Full Documentation of GAMA 1.4

Provided by the GAMA development team

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

Table of Contents

Introduction to GAMA 1.4

On this page: Links to the documentation:

Information

Release date: December 2011 [Splash14] < http://gama-platform.googlecode.com/files/splash1.4.png> The version 1.4 provides a new integrated development environment as well as a new modeling language. The new language is not anymore based on XML, and is much easier to read and to write. It integrates new important types such as geometry, graph, path and topology that ease the spatial manipulation of agents. Key points :

- Deep refactoring work of the source code
- New programming language (not based on XML)
- Integration of a true IDE based on Eclipse
- Deep refactoring of the meta-model
- Better integration of multi-levels
- New important notion: topology
- New variable types: geometry, graph, path, topology
- Many more novelties/improvements/enrichments…

News

Compared to GAMA 1.3, GAMA 1.4 provides (non-exhaustive list):

- A new version of the modeling language GAML (which gets rid of its XML roots).
- An IDE for developing models, that can assist modelers in a number of ways (reporting of errors, cross-referencing, completion, etc.)
- New data types (graph, path, container, topology) to better handle complex models
- The addition of "experiments", which allow to explicitly organize experimental and testing tasks
- An improved UI with dynamic inspectors, better handling of parameters, ..
- The inclusion of a new meta-model that relies on inner species and topologies to allow designing multiple scales models
- A comprehensive set of geometrical operators to better handle models with complex topologies and GIS data
- A unified set of commands for managing containers (lists, matrices, graphs, maps, etc.)
- A finer tuning of the scheduling of agents

How to convert models from GAMA 1.3 and GAMA 1.4

Converting a model from GAMA 1.3 to GAMA 1.4 is quite fastidious as the language has changed (it is not based on XML anymore). Even if the operators and the concepts remain the same, converting a model from GAMA 1.3 to GAMA 1.4 requires to rewrite most of the model. As an example, the following species (extracted from the model 2 of the stupid model tutorial) was defined in:

- GAMA 1.3:

- GAMA 1.4:

```
entities {
    species bug {
        var size type: float init: 1;
        var color type: rgb value: rgb [255, 255/size, 255/size];

        reflex basic_move {
            let place value: location as stupid_grid;
            let destination value: one_of ((place neighbours_at 4) where
empty(each.agents));
            if condition: destination != nil {
                set location value: destination;
            }
        }

        reflex grow {
            set size value: size + 0.1;
        }

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

First Steps with GAMA 1.4

Open a model file and run a simulation

The purpose of this section is to show you how to launch GAMA, open and "play" with a simple GAMA model and create a first empty model. First of all, please run the GAMA platform.

- At the first run (and eventually at each run if you do not check the box), you will have to choose a workspace. A workspace is a directory in which you will store your models

[GAMA_UI_1] < http://gama-platform.googlecode.com/files/firstSteps14_1.png>

- This is the graphical user interface of GAMA after starting up:

[GAMA_UI_2] < http://gama-platform.googlecode.com/files/firstSteps14_2.png>

- After having closed this first tab (by clicking on the top-left cross), you get the main GAMA interface:

[GAMA_UI_3] < http://gama-platform.googlecode.com/files/firstSteps14_3v2.png>

- In GAMA, model files are organised in 3 libraries:
  - Models library: it contains model example provided with GAMA.
  - Shared models: this library is linked to an SVN repository one which registrered users can share their projects with the GAMA community.
  - User models: this library will contain the models developed by the user and stored locally.
- In each library, models areorganized in projects. A project directory contains generally various subdirectory to store the GAML model files (in **models** or **includes** for submodels), the documentation of the model (in **doc**) and additional ressources necessary for the model (e.g. images in **images**).
- We will begin by displaying a first model file. We choose the StupidModel16.gaml in the project named stupid . We can display it in the central tab by double-clicking on it in the left tab. We can also notice that once the model has been opened, an outline of the various elementsof the model are displayed in the right tab.

[GAMA_UI_4] < http://gama-platform.googlecode.com/files/firstSteps14_4.png>

- We will now run the model. To this purpose, we right-click on the central tab. The contextual menu contains the "Available experiments for [StupidModel16] ". We choose "Load 'default'".

[GAMA_UI_5] < http://gama-platform.googlecode.com/svn/wiki/images/experiment1.png>

- An alternative way to load the same experiment is to click on the button labelled "Load experiment…", which provides the same menu.

[GAMA_UI_5bis] < http://gama-platform.googlecode.com/svn/wiki/images/experiment2.png>

- We can notice that the interface has been significantly modified. We said that we have changed the perspective. We are now in the simulation perspective designed to play and observe simulation.

[GAMA_UI_6] < http://gama-platform.googlecode.com/files/firstSteps14_6.png>

- The interface is also fully based on tab that can be moved, resized, openend and closed during the simulation. In the central tab, we often have the display of the environment. We also have a tab dedicated to the simulation parameters that can be changed to explore the simulation.

[GAMA_UI_8] < http://gama-platform.googlecode.com/files/firstSteps14_8.png>

- To run/pause the simulation, you have to click on the “Run/Pause” button. We can also choose to run it step by step with the Step button. To relaod the model after having changed the parameters, we can use the Reload button.

[GAMA_UI_7] < http://gama-platform.googlecode.com/files/firstSteps14_7v3.png>

- We can come back to the modeling perspective (the interface to display and modify the model files) using the change perspective button or using the shortcut F6.

[GAMA_UI_9] < http://gama-platform.googlecode.com/files/firstSteps14_9.png>

- We then comeback to the interface used to open the StupidModel_16.gaml .

Create a first project and model file

- In the modeling perspective, we will create a first project in the User Models library. We thus right-click on this library and choose **New** .

[GAMA_UI_10] < http://gama-platform.googlecode.com/files/firstSteps14_10.png>

- First we have to create a new project.

[GAMA_UI_11] < http://gama-platform.googlecode.com/files/firstSteps14_11.png>

- We choose to name it First Project . GAMA will produce several repositories in the project to storethevarious kinds of files (GAML models, documentation, ressources...).

[GAMA_UI_12] < http://gama-platform.googlecode.com/files/firstSteps14_12.png>

- In this project, we will create a first GAMA model. We thus right-click **on the project** , choose New . We choose Model file (in GAMA) and name it First Model . We thus get the following interface, with a new and almost empty model file in the central