hand operand) with weight given in the map (right-hand operand).

15.4.133.3 Comment:

this operand re-initializes the path finder

15.4.133.4 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
```

15.4.134 without_holes

15.4.134.1 Possible use:

- **without_holes** (geometry) —> geometry

15.4.134.2 Result:

A geometry corresponding to the operand geometry (geometry, agent, point) without its holes

15.4.134.3 Examples:

```
geometry var0 <- solid(self); // var0 equals the geometry corresponding to the geometry of the agent applying
```

15.4.135 `writable`

15.4.135.1 Possible use:

- `file` **`writable`** `bool` —> `file`
- **`writable`** (`file` , `bool`) —> `file`

15.4.135.2 Result:

Marks the file as read-only or not, depending on the second boolean argument, and returns the first argument

15.4.135.3 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)")

15.4.135.4 Examples:

```
file var0 <- shape_file("../images/point_eau.shp") writable false; // var0 equals returns a file in read-onl
```

15.4.135.5 See also:

file,

15.4.136 `xml_file`

15.4.136.1 Possible use:

- **`xml_file`** (`string`) —> `file`

15.4.136.2 Result:

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

---

15.4.137 `xor`

15.4.137.1 Possible use:

- `bool` **`xor`** `bool` —> `bool`
- **`xor`** (`bool` , `bool`) —> `bool`

15.4.137.2 Result:

a bool value, equal to the logical xor between the left-hand operand and the right-hand operand. False when they are equal

15.4.137.3 Comment:

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

15.4.137.4 See also:

or, and, !,

15.4.138 `years_between`

15.4.138.1 Possible use:

- `date` **`years_between`** `date` —> `int`
- **`years_between`** (`date` , `date`) —> `int`

15.4.138.2 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)

15.4.138.3 Examples:

```
int var0 <- years_between(date('2000-01-01'), date('2010-01-01')); // var0 equals 10
```

Chapter 16

Statements

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

16.1 Table of Contents

16.2 Statements by kinds

- **Batch method**
  - annealing, exhaustive, genetic, hill_climbing, reactive_tabu, tabu,
- **Behavior**
  - aspect, norm, plan, reflex, sanction, state, task, test, user_init, user_panel,
- **Behavior**
  - aspect, norm, plan, reflex, sanction, state, task, test, user_init, user_panel,
- **Experiment**
  - experiment,
- **Layer**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,
- **Output**

  - display, inspect, layout, monitor, output, output_file, permanent,

- **Parameter**
  - parameter,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Species**
  - species,

- **Variable (container)**
  - Variable_container,

- **Variable (number)**
  - Variable_number,

- **Variable (regular)**
  - Variable_regular,

16.3 Statements by embedment

- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Environment**
  - species,
- **Experiment**
  - action, annealing, exhaustive, genetic, hill_climbing, layout, output, parameter, permanent, reactive_tabu, reflex, setup, simulate, state, tabu, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Experiment**
  - action, annealing, exhaustive, genetic, hill_climbing, layout, output, parameter, permanent, reactive_tabu, reflex, setup, simulate, state, tabu, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Layer**
  - add, benchmark, draw, error, focus_on, highlight, if, let, loop, put, remove, set, status, switch, trace, try, using, warn, write,
- **Model**
  - action, aspect, equation, experiment, law, norm, output, perceive, plan, reflex, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Model**
  - action, aspect, equation, experiment, law, norm, output, perceive, plan, reflex, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Model**

  - action, aspect, equation, experiment, law, norm, output, perceive, plan, reflex, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Model**
  - action, aspect, equation, experiment, law, norm, output, perceive, plan, reflex, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Sequence of statements or action**
  - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, socialize, solve, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Sequence of statements or action**
  - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, socialize, solve, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Sequence of statements or action**
  - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, socialize, solve, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Sequence of statements or action**
  - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, socialize, solve, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Single statement**
  - run, start_simulation,
- **Species**
  - action, aspect, equation, law, norm, perceive, plan, reflex, rule, run, sanction, setup, simulate, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Species**
  - action, aspect, equation, law, norm, perceive, plan, reflex, rule, run, sanction, setup, simulate, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Species**
  - action, aspect, equation, law, norm, perceive, plan, reflex, rule, run, sanction, setup, simulate, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Species**
  - action, aspect, equation, law, norm, perceive, plan, reflex, rule, run, sanction, setup, simulate, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Species**
  - action, aspect, equation, law, norm, perceive, plan, reflex, rule, run, sanction, setup, simulate, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **action**
  - assert, return,
- **aspect**
  - draw,
- **chart**

  - add, ask, data, datalist, do, put, remove, set, simulate, using,
- **chart**
  - add, ask, data, datalist, do, put, remove, set, simulate, using,
- **display**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,
- **display_population**
  - display_population,
- **equation**
  - =,
- **fsm**
  - state, user_panel,
- **if**
  - else,
- **output**
  - display, inspect, monitor, output_file,
- **permanent**
  - display, inspect, monitor, output_file,
- **probabilistic_tasks**
  - task,
- **sorted_tasks**
  - task,
- **state**
  - enter, exit,
- **switch**
  - default, match,
- **test**
  - assert,
- **try**
  - catch,
- **user_command**
  - user_input,
- **user_first**
  - user_panel,
- **user_init**
  - user_panel,
- **user_last**
  - user_panel,
- **user_only**
  - user_panel,
- **user_panel**
  - user_command,
- **weighted_tasks**
  - task,

16.4 General syntax

A statement represents either a declaration or an imperative command. It consists in a keyword, followed by specific facets, some of them mandatory (in bold), some of them optional. One of the facet names can be omitted (the one denoted as omissible). It has to be the first one.

```
statement_keyword expression1 facet2: expression2 ... ;
or
statement_keyword facet1: expression1 facet2: expression2 ...;
```

If the statement encloses other statements, it is called a **sequence statement**, and its sub-statements (either sequence statements or single statements) are declared between curly brackets, as in:

```
statement_keyword1 expression1 facet2: expression2... { // a sequence statement
     statement_keyword2 expression1 facet2: expression2...;  // a single statement
     statement_keyword3 expression1 facet2: expression2...;
}
```

16.4.1 =

16.4.1.1 Facets

- **`right`** (float), (omissible) : the right part of the equation (it is mandatory that it can be evaluated as a float
- **`left`** (any type): the left part of the equation (it should be a variable or a call to the diff() or diff2() operators)

16.4.1.2 Definition

Allows to implement an equation in the form function(n, t) = expression. The left function is only here as a placeholder for enabling a simpler syntax and grabbing the variable as its left member.

16.4.1.3 Usages

- The syntax of the = statement is a bit different from the other statements. It has to be used as follows (in an equation):

```
float t; float S; float I; equation SI {    diff(S,t) = (- 0.3 * S * I / 100);   diff(I,t) = (0.3 * S * I / 100);


* See also: [equation](#equation), [solve](#solve),

#### Embedments
* The `=` statement is of type: **Single statement**
* The `=` statement can be embedded into: equation,
* The `=` statement embeds statements:

----

[//]: # (keyword|statement_action)
### action
#### Facets

  * **`name`** (an identifier), (omissible) : identifier of the action
  * `index` (a datatype identifier): if the action returns a map, the type of its keys
 * `of` (a datatype identifier): if the action returns a container, the type of its elements
  * `type` (a datatype identifier): the action returned type
 * `virtual` (boolean): whether the action is virtual (defined without a set of instructions) (false by defau

#### Definition
```

Allows to define in a species, model or experiment a new action that can be called elsewhere.

Usages

* The simplest syntax to define an action that does not take any parameter and does not return anything is:

action simple_action { // [set of statements] } “‘

- If the action needs some parameters, they can be specified betwee, braquets after the identifier of the action:

action action_parameters(int i, string s){   // [set of statements using i and s] } ```

* If the action returns any value, the returned type should be used instead of the "action" keyword. A return

int action_return_val(int i, string s){ // [set of statements using i and s] return i + i; } “‘

- If virtual: is true, then the action is abstract, which means that the action is defined without body. A species containing at least one abstract action is abstract. Agents of this species cannot be created. The common use of an abstract action is to define an action that can be used by all its sub-species, which should redefine all abstract actions and implements its body.

species parent_species {   int virtual_action(int i, string s); } species children parent: parent_species {

* See also: [do](#do),

Embedments
* The `action` statement is of type: **Sequence of statements or action**
* The `action` statement can be embedded into: Species, Experiment, Model,
* The `action` statement embeds statements: [assert](#assert), [return](#return),

----

[//]: # (keyword|statement_add)
add
Facets

 * **`to`** (any type in [container, species, agent, geometry]): an expression that evaluates to a container
  * `item` (any type), (omissible) : any expression to add in the container
 * `all` (any type): Allows to either pass a container so as to add all its element, or 'true', if the item to
  * `at` (any type): position in the container of added element
 * `edge` (any type): a pair that will be added to a graph as an edge (if nodes do not exist, they are also add
  * `node` (any type): an expression that will be added to a graph as a node.
  * `vertex` (any type):
 * `weight` (float): An optional float value representing the weight to attach to this element in case the co

Definition

Allows to add, i.e. to insert, a new element in a container (a list, matrix, map, ...).Incorrect use: The addi

Usages

* The new element can be added either at the end of the container or at a particular position.

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

- Case of a list, the expression in the facet at: should be an integer.

```
list<int> workingList <- []; add 0 at: 0 to: workingList ;//workingList equals [0]add 10 at: 0 to: workingLis
```

```
* Case of a map: As a map is basically a list of pairs key::value, we can also use the add statement on it. It i
```

map workingMap <- []; add "val1" at: "x" to: workingMap;//workingMap equals ["x"::"val1"]"'

- If the at facet is omitted, a pair expr_item::expr_item will be added to the map. An important exception is the case where the expr_item is a pair: in this case the pair is added.

```
add "val2" to: workingMap;//workingMap equals ["x"::"val1", "val2"::"val2"]add "5"::"val4" to: workingMap;
```

```
* Notice that, as the key should be unique, the addition of an item at an existing position (i.e. existing key
```

add "val3" at: "x" to: workingMap;//workingMap equals ["x"::"val3", "val2"::"val2", "5"::"val4"]"'

- On a map, the all facet will add all value of a container in the map (so as pair val_cont::val_cont)

```
add ["val4","val5"] all: true at: "x" to: workingMap;//workingMap equals ["x"::"val3", "val2"::"val2", "5":
```

```
* In case of a graph, we can use the facets `node`, `edge` and `weight` to add a node, an edge or weights to the
```

graph g <- as_edge_graph([{1,5}::{12,45}]); add edge: {1,5}::{2,3} to: g;
list var <- g.vertices; // var equals [{1,5},{12,45},{2,3}] list var <- g.edges; // var equals [polyline({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]add node: {5,5} to: g;
list var <- g.vertices; // var equals [{1.0,5.0},{12.0,45.0},{2.0,3.0},{5.0,5.0}] list var <- g.edges; // var equals [polyline({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]"'

- Case of a matrix: this statement can not be used on matrix. Please refer to the statement put.

- See also: put, remove,

16.4.1.4 Embedments

- The `add` statement is of type: **Single statement**
- The `add` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `add` statement embeds statements:

16.4.2 agents

16.4.2.1 Facets

- **`value`** (container): the set of agents to display
- **`name`** (a label), (omissible) : Human readable title of the layer
- **`aspect`** (an identifier): the name of the aspect that should be used to display the species
- **`fading`** (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false
- **`focus`** (agent): the agent on which the camera will be focused (it is dynamically computed)

- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, useful in case of agents that do not move)
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

16.4.2.2 Definition

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

16.4.2.3 Usages

- The general syntax is:

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


* For instance, in a segregation model, `agents` will only display unhappy agents:
```

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

- See also: display, chart, event, graphics, display_grid, image, overlay, display_population,

16.4.2.4 Embedments

- The `agents` statement is of type: **Layer**
- The `agents` statement can be embedded into: display,
- The `agents` statement embeds statements:

16.4.3 annealing

16.4.3.1 Facets

- **`name`** (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `nb_iter_cst_temp` (int): number of iterations per level of temperature

- `temp_decrease` (float): temperature decrease coefficient
- `temp_end` (float): final temperature
- `temp_init` (float): initial temperature

16.4.3.2 Definition

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

16.4.3.3 Usages

- As other batch methods, the basic syntax of the annealing statement uses `method annealing` instead of the expected `annealing name: id` :

```
method annealing [facet: value]; ```

* For example:
```

method annealing temp_init: 100 temp_end: 1 temp_decrease: 0.5 nb_iter_cst_temp: 5 maximize: food_gathered; “‘

16.4.3.4 Embedments

- The `annealing` statement is of type: **Batch method**
- The `annealing` statement can be embedded into: Experiment,
- The `annealing` statement embeds statements:

16.4.4 ask

16.4.4.1 Facets

- **`target`** (any type in [container, agent]), (omissible) : an expression that evaluates to an agent or a list of agents
- `as` (species): an expression that evaluates to a species
- `parallel` (any type in [boolean, int]): (experimental) setting this facet to ‘true’ will allow ‘ask’ to use concurrency when traversing the targets; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is false by default.

16.4.4.2 Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. If the value of the target facet is nil or empty, the statement is ignored.

16.4.4.3 Usages

- Ask a set of receiver agents, stored in a container, to perform a block of statements. The block is evaluated in the context of the agents' species

```
ask ${receiver_agents} {      ${cursor} } ```


* Ask  one agent to perform a block of statements. The block is evaluated in the context of the agent's specie
```

ask ${one_agent} { ${cursor} } “‘

- If the species of the receiver agent(s) cannot be determined, it is possible to force it using the `as` facet. An error is thrown if an agent is not a direct or undirect instance of this species

```
ask${receiver_agent(s)} as: ${a_species_expression} {      ${cursor} } ```


* To ask a set of agents to do something only if they belong to a given species, the `of_species` operator can
```

ask ${receiver_agents} of_species ${species_name} { ${cursor} } “‘

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

```
species animal {    float energy <- rnd (1000) min: 0.0 {    reflex when: energy > 500 { // executed when the e


* If the species of the receiver agent cannot be determined, it is possible to force it by casting the agent. I

#### Embedments
* The `ask` statement is of type: **Sequence of statements or action**
* The `ask` statement can be embedded into: chart, Behavior, Sequence of statements or action,
* The `ask` statement embeds statements:

----

[//]: # (keyword|statement_aspect)
### aspect
#### Facets

  * `name` (an identifier), (omissible) : identifier of the aspect (it can be used in a display to identify whi

#### Definition

Aspect statement is used to define a way to draw the current agent. Several aspects can be defined in one spec

#### Usages

* An example of use of the aspect statement:
```

species one_species { int a <- rnd(10); aspect aspect1 { if(a mod 2 = 0) { draw circle(a);} else {draw square(a);} draw text: “a=” + a color: #black size: 5; } } “‘

16.4.4.4 Embedments

- The `aspect` statement is of type: **Behavior**
- The `aspect` statement can be embedded into: Species, Model,
- The `aspect` statement embeds statements: draw,

16.4.5 assert

16.4.5.1 Facets

- **`value`** (boolean), (omissible) : a boolean expression. If its evaluation is true, the assertion is successful. Otherwise, an error (or a warning) is raised.
- **`warning`** (boolean): if set to true, makes the assertion emit a warning instead of an error

16.4.5.2 Definition

Allows to check if the evaluation of a given expression returns true. If not, an error (or a warning) is raised. If the statement is used inside a test, the error is not propagagated but invalidates the test (in case of a warning, it partially invalidates it). Otherwise, it is normally propagated

16.4.5.3 Usages

- Any boolean expression can be used

```
assert (2+2) = 4; assert self != nil; int t <- 0; assert is_error(3/t); (1 / 2) is float ```


* if the 'warn:' facet is set to true, the statement emits a warning (instead of an error) in case the express
```

assert ‘abc’ is string warning: true “‘

- See also: test, setup, is_error, is_warning,

16.4.5.4 Embedments

- The `assert` statement is of type: **Single statement**
- The `assert` statement can be embedded into: test, action, Sequence of statements or action, Behavior, Sequence of statements or action,
- The `assert` statement embeds statements:

16.4.6 benchmark

16.4.6.1 Facets

- `message` (any type), (omissible) : A message to display alongside the results. Should concisely describe the contents of the benchmark
- `repeat` (int): An int expression describing how many executions of the block must be handled. The output in this case will return the min, max and average durations

16.4.6.2 Definition

Displays in the console the duration in ms of the execution of the statements included in the block. It is possible to indicate, with the 'repeat' facet, how many times the sequence should be run

16.4.6.3 Usages

16.4.6.4 Embedments

- The `benchmark` statement is of type: **Sequence of statements or action**
- The `benchmark` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `benchmark` statement embeds statements:

16.4.7 break

16.4.7.1 Facets

16.4.7.2 Definition

`break` allows to interrupt the current sequence of statements.

16.4.7.3 Usages

16.4.7.4 Embedments

- The `break` statement is of type: **Single statement**
- The `break` statement can be embedded into: Sequence of statements or action,
- The `break` statement embeds statements:

16.4.8 camera

16.4.8.1 Facets

- **`name`** (string), (omissible) : The name of the camera
- `location` (point): The location of the camera in the world
- `look_at` (point): The location that the camera is looking
- `up_vector` (point): The up-vector of the camera.

16.4.8.2 Definition

`camera` allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way.

16.4.8.3 Usages

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

16.4.8.4 Embedments

- The `camera` statement is of type: **Layer**
- The `camera` statement can be embedded into: display,
- The `camera` statement embeds statements:

16.4.9 capture

16.4.9.1 Facets

- **`target`** (any type in [agent, container]), (omissible) : an expression that is evaluated as an agent or a list of the agent to be captured
- `as` (species): the species that the captured agent(s) will become, this is a micro-species of the calling agent's species
- `returns` (a new identifier): a list of the newly captured agent(s)

16.4.9.2 Definition

Allows an agent to capture other agent(s) as its micro-agent(s).

16.4.9.3 Usages

- The preliminary for an agent A to capture an agent B as its micro-agent is that the A's species must defined a micro-species which is a sub-species of B's species (cf. [Species161#Nesting_species Nesting species]).

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


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

capture list(B) as: C; "'

- Deprecated writing:

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


* See also: [release](#release),

#### Embedments
* The `capture` statement is of type: **Sequence of statements or action**
* The `capture` statement can be embedded into: Behavior, Sequence of statements or action,
* The `capture` statement embeds statements:

----

[//]: # (keyword|statement_catch)
### catch
#### Facets
```

Definition

This statement cannot be used alone

Usages

* See also: [try](#try),

Embedments
* The `catch` statement is of type: **Sequence of statements or action**
* The `catch` statement can be embedded into: try,
* The `catch` statement embeds statements:

[//]: # (keyword|statement_chart)
chart
Facets

  * **`name`** (string), (omissible) : the identifier of the chart layer
  * `axes` (rgb): the axis color
  * `background` (rgb): the background color
  * `color` (rgb): Text color
  * `gap` (float): minimum gap between bars (in proportion)
  * `label_font` (string): Label font face
  * `label_font_size` (int): Label font size
  * `label_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display lab
  * `legend_font` (string): Legend font face
  * `legend_font_size` (int): Legend font size
  * `legend_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display le
  * `memorize` (boolean): Whether or not to keep the values in memory (in order to produce a csv file, for inst
  * `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[,
  * `reverse_axes` (boolean): reverse X and Y axis (for example to get horizental bar charts
  * `series_label_position` (an identifier), takes values in: {default, none, legend, onchart, yaxis, xaxis}
  * `size` (point): the layer resize factor: {1,1} refers to the original size whereas {0.5,0.5} divides by 2
  * `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, e
  * `tick_font` (string): Tick font face
  * `tick_font_size` (int): Tick font size
  * `tick_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display tick
  * `tick_line_color` (rgb): the tick lines color
  * `title_font` (string): Title font face
  * `title_font_size` (int): Title font size
  * `title_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display tit
  * `title_visible` (boolean): chart title visible
  * `type` (an identifier), takes values in: {xy, scatter, histogram, series, pie, radar, heatmap, box_whiske
  * `x_label` (string): the title for the X axis
  * `x_log_scale` (boolean): use Log Scale for X axis
  * `x_range` (any type in [float, int, point, list]): range of the x-axis. Can be a number (which will set the
  * `x_serie` (any type in [list, float, int]): for series charts, change the default common x serie (simulati
  * `x_serie_labels` (any type in [list, float, int, a label]): change the default common x series labels (rep
  * `x_tick_line_visible` (boolean): X tick line visible
  * `x_tick_unit` (float): the tick unit for the y-axis (distance between horyzontal lines and values on the l
  * `x_tick_values_visible` (boolean): X tick values visible
  * `y_label` (string): the title for the Y axis

```
  * `y_log_scale` (boolean): use Log Scale for Y axis
 * `y_range` (any type in [float, int, point, list]): range of the y-axis. Can be a number (which will set the
 * `y_serie_labels` (any type in [list, float, int, a label]): for heatmaps/3d charts, change the default y s
  * `y_tick_line_visible` (boolean): Y tick line visible
 * `y_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the
 * `y_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the
  * `y_tick_values_visible` (boolean): Y tick values visible
  * `y2_label` (string): the title for the second Y axis
  * `y2_log_scale` (boolean): use Log Scale for second Y axis
 * `y2_range` (any type in [float, int, point, list]): range of the second y-axis. Can be a number (which wil

#### Definition

`chart` allows modeler to display a chart: this enables to display specific values of the model at each itera

#### Usages

* The general syntax is:
```

display chart_display { chart "chart name" type: series [additional options] { [Set of data, datalists statements] } } "'

- See also: display, agents, event, graphics, display_grid, image, overlay, quadtree, display_population, text,

16.4.9.4 Embedments

- The `chart` statement is of type: **Layer**
- The `chart` statement can be embedded into: display,
- The `chart` statement embeds statements: add, ask, data, datalist, do, put, remove, set, simulate, using,

---

16.4.10 conscious_contagion

16.4.10.1 Facets

- `emotion_created` (546706): the emotion that will be created with the contagion

- `emotion_detected` (546706): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `intensity` (float): The intensity value of the emotion added to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

16.4.10.2 Definition

enables to directly add an emotion of a perceived specie if the perceived agent ges a patricular emotion.

16.4.10.3 Usages

- Other examples of use: conscious_contagion emotion_detected:fear emotion_created:fearConfirmed; conscious_contagion emotion_detected:fear emotion_created:fearConfirmed charisma: 0.5 receptivity: 0.5;

16.4.10.4 Embedments

- The conscious_contagion statement is of type: **Single statement**
- The conscious_contagion statement can be embedded into: Behavior, Sequence of statements or action,
- The conscious_contagion statement embeds statements:

16.4.11 create

16.4.11.1 Facets

- species (any type in [species, agent]), (omissible) : an expression that evaluates to a species, the species of the agents to be created. In the case of simulations, the name 'simulation', which represents the current instance of simulation, can also be used as a proxy to their species
- as (species):
- from (any type): an expression that evaluates to a localized entity, a list of localized entities, a string (the path of a file), a file (shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a database
- header (boolean): an expression that evaluates to a boolean, when creating agents from csv file, specify whether the file header is loaded
- number (int): an expression that evaluates to an int, the number of created agents
- returns (a new identifier): a new temporary variable name containing the list of created agents (a list, even if only one agent has been created)
- with (map): an expression that evaluates to a map, for each pair the key is a species attribute and the value the assigned value

16.4.11.2 Definition

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

16.4.11.3 Usages

- Its simple syntax to create an_int agents of species a_species is:

create a_species number: an_int; create species_of(self) number: 5 returns: list5Agents; 5 ```

* In GAML modelers can create agents of species `a_species (with two attributes `type` and `nature` with typ

create a_species from: the_shapefile with: [type:: read('TYPE_OCC'), nature::read('NATURE')]; "'

- In order to create agents from a .csv file, facet header can be used to specified whether we can use columns header:

`create toto from: "toto.csv" header: true with:[att1::read("NAME"), att2::read("TYPE")]; or create toto fro`

`* Similarly to the creation from shapefile, modelers can create agents from a set of geometries. In this case`

create species_of(self) from: [square(4),circle(4)]; // 2 agents have been created, with shapes respectively square(4) and circle(4) "'

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

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

`* If one wants to specify a special initialization sequence for the agents created, create provides the same`

create a_species number: an_int { statements } "'

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

`create species(self) number: rnd (4) returns: children {    set location <- myself.location + {rnd (2), rnd`

`* Desprecated uses:`

// Simple syntax create species: a_species number: an_int; "'

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

16.4.11.4 Embedments

- The `create` statement is of type: **Sequence of statements or action**
- The `create` statement can be embedded into: Behavior, Sequence of statements or action,
- The `create` statement embeds statements:

16.4.12 data

16.4.12.1 Facets

- **`legend`** (string), (omissible) : The legend of the chart

- **`value`** (any type in [float, point, list]): The value to output on the chart
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (any type in [rgb, list]): color of the serie, for heatmap can be a list to specify [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false)
- `line_visible` (boolean): Whether lines are visible or not
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker

- `marker_size` (float): Size in pixels of the marker
- `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified on chart statement)
- `thickness` (float): The thickness of the lines to draw
- `use_second_y_axis` (boolean): Use second y axis for this serie
- `x_err_values` (any type in [float, list]): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_err_values` (any type in [float, list]): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

16.4.12.2 Definition

This statement allows to describe the values that will be displayed on the chart.

16.4.12.3 Usages

16.4.12.4 Embedments

- The `data` statement is of type: **Single statement**
- The `data` statement can be embedded into: chart, Sequence of statements or action,
- The `data` statement embeds statements:

16.4.13 datalist

16.4.13.1 Facets

- `value` (list): the values to display. Has to be a matrix, a list or a List of List. Each element can be a number (series/histogram) or a list with two values (XY chart)
- `legend` (list), (omissible) : the name of the series: a list of strings (can be a variable with dynamic names)
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (list): list of colors, for heatmaps can be a list of [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false), same for all series.
- `line_visible` (boolean): Line visible or not (same for all series)
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker. Same one for all series.
- `marker_size` (list): the marker sizes to display. Can be a list of numbers (same size for each marker of the series) or a list of list (different sizes by point)
- `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified on chart statement)
- `thickness` (float): The thickness of the lines to draw
- `use_second_y_axis` (boolean): Use second y axis for this serie
- `x_err_values` (list): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)

- `y_err_values` (list): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

16.4.13.2 Definition

add a list of series to a chart. The number of series can be dynamic (the size of the list changes each step). See Ant Foraging (Charts) model in ChartTest for examples.

16.4.13.3 Usages

16.4.13.4 Embedments

- The `datalist` statement is of type: **Single statement**
- The `datalist` statement can be embedded into: chart, Sequence of statements or action,
- The `datalist` statement embeds statements:

16.4.14 default

16.4.14.1 Facets

- `value` (any type), (omissible) : The value or values this statement tries to match

16.4.14.2 Definition

Used in a switch match structure, the block prefixed by default is executed only if no other block has matched (otherwise it is not).

16.4.14.3 Usages

- See also: switch, match,

16.4.14.4 Embedments

- The `default` statement is of type: **Sequence of statements or action**
- The `default` statement can be embedded into: switch,
- The `default` statement embeds statements:

16.4.15 diffuse

16.4.15.1 Facets

- **`var`** (an identifier), (omissible) : the variable to be diffused

- **on** (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the proportion value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix masking the diffusion (matrix created from a image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- `mat_diffu` (matrix): the diffusion matrix (can have any size)
- `matrix` (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- `min_value` (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- `propagation` (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : S' = (S / N / proportion) - variation. The intensity of S is then diminished by S `*` proportion on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals 'gradient', the original intensity is not modified, and each neighbors receives the intensity : S / proportion - variation. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to 'diffusion'.
- `proportion` (float): a diffusion rate
- `radius` (int): a diffusion radius (in number of cells from the center)
- `variation` (float): an absolute value to decrease at each neighbors

16.4.15.2 Definition

This statements allows a value to diffuse among a species on agents (generally on a grid) depending on a given diffusion matrix.

16.4.15.3 Usages

- A basic example of diffusion of the variable phero defined in the species cells, given a diffusion matrix math_diff is:

```
matrix<float> math_diff <- matrix([[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]); diffuse var: phero on: cel
```

`* The diffusion can be masked by obstacles, created from a bitmap image:`

diffuse var: phero on: cells mat_diffu: math_diff mask: mymask; "'

- A convenient way to have an uniform diffusion in a given radius is (which is equivalent to the above diffusion):

```
diffuse var: phero on: cells proportion: 1/9 radius: 1; ```
```

```
#### Embedments
* The `diffuse` statement is of type: **Single statement**
* The `diffuse` statement can be embedded into: Behavior, Sequence of statements or action,
* The `diffuse` statement embeds statements:

----

[//]: # (keyword|statement_display)
### display
#### Facets

  * **`name`** (a label), (omissible) : the identifier of the display
 * `ambient_light` (any type in [int, rgb]): Allows to define the value of the ambient light either using an i
 * `autosave` (any type in [boolean, point]): Allows to save this display on disk. A value of true/false will
  * `background` (rgb): Allows to fill the background of the display with a specific color
 * `camera_interaction` (boolean): If false, the user will not be able to modify the position and the orienta
  * `camera_lens` (int): Allows to define the lens of the camera
  * `camera_look_pos` (point): Allows to define the direction of the camera
  * `camera_pos` (any type in [point, agent]): Allows to define the position of the camera
  * `camera_up_vector` (point): Allows to define the orientation of the camera
 * `diffuse_light` (any type in [int, rgb]): Allows to define the value of the diffuse light either using an i
 * `diffuse_light_pos` (point): Allows to define the position of the diffuse light either using an point (di
 * `draw_diffuse_light` (boolean): Allows to show/hide a representation of the lights. Default is false.
 * `draw_env` (boolean): Allows to enable/disable the drawing of the world shape and the ordinate axes. Defau
 * `focus` (geometry): the geometry (or agent) on which the display will (dynamically) focus
 * `fullscreen` (any type in [boolean, int]): Indicates, when using a boolean value, whether or not the displ
 * `keystone` (container): Set the position of the 4 corners of your screen ([topLeft,topRight,botLeft,botRi
  * `light` (boolean): Allows to enable/disable the light. Default is true
 * `orthographic_projection` (boolean): Allows to enable/disable the orthographic projection. Default can b
 * `parent` (an identifier): Declares that this display inherits its layers and attributes from the parent di
 * `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true
  * `refresh_every` (int): Allows to refresh the display every n time steps (default is 1)
  * `rotate` (float): Set the angle for the rotation around the Z axis
 * `scale` (any type in [boolean, float]): Allows to display a scale bar in the overlay. Accepts true/false o
 * `show_fps` (boolean): Allows to enable/disable the drawing of the number of frames per second
 * `synchronized` (boolean): Indicates whether the display should be directly synchronized with the simulati
 * `toolbar` (boolean): Indicates whether the top toolbar of the display view should be initially visible or
 * `type` (a label): Allows to use either Java2D (for planar models) or OpenGL (for 3D models) as the renderin
  * `use_shader` (boolean): Under construction...
 * `virtual` (boolean): Declaring a display as virtual makes it invisible on screen, and only usable for disp
 * `z_fighting` (boolean): Allows to alleviate a problem where agents at the same z would overlap each other

#### Definition

A display refers to a independent and mobile part of the interface that can display species, images, texts or

#### Usages

* The general syntax is:
```

display my_display [additional options] { ... } “‘

- Each display can include different layers (like in a GIS).

`display gridWithElevationTriangulated type: opengl ambient_light: 100 {    grid cell elevation: true triang`

Embedments
* The `display` statement is of type: **Output**
* The `display` statement can be embedded into: output, permanent,
* The `display` statement embeds statements: [agents](#agents), [camera](#camera), [chart](#chart), [dis

----

[//]: # (keyword|statement_display_grid)
display_grid
Facets

  * **`species`** (species), (omissible) : the species of the agents in the grid
  * `dem` (matrix):
  * `draw_as_dem` (boolean):
 * `elevation` (any type in [matrix, float, int, boolean]): Allows to specify the elevation of each cell, if
 * `grayscale` (boolean): if true, givse a grey value to each polygon depending on its elevation (false by de
  * `lines` (rgb): the color to draw lines (borders of cells)
 * `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[,
 * `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by defa
 * `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. De
 * `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as p
 * `text` (boolean): specify whether the attribute used to compute the elevation is displayed on each cells
 * `texture` (any type in [boolean, file]): Either file containing the texture image to be applied on the gr
 * `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)
 * `triangulation` (boolean): specifies whther the cells will be triangulated: if it is false, they will be d

Definition

`display_grid` is used using the `grid` keyword. It allows the modeler to display in an optimized way all cel

Usages

* The general syntax is:

display my_display { grid ant_grid lines: #black position: { 0.5, 0 } size: {0.5,0.5}; } “‘

- To display a grid as a DEM:

display my_display {   grid cell texture: texture_file text: false triangulation: true elevation: true; } `

* See also: [display](#display), [agents](#agents), [chart](#chart), [event](#event), [graphics](#graphics

Embedments
* The `display_grid` statement is of type: **Layer**
* The `display_grid` statement can be embedded into: display,
* The `display_grid` statement embeds statements:

[//]: # (keyword|statement_display_population)
display_population

```
#### Facets

  * **`species`** (species), (omissible) : the species to be displayed
 * `aspect` (an identifier): the name of the aspect that should be used to display the species
 * `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous tra
 * `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[,
 * `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by defau
 * `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Def
 * `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[ are treated as p
 * `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on
 * `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

#### Definition

The `display_population` statement is used using the `species keyword`. It allows modeler to display all the

#### Usages

* The general syntax is:
```

display my_display { species species_name [additional options]; } “‘

- Species can be superposed on the same plan (be careful with the order, the last one will be above all the others):

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


* Each species layer can be placed at a different z value using the opengl display. position:{0,0,0} means the
```

display my_display type: opengl{ species agent1 aspect: base ; species agent2 aspect: base position:{0,0,0.5}; species agent3 aspect: base position:{0,0,1}; } “‘

- See also: display, agents, chart, event, graphics, display_grid, image, overlay,

16.4.15.4 Embedments

- The `display_population` statement is of type: **Layer**
- The `display_population` statement can be embedded into: display, display_population,
- The `display_population` statement embeds statements: display_population,

16.4.16 do

16.4.16.1 Facets

- **`action`** (an identifier), (omissible) : the name of an action or a primitive
- **`internal_function`** (any type):
- **`returns`** (a new identifier): create a new variable and assign to it the result of the action
- **`with`** (map): a map expression containing the parameters of the action

16.4.16.2 Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this [BuiltIn161 page]; for the list of primitives defined by the different skills, see this [Skills161 page]. Finally, see this [Species161 page] to know how to declare custom actions.

16.4.16.3 Usages

- The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

`do name_of_action_or_primitive; ````

`* In case the action expects one or more arguments to be passed, they are defined by using facets (enclosed ta`

do name_of_action_or_primitive arg1: expression1 arg2: expression2; "'

- In case the result of the action needs to be made available to the agent, the action can be called with the agent calling the action (`self` when the agent itself calls the action) instead of `do`; the result should be assigned to a temporary variable:

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

`* In case of an action expecting arguments and returning a value, the following syntax is used:`

type_returned_by_action result <- self name_of_action_or_primitive [arg1::expression1, arg2::expression2]; "'

- Deprecated uses: following uses of the `do` statement (still accepted) are now deprecated:

```
// Simple syntax:  do action: name_of_action_or_primitive;  // In case the result of the action needs to be ma



#### Embedments
* The `do` statement is of type: **Single statement**
* The `do` statement can be embedded into: chart, Behavior, Sequence of statements or action,
* The `do` statement embeds statements:

----

[//]: # (keyword|statement_draw)
### draw
#### Facets

  * `geometry` (any type), (omissible) : any type of data (it can be geometry, image, text)
 * `anchor` (point): the anchor point of the location with respect to the envelope of the text to draw, can ta
  * `at` (point): location where the shape/text/icon is drawn
 * `begin_arrow` (any type in [int, float]): the size of the arrow, located at the beginning of the drawn geor
  * `bitmap` (boolean): Whether to render the text in 3D or not
 * `border` (any type in [rgb, boolean]): if used with a color, represents the color of the geometry border. I
 * `color` (any type in [rgb, container]): the color to use to display the object. In case of images, will try
 * `depth` (float): (only if the display type is opengl) Add an artificial depth to the geometry previously d
  * `empty` (boolean): a condition specifying whether the geometry is empty or full
 * `end_arrow` (any type in [int, float]): the size of the arrow, located at the end of the drawn geometry
 * `font` (any type in [19, string]): the font used to draw the text, if any. Applying this facet to geometrie
```

```
 * `material` (25): Set a particular material to the object (only if you are in the "use_shader" mode).
 * `perspective` (boolean): Whether to render the text in perspective or facing the user. Default is true.
 * `rotate` (any type in [float, int, pair]): orientation of the shape/text/icon; can be either an int/float
  * `rounded` (boolean): specify whether the geometry have to be rounded (e.g. for squares)
 * `size` (any type in [float, point]): size of the object to draw, expressed as a bounding box (width, heigh
 * `texture` (any type in [string, list, file]): the texture(s) that should be applied to the geometry. Eithe
  * `width` (float): The line width to use for drawing this object

#### Definition

`draw` is used in an aspect block to express how agents of the species will be drawn. It is evaluated each tim

#### Usages

* Any kind of geometry as any location can be drawn when displaying an agent (independently of his shape)
```

aspect geometryAspect { draw circle(1.0) empty: !hasFood color: #orange ; } “‘

- Image or text can also be drawn

```
aspect arrowAspect {   draw "Current state= "+state at: location + {-3,1.5} color: #white font: font('Defaul


* Arrows can be drawn with any kind of geometry, using begin_arrow and end_arrow facets, combined with the em
```

aspect arrowAspect { draw line([{20, 20}, {40, 40}]) color: #black begin_arrow:5; draw line([{10, 10},{20, 50}, {40, 70}]) color: #green end_arrow: 2 begin_arrow: 2 empty: true; draw square(10) at: {80,20} color: #purple begin_arrow: 2 empty: true; } “‘

16.4.16.4 Embedments

- The `draw` statement is of type: **Single statement**
- The `draw` statement can be embedded into: aspect, Sequence of statements or action, Layer,
- The `draw` statement embeds statements:

16.4.17 else

16.4.17.1 Facets

16.4.17.2 Definition

This statement cannot be used alone

16.4.17.3 Usages

- See also: if,

16.4.17.4 Embedments

- The `else` statement is of type: **Sequence of statements or action**
- The `else` statement can be embedded into: if,

- The `else` statement embeds statements:

16.4.18 emotional_contagion

16.4.18.1 Facets

- `emotion_detected` (546706): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the emotional contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `emotion_created` (546706): the emotion that will be created with the contagion
- `intensity` (float): The intensity value of the emotion created to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

16.4.18.2 Definition

enables to make conscious or unconscious emotional contagion

16.4.18.3 Usages

- Other examples of use: `emotional_contagion emotion_detected:fearConfirmed; emotional_contagion emotion_detected:fear emotion_created:fearConfirmed; emotional_contagion emotion_detected:fear emotion_created:fearConfirmed charisma: 0.5 receptivity: 0.5;`

16.4.18.4 Embedments

- The `emotional_contagion` statement is of type: **Single statement**
- The `emotional_contagion` statement can be embedded into: Behavior, Sequence of statements or action,
- The `emotional_contagion` statement embeds statements:

16.4.19 enforcement

16.4.19.1 Facets

- `name` (an identifier), (omissible) : the identifier of the enforcement
- `law` (string): The law to enforce
- `norm` (string): The norm to enforce
- `obligation` (546704): The obligation to enforce
- `reward` (string): The positive sanction to apply if the norm has been followed
- `sanction` (string): The sanction to apply if the norm is violated
- `when` (boolean): A boolean value to enforce only with a certain condition

16.4.19.2 Definition

applay a sanction if the norm specified is violated, or a reward if the norm is applied by the perceived agent

16.4.19.3 Usages

- Other examples of use: `focus var:speed /*where speed is a variable from a species that is being perceived*/`

16.4.19.4 Embedments

- The `enforcement` statement is of type: **Single statement**
- The `enforcement` statement can be embedded into: Behavior, Sequence of statements or action,
- The `enforcement` statement embeds statements:

16.4.20 enter

16.4.20.1 Facets

16.4.20.2 Definition

In an FSM architecture, `enter` introduces a sequence of statements to execute upon entering a state.

16.4.20.3 Usages

- In the following example, at the step it enters into the state s_init, the message 'Enter in s_init' is displayed followed by the display of the state name:

```
state s_init {     enter { write "Enter in" + state; }          write "Enter in" + state;     }     write sta
```

```
* See also: [state](#state), [exit](#exit), [transition](#transition),

#### Embedments
* The `enter` statement is of type: **Sequence of statements or action**
* The `enter` statement can be embedded into: state,
* The `enter` statement embeds statements:

----

[//]: # (keyword|statement_equation)
### equation
#### Facets

  * **`name`** (an identifier), (omissible) : the equation identifier
  * `params` (list): the list of parameters used in predefined equation systems
 * `simultaneously` (list): a list of species containing a system of equations (all systems will be solved si
 * `type` (an identifier), takes values in: {SI, SIS, SIR, SIRS, SEIR, LV}: the choice of one among classical
  * `vars` (list): the list of variables used in predefined equation systems
```

Definition

The equation statement is used to create an equation system from several single equations.

Usages

* The basic syntax to define an equation system is:

float t; float S; float I; equation SI { diff(S,t) = (- 0.3 * S * I / 100); diff(I,t) = (0.3 * S * I / 100); } "'

- If the type: facet is used, a predefined equation system is defined using variables vars: and parameters params: in the right order. All possible predefined equation systems are the following ones (see [EquationPresentation161 EquationPresentation161] for precise definition of each classical equation system):

```
equation eqSI type: SI vars: [S,I,t] params: [N,beta]; equation eqSIS type: SIS vars: [S,I,t] params: [N,beta
```

```
* If the simultaneously: facet is used, system of all the agents will be solved simultaneously.
* See also: [=](#=), [solve](#solve),

#### Embedments
* The `equation` statement is of type: **Sequence of statements or action**
* The `equation` statement can be embedded into: Species, Model,
* The `equation` statement embeds statements: [=](#=),

----

[//]: # (keyword|statement_error)
### error
#### Facets

  * **`message`** (string), (omissible) : the message to display in the error.

#### Definition

The statement makes the agent output an error dialog (if the simulation contains a user interface). Otherwise

#### Usages

* Throwing an error
```

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

16.4.20.4 Embedments

- The `error` statement is of type: **Single statement**
- The `error` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `error` statement embeds statements:

16.4.21 event

16.4.21.1 Facets

- `name` (an identifier), (omissible) : the type of event captured: can be "mouse_up", "mouse_down", "mouse_move", "mouse_exit", "mouse_enter" or a character

- `action` (26): Either a block of statements to execute in the context of the simulation or the identifier of the action to be executed. This action needs to be defined in 'global' or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the pseudo-constant #user_location
- `type` (string): Type of peripheric used to generate events. Defaults to 'default', which encompasses keyboard and mouse
- `unused` (an identifier), takes values in: {mouse_up, mouse_down, mouse_move, mouse_enter, mouse_exit}: an unused facet that serves only for the purpose of declaring the string values

16.4.21.2 Definition

`event` allows to interact with the simulation by capturing mouse or key events and doing an action. This action needs to be defined in 'global' or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the pseudo-constant #user_location

16.4.21.3 Usages

- The general syntax is:

```
event [event_type] action: myAction; ```


* For instance:
```

global { // … action myAction () { point loc <- #user_location; // contains the location of the mouse in the world list selected_agents <- agents inside (10#m around loc); // contains agents clicked by the event // code written by modelers } } experiment Simple type:gui { display my_display { event mouse_up action: myAction; } } "'

- See also: display, agents, chart, graphics, display_grid, image, overlay, display_population,

16.4.21.4 Embedments

- The `event` statement is of type: **Layer**
- The `event` statement can be embedded into: display,
- The `event` statement embeds statements:

16.4.22 exhaustive

16.4.22.1 Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: The aggregation method to use (either min or max)

* `maximize` (float): the value the algorithm tries to maximize
* `minimize` (float): the value the algorithm tries to minimize

16.4.22.2 Definition

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way. See [batch161 the batch dedicated page].

16.4.22.3 Usages

* As other batch methods, the basic syntax of the exhaustive statement uses `method exhaustive` instead of the expected `exhaustive name: id` :

```
method exhaustive [facet: value]; ```


* For example:
```

method exhaustive maximize: food_gathered; “‘

16.4.22.4 Embedments

* The `exhaustive` statement is of type: **Batch method**
* The `exhaustive` statement can be embedded into: Experiment,
* The `exhaustive` statement embeds statements:

16.4.23 exit

16.4.23.1 Facets

16.4.23.2 Definition

In an FSM architecture, `exit` introduces a sequence of statements to execute right before exiting the state.

16.4.23.3 Usages

* In the following example, at the state it leaves the state s_init, he will display the message ‘EXIT from s_init’:

```
state s_init initial: true {      write state;      transition to: s1 when: (cycle > 2) {        write "tra
```

```
* See also: [enter](#enter), [state](#state), [transition](#transition),

#### Embedments
* The `exit` statement is of type: **Sequence of statements or action**
* The `exit` statement can be embedded into: state,
* The `exit` statement embeds statements:
```

[//]: # (keyword|statement_experiment)
experiment
Facets

 * **`name`** (a label), (omissible) : identifier of the experiment
 * **`title`** (a label):
 * **`type`** (a label), takes values in: {batch, memorize, gui, test, headless}: the type of the experiment
 * `autorun` (boolean): whether this experiment should be run automatically when launched (false by default)
 * `control` (an identifier):
 * `frequency` (int): the execution frequence of the experiment (default value: 1). If frequency: 10, the exp
 * `keep_seed` (boolean):
 * `keep_simulations` (boolean): In the case of a batch experiment, specifies whether or not the simulations
 * `parallel` (any type in [boolean, int]): When set to true, use multiple threads to run its simulations. Se
 * `parent` (an identifier): the parent experiment (in case of inheritance between experiments)
 * `repeat` (int): In the case of a batch experiment, expresses hom many times the simulations must be repeat
 * `schedules` (container): A container of agents (a species, a dynamic list, or a combination of species and
 * `skills` (list):
 * `until` (boolean): In the case of a batch experiment, an expression that will be evaluated to know when a s
 * `virtual` (boolean): whether the experiment is virtual (cannot be instantiated, but only used as a parent

Definition

Declaration of a particular type of agent that can manage simulations

Usages

Embedments
* The `experiment` statement is of type: **Experiment**
* The `experiment` statement can be embedded into: Model,
* The `experiment` statement embeds statements:

[//]: # (keyword|statement_focus)
focus
Facets

 * `agent_cause` (agent): the agentCause value of the created belief (can be nil
 * `belief` (546704): The predicate to focus on the beliefs of the other agent
 * `desire` (546704): The predicate to focus on the desires of the other agent
 * `emotion` (546706): The emotion to focus on the emotions of the other agent
 * `expression` (any type): an expression that will be the value kept in the belief
 * `id` (string): the identifier of the focus
 * `ideal` (546704): The predicate to focus on the ideals of the other agent
 * `is_uncertain` (boolean): a boolean to indicate if the mental state created is an uncertainty
 * `lifetime` (int): the lifetime value of the created belief
 * `strength` (any type in [float, int]): The priority of the created predicate
 * `truth` (boolean): the truth value of the created belief
 * `uncertainty` (546704): The predicate to focus on the uncertainties of the other agent
 * `var` (any type in [any type, list, container]): the variable of the perceived agent you want to add to you
 * `when` (boolean): A boolean value to focus only with a certain condition

```
#### Definition

enables to directly add a belief from the variable of a perceived specie.

#### Usages

* Other examples of use:
```

focus var:speed /*where speed is a variable from a species that is being perceived*/ “‘

16.4.23.4 Embedments

- The `focus` statement is of type: **Single statement**
- The `focus` statement can be embedded into: Behavior, Sequence of statements or action,
- The `focus` statement embeds statements:

16.4.24 focus_on

16.4.24.1 Facets

- **`value`** (any type), (omissible) : The agent, list of agents, geometry to focus on

16.4.24.2 Definition

Allows to focus on the passed parameter in all available displays. Passing ‘nil’ for the parameter will make all screens return to their normal zoom

16.4.24.3 Usages

- Focuses on an agent, a geometry, a set of agents, etc…)

```
focus_on my_species(0); ```



#### Embedments
* The `focus_on` statement is of type: **Single statement**
* The `focus_on` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `focus_on` statement embeds statements:

----

[//]: # (keyword|statement_genetic)
### genetic
#### Facets

  * **`name`** (an identifier), (omissible) : The name of this method. For internal use only
  * `aggregation` (a label), takes values in: {min, max}: the agregation method
  * `crossover_prob` (float): crossover probability between two individual solutions
```

```
 * `improve_sol` (boolean): if true, use a hill climbing algorithm to improve the solutions at each generatio
  * `max_gen` (int): number of generations
  * `maximize` (float): the value the algorithm tries to maximize
  * `minimize` (float): the value the algorithm tries to minimize
  * `mutation_prob` (float): mutation probability for an individual solution
 * `nb_prelim_gen` (int): number of random populations used to build the initial population
  * `pop_dim` (int): size of the population (number of individual solutions)
 * `stochastic_sel` (boolean): if true, use a stochastic selection algorithm (roulette) rather a determistic

#### Definition

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article and [batch161 the batch

#### Usages

* As other batch methods, the basic syntax of the `genetic` statement uses `method genetic` instead of the exp
```

method genetic [facet: value]; “‘

- For example:

```
method genetic maximize: food_gathered pop_dim: 5 crossover_prob: 0.7 mutation_prob: 0.1 nb_prelim_gen: 1 ma



#### Embedments
* The `genetic` statement is of type: **Batch method**
* The `genetic` statement can be embedded into: Experiment,
* The `genetic` statement embeds statements:

----

[//]: # (keyword|statement_graphics)
### graphics
#### Facets

  * `name` (a label), (omissible) : the human readable title of the graphics
 * `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous tra
 * `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[,
 * `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by defau
 * `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[ are treated as pe
 * `trace` (any type in [boolean, int]): Allows to aggregate the visualization at each timestep on the displa
 * `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

#### Definition

`graphics` allows the modeler to freely draw shapes/geometries/texts without having to define a species. It w

#### Usages

* The general syntax is:
```

display my_display { graphics “my new layer” { draw circle(5) at: {10,10} color: #red; draw “test” at: {10,10} size: 20 color: #black; } } “‘

- See also: display, agents, chart, event, graphics, display_grid, image, overlay, display_population,

16.4.24.4 Embedments

- The `graphics` statement is of type: **Layer**
- The `graphics` statement can be embedded into: display,
- The `graphics` statement embeds statements:

16.4.25 highlight

16.4.25.1 Facets

- **`value`** (agent), (omissible) : The agent to hightlight
- `color` (rgb): An optional color to highlight the agent. Note that this color will become the default color for further higlight operations

16.4.25.2 Definition

Allows to highlight the agent passed in parameter in all available displays, optionaly setting a color. Passing 'nil' for the agent will remove the current highlight

16.4.25.3 Usages

- Highlighting an agent

~~~~
highlight my_species(0) color: #blue; ```



#### Embedments
* The `highlight` statement is of type: **Single statement**
* The `highlight` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `highlight` statement embeds statements:

----

[//]: # (keyword|statement_hill_climbing)
### hill_climbing
#### Facets

  * **`name`** (an identifier), (omissible) : The name of the method. For internal use only
  * `aggregation` (a label), takes values in: {min, max}: the agregation method
  * `iter_max` (int): number of iterations
  * `maximize` (float): the value the algorithm tries to maximize
  * `minimize` (float): the value the algorithm tries to minimize

#### Definition

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article and [batch161 t

#### Usages

* As other batch methods, the basic syntax of the `hill_climbing` statement uses `method hill_climbing` inst
~~~~

method hill_climbing [facet: value]; “‘

- For example:

```
method hill_climbing iter_max: 50 maximize : food_gathered;  ```



#### Embedments
* The `hill_climbing` statement is of type: **Batch method**
* The `hill_climbing` statement can be embedded into: Experiment,
* The `hill_climbing` statement embeds statements:

----

[//]: # (keyword|statement_if)
### if
#### Facets

  * **`condition`** (boolean), (omissible) : A boolean expression: the condition that is evaluated.

#### Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true.

#### Usages

* The generic syntax is:
```

if bool_expr { statements } “‘

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

```
if bool_expr {    [statements] } else {    [statements] } string valTrue <- ""; if true {    valTrue <- "true"


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

if bool_expr { statements } else if bool_expr2 { statements } else { statements } “‘

16.4.25.4 Embedments

- The `if` statement is of type: **Sequence of statements or action**
- The `if` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `if` statement embeds statements: else,

16.4.26 image

16.4.26.1 Facets

- `name` (any type in [string, file]), (omissible) : Human readable title of the image layer
- `color` (rgb): in the case of a shapefile, this the color used to fill in geometries of the shapefile. In the case of an image, it is used to tint the image

- `file` (any type in [string, file]): the name/path of the image (in the case of a raster image)
- `gis` (any type in [file, string]): the name/path of the shape file (to display a shapefile as background, without creating agents from it)
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the image display is refreshed or not. (false by default, true should be used in cases of images that are modified over the simulation)
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

16.4.26.2 Definition

`image` allows modeler to display an image (e.g. as background of a simulation).

16.4.26.3 Usages

- The general syntax is:

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

`* For instance, in the case of a bitmap image`

display my_display { image background file:"../images/my__backgound.jpg"; } "'

- Or in the case of a shapefile:

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

`* It is also possible to superpose images on different layers in the same way as for species using opengl disp`

display my_display { image image1 file:"../images/image1.jpg"; image image2 file:"../images/image2.jpg"; image image3 file:"../images/image3.jpg" position: {0,0,0.5}; } "'

- See also: display, agents, chart, event, graphics, display__grid, overlay, display__population,

16.4.26.4 Embedments

- The `image` statement is of type: **Layer**
- The `image` statement can be embedded into: display,
- The `image` statement embeds statements:

16.4.27 inspect

16.4.27.1 Facets

- **`name`** (any type), (omissible) : the identifier of the inspector
- `attributes` (list): the list of attributes to inspect. A list that can contain strings or pair, or a mix of them. These can be variables of the species, but also attributes present in the attributes table of the agent. The type is necessary in that case
- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)
- `refresh_every` (int): Allows to refresh the inspector every n time steps (default is 1)
- `type` (an identifier), takes values in: {agent, table}: the way to inspect agents: in a table, or a set of inspectors
- `value` (any type): the set of agents to inspect, could be a species, a list of agents or an agent

16.4.27.2 Definition

`inspect` (and `browse`) statements allows modeler to inspect a set of agents, in a table with agents and all their attributes or an agent inspector per agent, depending on the type: chosen. Modeler can choose which attributes to display. When `browse` is used, type: default value is table, whereas when`inspect` is used, type: default value is agent.

16.4.27.3 Usages

- An example of syntax is:

```
inspect "my_inspector" value: ant attributes: ["name", "location"]; ```



#### Embedments
* The `inspect` statement is of type: **Output**
* The `inspect` statement can be embedded into: output, permanent, Behavior, Sequence of statements or actio
* The `inspect` statement embeds statements:

----

[//]: # (keyword|statement_law)
### law
#### Facets

  * `name` (an identifier), (omissible) : The name of the law
  * `all` (boolean): add an obligation for each belief
  * `belief` (546704): The mandatory belief
  * `beliefs` (list): The mandatory beliefs
  * `lifetime` (int): the lifetime value of the mental state created
  * `new_obligation` (546704): The predicate that will be added as an obligation
  * `new_obligations` (list): The list of predicates that will be added as obligations
 * `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurr
  * `strength` (any type in [float, int]): The stregth of the mental state created
  * `threshold` (float): Threshold linked to the obedience value.
  * `when` (boolean):
```

```
#### Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belie

#### Usages

* Other examples of use:
```

rule belief: new_predicate(“test”) when: flip(0.5) new_desire: new_predicate(“test”) “‘

16.4.27.4 Embedments

- The `law` statement is of type: **Single statement**
- The `law` statement can be embedded into: Species, Model,
- The `law` statement embeds statements:

16.4.28 layout

16.4.28.1 Facets

- `value` (any type), (omissible) : Either #none, to indicate that no layout will be imposed, or one of the four possible predefined layouts: #stack, #split, #horizontal or #vertical. This layout will be applied to both experiment and simulation display views. In addition, it is possible to define a custom layout using the horizontal() and vertical() operators
- `tabs` (boolean): Whether the displays should show their tab or not
- `toolbars` (boolean): Whether the displays should show their toolbar or not

16.4.28.2 Definition

Represents the layout of the display views of simulations and experiments

16.4.28.3 Usages

- For instance, this layout statement will allow to split the screen occupied by displays in four equal parts, with no tabs. Pairs of display::weight represent the number of the display in their order of definition and their respective weight within a horizontal and vertical section

```
layout horizontal([vertical([0::5000,1::5000])::5000,vertical([2::5000,3::5000])::5000]) tabs: false; ```



#### Embedments
* The `layout` statement is of type: **Output**
* The `layout` statement can be embedded into: Experiment,
* The `layout` statement embeds statements:

----

[//]: # (keyword|statement_let)
### let
```

Facets

  * **`name`** (a new identifier), (omissible) : The name of the variable declared
 * `index` (a datatype identifier): The type of the index if this declaration concerns a container
 * `of` (a datatype identifier): The type of the contents if this declaration concerns a container
  * `type` (a datatype identifier): The type of the variable
  * `value` (any type): The value assigned to this variable

Definition

Allows to declare a temporary variable of the specified type and to initialize it with a value

Usages

Embedments
* The `let` statement is of type: **Single statement**
* The `let` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `let` statement embeds statements:

[//]: # (keyword|statement_light)
light
Facets

  * **`id`** (int), (omissible) : a number from 1 to 7 to specify which light we are using
 * `active` (boolean): a boolean expression telling if you want this light to be switch on or not. (default va
 * `color` (any type in [int, rgb]): an int / rgb / rgba value to specify the color and the intensity of the li
 * `direction` (point): the direction of the light (only for direction and spot light). (default value : {0.5
  * `draw_light` (boolean): draw or not the light. (default value : false).
 * `linear_attenuation` (float): the linear attenuation of the positionnal light. (default value : 0)
 * `position` (point): the position of the light (only for point and spot light). (default value : {0,0,1})
 * `quadratic_attenuation` (float): the linear attenuation of the positionnal light. (default value : 0)
 * `spot_angle` (float): the angle of the spot light in degree (only for spot light). (default value : 45)
 * `type` (a label): the type of light to create. A value among {point, direction, spot}. (default value : di
  * `update` (boolean): specify if the light has to be updated. (default value : true).

Definition

`light` allows to define diffusion lights in your 3D display.

Usages

* The general syntax is:

light 1 type:point position:{20,20,20} color:255, linear_attenuation:0.01 quadratic_attenuation:0.0001 draw_light:true update:false light 2 type:spot position:{20,20,20} direction:{0,0,-1} color:255 spot_angle:25 linear_attenuation:0.01 quadratic_attenuation:0.0001 draw_light:true update:false light 3 type:point direction:{1,1,-1} color:255 draw_light:true update:false “‘

- See also: display,

16.4.28.4 Embedments

- The `light` statement is of type: **Layer**
- The `light` statement can be embedded into: display,
- The `light` statement embeds statements:

16.4.29 loop

16.4.29.1 Facets

- `name` (a new identifier), (omissible) : a temporary variable name
- `from` (int): an int expression
- `over` (any type in [container, point]): a list, point, matrix or map expression
- `step` (int): an int expression
- `times` (int): an int expression
- `to` (int): an int expression
- `while` (boolean): a boolean expression

16.4.29.2 Definition

Allows the agent to perform the same set of statements either a fixed number of times, or while a condition is true, or by progressing in a collection of elements or along an interval of integers. Be aware that there are no prevention of infinite loops. As a consequence, open loops should be used with caution, as one agent may block the execution of the whole model.

16.4.29.3 Usages

- The basic syntax for repeating a fixed number of times a set of statements is:

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

`* The basic syntax for repeating a set of statements while a condition holds is:`

loop while: a_bool_expression { // statements } “‘

- The basic syntax for repeating a set of statements by progressing over a container of a point is:

`loop a_temp_var over: a_collection_expression {   // [statements] } ````

`* The basic syntax for repeating a set of statements while an index iterates over a range of values with a fix`

loop a_temp_var from: int_expression_1 to: int_expression_2 { // statements } “‘

- The incrementation step of the index can also be chosen:

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

`* In these latter three cases, the name facet designates the name of a temporary variable, whose scope is the`

int a <- 0; loop i over: [10, 20, 30] { a <- a + i; } // a now equals 60 “‘

- The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to facets take an integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the inclusive interval (i.e. [to, from]). If the step is not defined, it is assumed to be equal to 1 or -1, depending on the direction of the range. If it is defined, its sign will be respected, so that a positive step will never allow the loop to enter a loop from i to j where i is greater than j

```
list the_list <-list (species_of (self)); loop i from: 0 to: length (the_list) - 1 {    ask the_list at i {



#### Embedments
* The `loop` statement is of type: **Sequence of statements or action**
* The `loop` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `loop` statement embeds statements:

----

[//]: # (keyword|statement_match)
### match
#### Facets

  * `value` (any type), (omissible) : The value or values this statement tries to match

#### Definition

In a switch...match structure, the value of each match block is compared to the value in the switch. If they m

#### Usages

* match block is executed if the switch value is equals to the value of the match:
```

switch 3 { match 1 {write “Match 1”; } match 3 {write “Match 2”; } } “‘

- match_between block is executed if the switch value is in the interval given in value of the match_between:

```
switch 3 {   match_between [1,2] {write "Match OK between [1,2]"; }   match_between [2,5] {write "Match OK be


* match_one block is executed if the switch value is equals to one of the values of the match_one:
```

switch 3 { match_one [0,1,2] {write “Match OK with one of [0,1,2]”; } match_between [2,3,4,5] {write “Match OK with one of [2,3,4,5]”; } } “‘

- See also: switch, default,

16.4.29.4 Embedments

- The `match` statement is of type: **Sequence of statements or action**
- The `match` statement can be embedded into: switch,
- The `match` statement embeds statements:

16.4.30 migrate

16.4.30.1 Facets

- **source** (any type in [agent, species, container, an identifier]), (omissible) : can be an agent, a list of agents, a agent's population to be migrated

- **target** (species): target species/population that source agent(s) migrate to.
- **returns** (a new identifier): the list of returned agents in a new local variable

16.4.30.2 Definition

This command permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Species of source agents and target species respect the following constraints: (i) they are "peer" species (sharing the same direct macro-species), (ii) they have sub-species vs. parent-species relationship.

16.4.30.3 Usages

- It can be used in a 3-levels model, in case where individual agents can be captured into group meso agents and groups into clouds macro agents. migrate is used to allows agents captured by groups to migrate into clouds. See the model 'Balls, Groups and Clouds.gaml' in the library.

```
migrate ball_in_group target: ball_in_cloud; ```


* See also: [capture](#capture), [release](#release),

#### Embedments
* The `migrate` statement is of type: **Sequence of statements or action**
* The `migrate` statement can be embedded into: Behavior, Sequence of statements or action,
* The `migrate` statement embeds statements:

----

[//]: # (keyword|statement_monitor)
### monitor
#### Facets

  * **`name`** (a label), (omissible) : identifier of the monitor
  * **`value`** (any type): expression that will be evaluated to be displayed in the monitor
  * `color` (rgb): Indicates the (possibly dynamic) color of this output (default is a light gray)
  * `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true
  * `refresh_every` (int): Allows to refresh the monitor every n time steps (default is 1)

#### Definition

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

#### Usages

* An example of use is:
```

monitor "nb preys" value: length(prey as list) refresh_every: 5; "'

16.4.30.4 Embedments

- The `monitor` statement is of type: **Output**
- The `monitor` statement can be embedded into: output, permanent,
- The `monitor` statement embeds statements:

16.4.31 norm

16.4.31.1 Facets

- `name` (an identifier), (omissible) :
- `finished_when` (boolean):
- `instantaneous` (boolean):
- `intention` (546704):
- `lifetime` (int):
- `obligation` (546704):
- `priority` (float):
- `threshold` (float):
- `when` (boolean):

16.4.31.2 Embedments

- The `norm` statement is of type: **Behavior**
- The `norm` statement can be embedded into: Species, Model,
- The `norm` statement embeds statements:

16.4.32 output

16.4.32.1 Facets

16.4.32.2 Definition

`output` blocks define how to visualize a simulation (with one or more display blocks that define separate windows). It will include a set of displays, monitors and files statements. It will be taken into account only if the experiment type is `gui`.

16.4.32.3 Usages

- Its basic syntax is:

```
experiment exp_name type: gui {   // [inputs]   output {     // [display, file, inspect, layout or monitor st


* See also: [display](#display), [monitor](#monitor), [inspect](#inspect), [output_file](#output_file), [I

#### Embedments
* The `output` statement is of type: **Output**
* The `output` statement can be embedded into: Model, Experiment,
```

* The `output` statement embeds statements: [display](#display), [inspect](#inspect), [monitor](#monitor)

[//]: # (keyword|statement_output_file)
output_file
Facets

 * **`name`** (an identifier), (omissible) : The name of the file where you want to export the data
 * **`data`** (string): The data you want to export
 * `footer` (string): Define a footer for your export file
 * `header` (string): Define a header for your export file
 * `refresh` (boolean): Indicates the condition under which this file should be saved (default is true)
 * `refresh_every` (int): Allows to save the file every n time steps (default is 1)
 * `rewrite` (boolean): Rewrite or not the existing file
 * `type` (an identifier), takes values in: {csv, text, xml}: The type of your output data

Definition

Represents an output that writes the result of expressions into a file

Usages

Embedments
* The `output_file` statement is of type: **Output**
* The `output_file` statement can be embedded into: output, permanent,
* The `output_file` statement embeds statements:

[//]: # (keyword|statement_overlay)
overlay
Facets

 * `background` (rgb): the background color of the overlay displayed inside the view (the bottom overlay rema
 * `border` (rgb): Color to apply to the border of the rectangular shape of the overlay. Nil by default
 * `center` (any type): an expression that will be evaluated and displayed in the center section of the botto
 * `color` (any type in [list, rgb]): the color(s) used to display the expressions given in the 'left', 'cent
 * `left` (any type): an expression that will be evaluated and displayed in the left section of the bottom ov
 * `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[,
 * `right` (any type): an expression that will be evaluated and displayed in the right section of the bottom
 * `rounded` (boolean): Whether or not the rectangular shape of the overlay should be rounded. True by defaul
 * `size` (point): extent of the layer in the view from its position. Coordinates in [0,1[are treated as per
 * `transparency` (float): the transparency rate of the overlay (between 0 and 1, 1 means no transparency) wh

Definition

`overlay` allows the modeler to display a line to the already existing bottom overlay, where the results of '

Usages

* To display information in the bottom overlay, the syntax is:

overlay "Cycle:" + (cycle) center: "Duration:" + total_duration + "ms" right: "Model time:" + as_date(time,"") color: [#yellow, #orange, #yellow]; "'

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

16.4.32.4 Embedments

- The `overlay` statement is of type: **Layer**
- The `overlay` statement can be embedded into: display,
- The `overlay` statement embeds statements:

16.4.33 parameter

16.4.33.1 Facets

- `var` (an identifier): the name of the variable (that should be declared in the global)
- `name` (a label), (omissible) : The message displayed in the interface
- `among` (list): the list of possible values
- `category` (a label): a category label, used to group parameters in the interface
- `disables` (list): a list of global variables whose parameter editors will be disabled when this parameter value is set to true (they are otherwise enabled)
- `enables` (list): a list of global variables whose parameter editors will be enabled when this parameter value is set to true (they are otherwise disabled)
- `init` (any type): the init value
- `max` (any type): the maximum value
- `min` (any type): the minimum value
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the parameter changes
- `slider` (boolean): Whether or not to display a slider for entering an int or float value. Default is true when max and min values are defined, false otherwise. If no max or min value is defined, setting this facet to true will have no effect
- `step` (float): the increment step (mainly used in batch mode to express the variation step between simulation)
- `type` (a datatype identifier): the variable type
- `unit` (a label): the variable unit

16.4.33.2 Definition

The parameter statement specifies which global attributes (i) will change through the successive simulations (in batch experiments), (ii) can be modified by user via the interface (in gui experiments). In GUI experiments, parameters are displayed depending on their type.

16.4.33.3 Usages

- In gui experiment, the general syntax is the following:

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

```
* In batch experiment, the two following syntaxes can be used to describe the possible values of a parameter:
```

parameter 'Value of toto:' var: toto among: [1, 3, 7, 15, 100]; parameter 'Value of titi:' var: titi min: 1 max: 100 step: 2; "'

16.4.33.4 Embedments

- The `parameter` statement is of type: **Parameter**
- The `parameter` statement can be embedded into: Experiment,
- The `parameter` statement embeds statements:

16.4.34 perceive

16.4.34.1 Facets

- **`target`** (any type in [container, agent]): the list of the agent you want to perceive
- **`name`** (an identifier), (omissible) : the name of the perception
- **`as`** (species): an expression that evaluates to a species
- **`emotion`** (546706): The emotion needed to do the perception
- **`in`** (any type in [float, geometry]): a float or a geometry. If it is a float, it's a radius of a detection area. If it is a geometry, it is the area of detection of others species.
- **`parallel`** (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- **`threshold`** (float): Threshold linked to the emotion.
- **`when`** (boolean): a boolean to tell when does the perceive is active

16.4.34.2 Definition

Allow the agent, with a bdi architecture, to perceive others agents

16.4.34.3 Usages

- the basic syntax to perceive agents inside a circle of perception

```
perceive name_of-perception target: the_agents_you_want_to_perceive in: a_distance when: a_certain_conditi



#### Embedments
* The `perceive` statement is of type: **Sequence of statements or action**
* The `perceive` statement can be embedded into: Species, Model,
* The `perceive` statement embeds statements:

----

[//]: # (keyword|statement_permanent)
### permanent
#### Facets

 * `layout` (any type), (omissible) : Either #none, to indicate that no layout will be imposed, or one of the
```

```
  * `tabs` (boolean): Whether the displays should show their tab or not
  * `toolbars` (boolean): Whether the displays should show their toolbar or not

#### Definition

Represents the outputs of the experiment itself. In a batch experiment, the permanent section allows to defi

#### Usages

* For instance, this permanent section will allow to display for each simulation the end value of the food_ga
```

permanent { display Ants background: rgb('white') refresh_every: 1 { chart "Food Gathered" type: series { data "Food" value: food_gathered; } } } "'

16.4.34.4 Embedments

- The `permanent` statement is of type: **Output**
- The `permanent` statement can be embedded into: Experiment,
- The `permanent` statement embeds statements: display, inspect, monitor, output_file,

---

16.4.35 plan

16.4.35.1 Facets

- `name` (an identifier), (omissible) :
- `emotion` (546706):
- `finished_when` (boolean):
- `instantaneous` (boolean):
- `intention` (546704):
- `priority` (float):
- `threshold` (float):
- `when` (boolean):

16.4.35.2 Embedments

- The `plan` statement is of type: **Behavior**
- The `plan` statement can be embedded into: Species, Model,
- The `plan` statement embeds statements:

16.4.36 put

16.4.36.1 Facets

- `in` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression
- `all` (any type): any expression
- `at` (any type): any expression

- `edge` (any type): Indicates that the item to put should be considered as an edge of the receiving graph. Soon to be deprecated, use 'put edge(item)…' instead
- `key` (any type): any expression
- `weight` (float): an expression that evaluates to a float

16.4.36.2 Definition

Allows the agent to replace a value in a container at a given position (in a list or a map) or for a given key (in a map). Note that the behavior and the type of the attributes depends on the specific kind of container.

16.4.36.3 Usages

- The allowed parameters configurations are the following ones:

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

```
* In the case of a list, the position should an integer in the bound of the list. The facet all: is used to repl
```

list putList <- [1,2,3,4,5]; //putList equals [1,2,3,4,5]put -10 at: 1 in: putList;//putList equals [1,-10,3,4,5]put 10 all: true in: putList;//putList equals [10,10,10,10,10]"'

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

```
matrix<int>
putMatrix <- matrix([[0,1],[2,3]]); //putMatrix equals matrix([[0,1],[2,3]])put -10 at: {1,1} in: putMatrix
```

```
* In the case of a map, the position should be one of the key values of the map. Notice that if the given key va
```

map putMap <- ["x"::4,"y"::7]; //putMap equals ["x"::4,"y"::7]put -10 key: "y" in: putMap;//putMap equals ["x"::4,"y"::-10]put -20 key: "z" in: putMap;//putMap equals ["x"::4,"y"::-10, "z"::-20]put -30 all: true in: putMap;//putMap equals ["x"::-30,"y"::-30, "z"::-30]"'

16.4.36.4 Embedments

- The `put` statement is of type: **Single statement**
- The `put` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `put` statement embeds statements:

16.4.37 reactive_tabu

16.4.37.1 Facets

- **`name`** (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `cycle_size_max` (int): minimal size of the considered cycles
- `cycle_size_min` (int): maximal size of the considered cycles
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize

- `nb_tests_wthout_col_max` (int): number of movements without collision before shortening the tabu list
- `tabu_list_size_init` (int): initial size of the tabu list
- `tabu_list_size_max` (int): maximal size of the tabu list
- `tabu_list_size_min` (int): minimal size of the tabu list

16.4.37.2 Definition

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle. See [batch161 the batch dedicated page].

16.4.37.3 Usages

- As other batch methods, the basic syntax of the reactive_tabu statement uses `method reactive_tabu` instead of the expected `reactive_tabu name: id` :

```
method reactive_tabu [facet: value]; ```


* For example:
```

method reactive_tabu iter_max: 50 tabu_list_size_init: 5 tabu_list_size_min: 2 tabu_list_size_max: 10 nb_tests_wthout_col_max: 20 cycle_size_min: 2 cycle_size_max: 20 maximize: food_gathered; “‘

16.4.37.4 Embedments

- The `reactive_tabu` statement is of type: **Batch method**
- The `reactive_tabu` statement can be embedded into: Experiment,
- The `reactive_tabu` statement embeds statements:

16.4.38 reflex

16.4.38.1 Facets

- `name` (an identifier), (omissible) : the identifier of the reflex
- `when` (boolean): an expression that evaluates a boolean, the condition to fulfill in order to execute the statements embedded in the reflex.

16.4.38.2 Definition

Reflexes are sequences of statements that can be executed by the agent. Reflexes prefixed by the ‘reflex’ keyword are executed continuously. Reflexes prefixed by ‘init’ are executed only immediately after the agent has been created. Reflexes prefixed by 'abort' just before the agent is killed. If a facet when: is defined, a reflex is executed only if the boolean expression evaluates to true.

16.4.38.3 Usages

- Example:

````
reflex my_reflex when: flip (0.5){    //Only executed when flip returns true    write "Executing the uncondi



#### Embedments
* The `reflex` statement is of type: **Behavior**
* The `reflex` statement can be embedded into: Species, Experiment, Model,
* The `reflex` statement embeds statements:

----

[//]: # (keyword|statement_release)
### release
#### Facets

  * **`target`** (any type in [agent, list, 27]), (omissible) : an expression that is evaluated as an agent/a
  * `as` (species): an expression that is evaluated as a species in which the micro-agent will be released
  * `in` (agent): an expression that is evaluated as an agent that will be the macro-agent in which micro-agen
  * `returns` (a new identifier): a new variable containing a list of the newly released agent(s)

#### Definition

Allows an agent to release its micro-agent(s). The preliminary for an agent to release its micro-agents is th

#### Usages

* We consider the following species. Agents of "C" species can be released from a "B" agent to become agents o
````

species A { … } species B { … species C parent: A { … } species D { … } … } "'

- To release all "C" agents from a "B" agent, agent "C" has to execute the following statement. The "C" agent will change to "A" agent. The won't consider "B" agent as their macro-agent (host) anymore. Their host (macro-agent) will the be the host (macro-agent) of the "B" agent.

````
release list(C); ```


* The modeler can specify the new host and the new species of the released agents:
````

release list (C) as: new_species in: new host; "'

- See also: capture,

16.4.38.4 Embedments

- The `release` statement is of type: **Sequence of statements or action**
- The `release` statement can be embedded into: Behavior, Sequence of statements or action,
- The `release` statement embeds statements:

16.4.39 remove

16.4.39.1 Facets

- **`from`** (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression to remove from the container
- `all` (any type): an expression that evaluates to a container. If it is true and if the value a list, it removes the first instance of each element of the list. If it is true and the value is not a container, it will remove all instances of this value.
- `edge` (any type): Indicates that the item to remove should be considered as an edge of the receiving graph
- `index` (any type): any expression, the key at which to remove the element from the container
- `key` (any type): any expression, the key at which to remove the element from the container
- `node` (any type): Indicates that the item to remove should be considered as a node of the receiving graph
- `vertex` (any type):

16.4.39.2 Definition

Allows the agent to remove an element from a container (a list, matrix, map…).

16.4.39.3 Usages

- This statement should be used in the following ways, depending on the kind of container used and the expected action on it:

```
remove expr from: expr_container; remove index: expr from: expr_container; remove key: expr from: expr_conta
```

```
* In the case of list, the facet `item:` is used to remove the first occurence of a given expression, whereas
```

list removeList <- [3,2,1,2,3]; remove 2 from: removeList;//removeList equals [3,1,2,3]remove 3 all: true from: removeList;//removeList equals [1,2]remove index: 1 from: removeList;//removeList equals [1]"'

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

```
map<string,int> removeMap <- ["x"::5, "y"::7, "z"::7]; remove key: "x" from: removeMap;//removeMap equals [
```

```
* In addition, a map a be managed as a list with pair key as index. Given that, facets item:, all: and index: c
```

map removeMapList <- ["x"::5, "y"::7, "z"::7, "t"::5]; remove 7 from: removeMapList;//removeMapList equals ["x"::5, "z"::7, "t"::5]remove [5,7] all: true from: removeMapList;//removeMapList equals ["t"::5]remove index: "t" from: removeMapList;//removeMapList equals map([])"'

- In the case of a graph, both edges and nodes can be removes using node: and edge facets. If a node is removed, all edges to and from this node are also removed.

```
graph removeGraph <- as_edge_graph([{1,2}::{3,4},{3,4}::{5,6}]); remove node: {1,2} from: removeGraph; rem
list var <- removeGraph.vertices; // var equals [{3,4},{5,6}]
list var <- removeGraph.edges; // var equals [polyline({3,4}::{5,6})]remove edge: {3,4}::{5,6} from: remove
list var <- removeGraph.vertices; // var equals [{3,4},{5,6}]
list var <- removeGraph.edges; // var equals []```
```

`* In the case of an agent or a shape, `remove` allows to remove an attribute from the attributes map of the rec`

global { init { create speciesRemove; speciesRemove sR <- speciesRemove(0); // sR.a now equals 100 remove key:"a" from: sR; // sR.a now equals nil } } species speciesRemove { int a <- 100; } "'

- This statement can not be used on *matrix*.
- See also: add, put,

16.4.39.4 Embedments

- The `remove` statement is of type: **Single statement**
- The `remove` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `remove` statement embeds statements:

16.4.40 return

16.4.40.1 Facets

- `value` (any type), (omissible) : an expression that is returned

16.4.40.2 Definition

Allows to immediately stop and tell which value to return from the evaluation of the surrounding action or top-level statement (reflex, init, etc.). Usually used within the declaration of an action. For more details about actions, see the following [Section161 section].

16.4.40.3 Usages

- Example:

```
string foo {    return "foo"; } reflex {    string foo_result <- foo();    // foos_result is now equals to "f
```

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

// In Species A: string foo_different { return "foo_not_same"; } /// .... // In Species B: reflex writing { string temp <- some_agent_A.foo_different []; // temp is now equals to "foo_not_same" } "'

16.4.40.4 Embedments

- The `return` statement is of type: **Single statement**
- The `return` statement can be embedded into: action, Behavior, Sequence of statements or action,
- The `return` statement embeds statements:

16.4.41 rule

16.4.41.1 Facets

- `name` (an identifier), (omissible) : The name of the rule
- `all` (boolean): add a desire for each belief
- `belief` (546704): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `desire` (546704): The mandatory desire
- `desires` (list): The mandatory desires
- `emotion` (546706): The mandatory emotion
- `emotions` (list): The mandatory emotions
- `ideal` (546704): The mandatory ideal
- `ideals` (list): The mandatory ideals
- `lifetime` (int): the lifetime value of the mental state created
- `new_belief` (546704): The belief that will be added
- `new_beliefs` (list): The belief that will be added
- `new_desire` (546704): The desire that will be added
- `new_desires` (list): The desire that will be added
- `new_emotion` (546706): The emotion that will be added
- `new_emotions` (list): The emotion that will be added
- `new_ideal` (546704): The ideal that will be added
- `new_ideals` (list): The ideals that will be added
- `new_uncertainties` (list): The uncertainty that will be added
- `new_uncertainty` (546704): The uncertainty that will be added
- `obligation` (546704): The mandatory obligation
- `obligations` (list): The mandatory obligations
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `remove_belief` (546704): The belief that will be removed
- `remove_beliefs` (list): The belief that will be removed
- `remove_desire` (546704): The desire that will be removed
- `remove_desires` (list): The desire that will be removed
- `remove_emotion` (546706): The emotion that will be removed
- `remove_emotions` (list): The emotion that will be removed
- `remove_ideal` (546704): The ideal that will be removed
- `remove_ideals` (list): The ideals that will be removed
- `remove_intention` (546704): The intention that will be removed
- `remove_obligation` (546704): The obligation that will be removed
- `remove_obligations` (list): The obligation that will be removed
- `remove_uncertainties` (list): The uncertainty that will be removed
- `remove_uncertainty` (546704): The uncertainty that will be removed
- `strength` (any type in [float, int]): The stregth of the mental state created
- `threshold` (float): Threshold linked to the emotion.
- `uncertainties` (list): The mandatory uncertainties
- `uncertainty` (546704): The mandatory uncertainty
- `when` (boolean):

16.4.41.2 Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

16.4.41.3 Usages

- Other examples of use: `rule belief: new_predicate("test") when: flip(0.5) new_desire: new_predicate("test")`

16.4.41.4 Embedments

- The `rule` statement is of type: **Single statement**
- The `rule` statement can be embedded into: Species, Model,
- The `rule` statement embeds statements:

16.4.42 run

16.4.42.1 Facets

- **`name`** (string), (omissible) :

- **`of`** (string):
- `core` (int):
- `end_cycle` (int):
- `seed` (int):
- `with_output` (map):
- `with_param` (map):

16.4.42.2 Embedments

- The `run` statement is of type: **Sequence of statements or action**
- The `run` statement can be embedded into: Behavior, Single statement, Species, Model,
- The `run` statement embeds statements:

16.4.43 sanction

16.4.43.1 Facets

- `name` (an identifier), (omissible) :

16.4.43.2 Embedments

- The `sanction` statement is of type: **Behavior**
- The `sanction` statement can be embedded into: Species, Model,
- The `sanction` statement embeds statements:

16.4.44 save

16.4.44.1 Facets

- `data` (any type), (omissible) : any expression, that will be saved in the file
- `attributes` (map): Allows to specify the attributes of a shape file where agents are saved. The keys of the map are the names of the attributes that will be present in the file, the values are whatever expressions neeeded to define their value
- `crs` (any type): the name of the projection, e.g. crs:"EPSG:4326" or its EPSG id, e.g. crs:4326. Here a list of the CRS codes (and EPSG id): http://spatialreference.org
- `header` (boolean): an expression that evaluates to a boolean, specifying whether the save will write a header if the file does not exist
- `rewrite` (boolean): an expression that evaluates to a boolean, specifying whether the save will ecrase the file or append data at the end of it. Default is true
- `to` (string): an expression that evaluates to an string, the path to the file, or directly to a file
- `type` (an identifier), takes values in: {shp, text, csv, asc, geotiff, image}: an expression that evaluates to an string, the type of the output file (it can be only "shp", "asc", "geotiff", "image", "text" or "csv")
- `with` (map): Allows to define the attributes of a shape file. Keys of the map are the attributes of agents to save, values are the names of attributes in the shape file

16.4.44.2 Definition

Allows to save data in a file. The type of file can be "shp", "asc", "geotiff", "text" or "csv".

16.4.44.3 Usages

- Its simple syntax is:

`save data to: output_file type: a_type_file; ````

`* To save data in a text file:`

save (string(cycle) + "->" + name + ":" + location) to: "save_data.txt" type: "text"; "'

- To save the values of some attributes of the current agent in csv file:

`save [name, location, host] to: "save_data.csv" type: "csv"; ````

`* To save the values of all attributes of all the agents of a species into a csv (with optional attributes):`

save species_of(self) to: "save_csvfile.csv" type: "csv" header: false; "'

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

`save species_of(self) to: "save_shapefile.shp" type: "shp" with: [name::"nameAgent", location::"locationAg`

`* To save the grid_value attributes of all the cells of a grid into an ESRI ASCII Raster file:`

save grid to: "save_grid.asc" type: "asc"; "'

- To save the grid_value attributes of all the cells of a grid into geotiff:

`save grid to: "save_grid.tif" type: "geotiff"; ```

`* To save the grid_value attributes of all the cells of a grid into png (with a worldfile):`

save grid to: “save_grid.png” type: “image”; “‘

- The save statement can be use in an init block, a reflex, an action or in a user command. Do not use it in experiments.

16.4.44.4 Embedments

- The `save` statement is of type: **Single statement**
- The `save` statement can be embedded into: Behavior, Sequence of statements or action,
- The `save` statement embeds statements:

16.4.45 set

16.4.45.1 Facets

- **`name`** (any type), (omissible) : the name of an existing variable or attribute to be modified
- **`value`** (any type): the value to affect to the variable or attribute

16.4.45.2 Definition

Allows to assign a value to the variable or attribute specified

16.4.45.3 Usages

16.4.45.4 Embedments

- The `set` statement is of type: **Single statement**
- The `set` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `set` statement embeds statements:

16.4.46 setup

16.4.46.1 Facets

16.4.46.2 Definition

The setup statement is used to define the set of instructions that will be executed before every [#test test].

16.4.46.3 Usages

- As every test should be independent from the others, the setup will mainly contain initialization of variables that will be used in each test.

```
species Tester {    int val_to_test;    setup {       val_to_test <- 0;    }    test t1 {      // [set of inst


* See also: [test](#test), [assert](#assert),

#### Embedments
* The `setup` statement is of type: **Sequence of statements or action**
* The `setup` statement can be embedded into: Species, Experiment, Model,
* The `setup` statement embeds statements:

----

[//]: # (keyword|statement_simulate)
### simulate
#### Facets

  * **`comodel`** (file), (omissible) :
  * `repeat` (int):
  * `reset` (boolean):
  * `share` (list):
  * `until` (boolean):
  * `with_experiment` (string):
  * `with_input` (map):
  * `with_output` (map):

#### Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or othe

#### Usages

* Other examples of use:
```

ask receiver_agent(s) { // statements } “‘

16.4.46.4 Embedments

- The `simulate` statement is of type: **Single statement**
- The `simulate` statement can be embedded into: chart, Experiment, Species, Behavior, Sequence of statements or action,
- The `simulate` statement embeds statements:

16.4.47 socialize

16.4.47.1 Facets

- `name` (an identifier), (omissible) : the identifier of the socialize statement

- `agent` (agent): the agent value of the created social link
- `dominance` (float): the dominance value of the created social link
- `familiarity` (float): the familiarity value of the created social link
- `liking` (float): the appreciation value of the created social link
- `solidarity` (float): the solidarity value of the created social link
- `trust` (float): the trust value of the created social link
- `when` (boolean): A boolean value to socialize only with a certain condition

16.4.47.2 Definition

enables to directly add a social link from a perceived agent.

16.4.47.3 Usages

- Other examples of use: `socialize;`

16.4.47.4 Embedments

- The `socialize` statement is of type: **Single statement**
- The `socialize` statement can be embedded into: Behavior, Sequence of statements or action,
- The `socialize` statement embeds statements:

16.4.48 solve

16.4.48.1 Facets

- `equation` (an identifier), (omissible) : the equation system identifier to be numerically solved
- `cycle_length` (int): length of simulation cycle which will be synchronize with step of integrator (default value: 1)
- `discretizing_step` (int): number of discrete between 2 steps of simulation (default value: 0)
- `integrated_times` (list): time interval inside integration process
- `integrated_values` (list): list of variables's value inside integration process
- `max_step` (float): maximal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- `method` (an identifier), takes values in: {Euler, ThreeEighthes, Midpoint, Gill, Luther, rk4, dp853, AdamsBashforth, AdamsMoulton, DormandPrince54, GraggBulirschStoer, HighamHall54}: integrate method (can be only "Euler", "ThreeEighthes", "Midpoint", "Gill", "Luther", "rk4" or "dp853", "AdamsBashforth", "AdamsMoulton", "DormandPrince54", "GraggBulirschStoer", "HighamHall54") (default value: "rk4")
- `min_step` (float): minimal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- `scalAbsoluteTolerance` (float): allowed absolute error (used with dp853 method only)
- `scalRelativeTolerance` (float): allowed relative error (used with dp853 method only)
- `step` (float): integration step, use with most integrator methods (default value: 1)
- `time_final` (float): target time for the integration (can be set to a value smaller than t0 for backward integration)
- `time_initial` (float): initial time

16.4.48.2 Definition

Solves all equations which matched the given name, with all systems of agents that should solved simultaneously.

16.4.48.3 Usages

- Other examples of use: `solve SIR method: "rk4" step:0.001;`

16.4.48.4 Embedments

- The `solve` statement is of type: **Single statement**
- The `solve` statement can be embedded into: Behavior, Sequence of statements or action,
- The `solve` statement embeds statements:

16.4.49 species

16.4.49.1 Facets

- **`name`** (an identifier), (omissible) : the identifier of the species
- **`cell_height`** (float): (grid only), the height of the cells of the grid
- **`cell_width`** (float): (grid only), the width of the cells of the grid
- **`compile`** (boolean):
- **`control`** (22): defines the architecture of the species (e.g. fsm…)
- **`edge_species`** (species): In the case of a species defining a graph topology for its instances (nodes of the graph), specifies the species to use for representing the edges
- **`file`** (file): (grid only), a bitmap file that will be loaded at runtime so that the value of each pixel can be assigned to the attribute 'grid_value'
- **`files`** (list): (grid only), a list of bitmap file that will be loaded at runtime so that the value of each pixel of each file can be assigned to the attribute 'bands'
- **`frequency`** (int): The execution frequency of the species (default value: 1). For instance, if frequency is set to 10, the population of agents will be executed only every 10 cycles.
- **`height`** (int): (grid only), the height of the grid (in terms of agent number)
- **`horizontal_orientation`** (boolean): (hexagonal grid only),(true by default). Allows use a hexagonal grid with a horizontal or vertical orientation.
- **`mirrors`** (any type in [list, species]): The species this species is mirroring. The population of this current species will be dependent of that of the species mirrored (i.e. agents creation and death are entirely taken in charge by GAMA with respect to the demographics of the species mirrored). In addition, this species is provided with an attribute called 'target', which allows each agent to know which agent of the mirrored species it is representing.
- **`neighbors`** (int): (grid only), the chosen neighborhood (4, 6 or 8)
- **`neighbours`** (int): (grid only), the chosen neighborhood (4, 6 or 8)
- **`optimizer`** (string): (grid only),("A*" by default). Allows to specify the algorithm for the shortest path computation ("BF", "Dijkstra", "A*" or "JPS*"
- **`parallel`** (any type in [boolean, int]): (experimental) setting this facet to 'true' will allow this species to use concurrency when scheduling its agents; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet has a default set in the preferences (Under Performances > Concurrency)
- **`parent`** (species): the parent class (inheritance)

- `schedules` (container): A container of agents (a species, a dynamic list, or a combination of species and containers) , which represents which agents will be actually scheduled when the population is scheduled for execution. For instance, 'species a schedules: (10 among a)' will result in a population that schedules only 10 of its own agents every cycle. 'species b schedules: []' will prevent the agents of 'b' to be scheduled. Note that the scope of agents covered here can be larger than the population, which allows to build complex scheduling controls; for instance, defining 'global schedules: [] {...} species b schedules: []; species c schedules: b + world;' allows to simulate a model where the agents of b are scheduled first, followed by the world, without even having to create an instance of c.
- `skills` (list): The list of skills that will be made available to the instances of this species. Each new skill provides attributes and actions that will be added to the ones defined in this species
- `topology` (topology): The topology of the population of agents defined by this species. In case of nested species, it can for example be the shape of the macro-agent. In case of grid or graph species, the topology is automatically computed and cannot be redefined
- `torus` (boolean): is the topology toric (defaut: false). Needs to be defined on the global species.
- `use_individual_shapes` (boolean): (grid only),(true by default). Allows to specify whether or not the agents of the grid will have distinct geometries. If set to false, they will all have simpler proxy geometries
- `use_neighbors_cache` (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbors cache used for grids. Note that if a diffusion of variable occurs, GAMA will emit a warning and automatically switch to a caching version
- `use_regular_agents` (boolean): (grid only),(true by default). Allows to specify if the agents of the grid are regular agents (like those of any other species) or minimal ones (which can't have sub-populations, can't inherit from a regular species, etc.)
- `virtual` (boolean): whether the species is virtual (cannot be instantiated, but only used as a parent) (false by default)
- `width` (int): (grid only), the width of the grid (in terms of agent number)

16.4.49.2 Definition

The species statement allows modelers to define new species in the model. `global` and `grid` are speciel cases of species: `global` being the definition of the global agent (which has automatically one instance, world) and `grid` being a species with a grid topology.

16.4.49.3 Usages

- Here is an example of a species definition with a FSM architecture and the additional skill moving:

`species ant skills: [moving] control: fsm { ```

```
* In the case of a species aiming at mirroring another one:
```

species node_agent mirrors: list(bug) parent: graph_node edge_species: edge_agent { "'

- The definition of the single grid of a model will automatically create gridwidth x gridheight agents:

`grid ant_grid width: gridwidth height: gridheight file: grid_file neighbors: 8 use_regular_agents: false {`

```
* Using a file to initialize the grid can replace width/height facets:
```

grid ant_grid file: grid_file neighbors: 8 use_regular_agents: false { "'

16.4.49.4 Embedments

- The `species` statement is of type: **Species**
- The `species` statement can be embedded into: Model, Environment, Species,
- The `species` statement embeds statements:

16.4.50 start_simulation

16.4.50.1 Facets

- **`name`** (string), (omissible) :

- **`of`** (string):
- **`seed`** (int):
- **`with_param`** (map):

16.4.50.2 Embedments

- The `start_simulation` statement is of type: **Sequence of statements or action**
- The `start_simulation` statement can be embedded into: Behavior, Single statement, Species, Model,
- The `start_simulation` statement embeds statements:

16.4.51 state

16.4.51.1 Facets

- **`name`** (an identifier), (omissible) : the identifier of the state
- **`final`** (boolean): specifies whether the state is a final one (i.e. there is no transition from this state to another state) (default value= false)
- **`initial`** (boolean): specifies whether the state is the initial one (default value = false)

16.4.51.2 Definition

A state, like a reflex, can contains several statements that can be executed at each time step by the agent.

16.4.51.3 Usages

- Here is an exemple integrating 2 states and the statements in the FSM architecture:

  `state s_init initial: true {      enter { write "Enter in" + state; }          write "Enter in" + state;`

```
* See also: [enter](#enter), [exit](#exit), [transition](#transition),

#### Embedments
* The `state` statement is of type: **Behavior**
* The `state` statement can be embedded into: fsm, Species, Experiment, Model,
```

```
* The `state` statement embeds statements: [enter](#enter),  [exit](#exit),

----

[//]: # (keyword|statement_status)
### status
#### Facets

  * **`message`** (any type), (omissible) : Allows to display a necessarily short message in the status box in
   * `color` (rgb): The color used for displaying the background of the status message

#### Definition

The statement makes the agent output an arbitrary message in the status box.

#### Usages

* Outputting a message
```

status ('This is my status' + self) color: #yellow; "'

16.4.51.4 Embedments

- The `status` statement is of type: **Single statement**
- The `status` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `status` statement embeds statements:

16.4.52 switch

16.4.52.1 Facets

- **`value`** (any type), (omissible) : an expression

16.4.52.2 Definition

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

16.4.52.3 Usages

- The prototypical syntax is as follows:

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

`* Example:`

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

- See also: match, default, if,

16.4.52.4 Embedments

- The `switch` statement is of type: **Sequence of statements or action**
- The `switch` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `switch` statement embeds statements: default, match,

16.4.53 tabu

16.4.53.1 Facets

- **`name`** (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `tabu_list_size` (int): size of the tabu list

16.4.53.2 Definition

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article and [batch161 the batch dedicated page].

16.4.53.3 Usages

- As other batch methods, the basic syntax of the tabu statement uses `method tabu` instead of the expected `tabu name: id` :

```
method tabu [facet: value]; ```


* For example:
```

method tabu iter_max: 50 tabu_list_size: 5 maximize: food_gathered; “‘

16.4.53.4 Embedments

- The `tabu` statement is of type: **Batch method**
- The `tabu` statement can be embedded into: Experiment,
- The `tabu` statement embeds statements:

16.4.54 task

16.4.54.1 Facets

- **`name`** (an identifier), (omissible) : the identifier of the task

- **weight** (float): the priority level of the task

16.4.54.2 Definition

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

16.4.54.3 Usages

16.4.54.4 Embedments

- The `task` statement is of type: **Behavior**
- The `task` statement can be embedded into: weighted_tasks, sorted_tasks, probabilistic_tasks, Species, Experiment, Model,
- The `task` statement embeds statements:

16.4.55 test

16.4.55.1 Facets

- **name** (an identifier), (omissible) : identifier of the test

16.4.55.2 Definition

The test statement allows modeler to define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue.

16.4.55.3 Usages

- An example of use:

```
species Tester {    // set of attributes that will be used in test     setup {       // [set of instructions...


* See also: [setup](#setup), [assert](#assert),

#### Embedments
* The `test` statement is of type: **Behavior**
* The `test` statement can be embedded into: Species, Experiment, Model,
* The `test` statement embeds statements: [assert](#assert),

----

[//]: # (keyword|statement_trace)
### trace
#### Facets
```

```
#### Definition

All the statements executed in the trace statement are displayed in the console.

#### Usages


#### Embedments
* The `trace` statement is of type: **Sequence of statements or action**
* The `trace` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `trace` statement embeds statements:

----

[//]: # (keyword|statement_transition)
### transition
#### Facets

  * **`to`** (an identifier): the identifier of the next state
 * `when` (boolean), (omissible) : a condition to be fulfilled to have a transition to another given state

#### Definition

In an FSM architecture, `transition` specifies the next state of the life cycle. The transition occurs when t

#### Usages

* In the following example, the transition is executed when after 2 steps:
```

`state s_init initial: true {      write state;      transition to: s1 when: (cycle > 2) {        write "transi`

- See also: enter, state, exit,

16.4.55.4 Embedments

- The `transition` statement is of type: **Sequence of statements or action**
- The `transition` statement can be embedded into: Sequence of statements or action, Behavior,
- The `transition` statement embeds statements:

16.4.56 try

16.4.56.1 Facets

16.4.56.2 Definition

Allows the agent to execute a sequence of statements and to catch any runtime error that might happen in a subsequent `catch` block, either to ignore it (not a good idea, usually) or to safely stop the model

16.4.56.3 Usages

- The generic syntax is:

`try {   [statements] } ```

`* Optionally, the statements to execute when a runtime error happens in the block can be defined in a followin`

try { statements } catch { statements } “‘

16.4.56.4 Embedments

- The `try` statement is of type: **Sequence of statements or action**
- The `try` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `try` statement embeds statements: catch,

16.4.57 unconscious_contagion

16.4.57.1 Facets

- **`emotion`** (546706): the emotion that will be copied with the contagion
- **`name`** (an identifier), (omissible) : the identifier of the unconscious contagion
- **`charisma`** (float): The charisma value of the perceived agent (between 0 and 1)
- **`decay`** (float): The decay value of the emotion added to the agent
- **`receptivity`** (float): The receptivity value of the current agent (between 0 and 1)
- **`threshold`** (float): The threshold value to make the contagion
- **`when`** (boolean): A boolean value to get the emotion only with a certain condition

16.4.57.2 Definition

enables to directly copy an emotion presents in the perceived specie.

16.4.57.3 Usages

- Other examples of use: `unconscious_contagion emotion:fearConfirmed; unconscious_contagion emotion:fearConfirmed charisma: 0.5 receptivity: 0.5;`

16.4.57.4 Embedments

- The `unconscious_contagion` statement is of type: **Single statement**
- The `unconscious_contagion` statement can be embedded into: Behavior, Sequence of statements or action,
- The `unconscious_contagion` statement embeds statements:

16.4.58 user_command

16.4.58.1 Facets

- **`name`** (a label), (omissible) : the identifier of the user_command

- `action` (26): the identifier of the action to be executed. This action should be accessible in the context in which the user_command is defined (an experiment, the global section or a species). A special case is allowed to maintain the compatibility with older versions of GAMA, when the user_command is declared in an experiment and the action is declared in 'global'. In that case, all the simulations managed by the experiment will run the action in response to the user executing the command
- `category` (a label): a category label, used to group parameters in the interface
- `color` (rgb): The color of the button to display
- `continue` (boolean): Whether or not the button, when clicked, should dismiss the user panel it is defined in. Has no effect in other contexts (menu, parameters, inspectors)
- `when` (boolean): the condition that should be fulfilled (in addition to the user clicking it) in order to execute this action
- `with` (map): the map of the parameters::values required by the action

16.4.58.2 Definition

Anywhere in the global block, in a species or in an (GUI) experiment, user_command statements allows to either call directly an existing action (with or without arguments) or to be followed by a block that describes what to do when this command is run.

16.4.58.3 Usages

- The general syntax is for example:

```
user_command kill_myself action: some_action with: [arg1::val1, arg2::val2, ...]; ```


* See also: [user_init](#user_init), [user_panel](#user_panel), [user_input](#user_input),

#### Embedments
* The `user_command` statement is of type: **Sequence of statements or action**
* The `user_command` statement can be embedded into: user_panel, Species, Experiment, Model,
* The `user_command` statement embeds statements: [user_input](#user_input),

----

[//]: # (keyword|statement_user_init)
### user_init
#### Facets

  * **`name`** (an identifier), (omissible) : The name of the panel
  * `initial` (boolean): Whether or not this panel will be the initial one

#### Definition

Used in the user control architecture, user_init is executed only once when the agent is created. It opens a s

#### Usages

* See also: [user_command](#user_command), [user_init](#user_init), [user_input](#user_input),

#### Embedments
* The `user_init` statement is of type: **Behavior**
* The `user_init` statement can be embedded into: Species, Experiment, Model,
```

```
* The `user_init` statement embeds statements: [user_panel](#user_panel),

----

[//]: # (keyword|statement_user_input)
### user_input
#### Facets

 * **`returns`** (a new identifier): a new local variable containing the value given by the user
  * `name` (a label), (omissible) : the displayed name
  * `among` (list): the set of acceptable values for the variable
  * `init` (any type): the init value
  * `max` (float): the maximum value
  * `min` (float): the minimum value
  * `slider` (boolean): Whether to display a slider or not when applicable
  * `type` (a datatype identifier): the variable type

#### Definition

It allows to let the user define the value of a variable.

#### Usages

* Other examples of use:
```

user_panel "Advanced Control" { user_input "Location" returns: loc type: point <- {0,0}; create cells number: 10 with: [location::loc]; } "'

- See also: user_command, user_init, user_panel,

16.4.58.4 Embedments

- The `user_input` statement is of type: **Single statement**
- The `user_input` statement can be embedded into: user_command,
- The `user_input` statement embeds statements:

16.4.59 user_panel

16.4.59.1 Facets

- **`name`** (an identifier), (omissible) : The name of the panel
- **`initial`** (boolean): Whether or not this panel will be the initial one

16.4.59.2 Definition

It is the basic behavior of the user control architecture (it is similar to state for the FSM architecture). This user_panel translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each user_panel, like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of user_commands which describe the different action buttons present in the panel.

16.4.59.3 Usages

- The general syntax is for example:

```
user_panel default initial: true {  user_input 'Number' returns: number type: int <- 10;   ask (number among


* See also: [user_command](#user_command), [user_init](#user_init), [user_input](#user_input),

#### Embedments
* The `user_panel` statement is of type: **Behavior**
* The `user_panel` statement can be embedded into: fsm, user_first, user_last, user_init, user_only, Species
* The `user_panel` statement embeds statements: [user_command](#user_command),

----

[//]: # (keyword|statement_using)
### using
#### Facets

  * **`topology`** (topology), (omissible) : the topology

#### Definition

`using` is a statement that allows to set the topology to use by its sub-statements. They can gather it by ask

#### Usages

* All the spatial operations are topology-dependent (e.g. neighbors are not the same in a continuous and in a
```

float dist <- 0.0; using topology(grid_ant) { d (self.location distance_to target.location); } “‘

16.4.59.4 Embedments

- The `using` statement is of type: **Sequence of statements or action**
- The `using` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `using` statement embeds statements:

16.4.60 Variable_container

16.4.60.1 Facets

- **`name`** (a new identifier), (omissible) : The name of the attribute
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `fill_with` (any type):
- `function` (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both ‘init:’ and ‘update:’
- `index` (a datatype identifier): The type of the key used to retrieve the contents of this attribute
- `init` (any type): The initial value of the attribute
- `of` (a datatype identifier): The type of the contents of this container attribute

- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `size` (any type in [int, point]):
- `type` (a datatype identifier): The type of the attribute
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type):

16.4.60.2 Definition

Allows to declare an attribute of a species or an experiment

16.4.60.3 Usages

16.4.60.4 Embedments

- The `Variable_container` statement is of type: **Variable (container)**
- The `Variable_container` statement can be embedded into: Species, Experiment, Model,
- The `Variable_container` statement embeds statements:

16.4.61 Variable_number

16.4.61.1 Facets

- **`name`** (a new identifier), (omissible) : The name of the attribute
- `among` (list): A list of constant values among which the attribute can take its value
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `function` (any type in [int, float]): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- `init` (any type in [int, float]): The initial value of the attribute
- `max` (any type in [int, float]): The maximum value this attribute can take.
- `min` (any type in [int, float]): The minimum value this attribute can take
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `step` (int): A discrete step (used in conjunction with min and max) that constrains the values this variable can take
- `type` (a datatype identifier): The type of the attribute, either 'int' or 'float'
- `update` (any type in [int, float]): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type in [int, float]):

16.4.61.2 Definition

Allows to declare an attribute of a species or experiment

16.4.61.3 Usages

16.4.61.4 Embedments

- The `Variable_number` statement is of type: **Variable (number)**
- The `Variable_number` statement can be embedded into: Species, Experiment, Model,
- The `Variable_number` statement embeds statements:

16.4.62 Variable_regular

16.4.62.1 Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `among` (list): A list of constant values among which the attribute can take its value
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `function` (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- `index` (a datatype identifier): The type of the index used to retrieve elements if the type of the attribute is a container type
- `init` (any type): The initial value of the attribute
- `of` (a datatype identifier): The type of the elements contained in the type of this attribute if it is a container type
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `type` (a datatype identifier): The type of this attribute. Can be combined with facets 'of' and 'index' to describe container types
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type):

16.4.62.2 Definition

Allows to declare an attribute of a species or an experiment

16.4.62.3 Usages

16.4.62.4 Embedments

- The `Variable_regular` statement is of type: **Variable (regular)**
- The `Variable_regular` statement can be embedded into: Species, Experiment, Model,
- The `Variable_regular` statement embeds statements:

16.4.63 warn

16.4.63.1 Facets

- **`message`** (string), (omissible) : the message to display as a warning.

16.4.63.2 Definition

The statement makes the agent output an arbitrary message in the error view as a warning.

16.4.63.3 Usages

- Emmitting a warning

```
warn 'This is a warning from ' + self; ```



#### Embedments
* The `warn` statement is of type: **Single statement**
* The `warn` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
* The `warn` statement embeds statements:

----

[//]: # (keyword|statement_write)
### write
#### Facets

  * **`message`** (any type), (omissible) : the message to display. Modelers can add some formatting characte
  * `color` (rgb): The color with wich the message will be displayed. Note that different simulations will hav

#### Definition

The statement makes the agent output an arbitrary message in the console.

#### Usages

* Outputting a message
```

write 'This is a message from' + self; “‘

16.4.63.4 Embedments

- The `write` statement is of type: **Single statement**
- The `write` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `write` statement embeds statements: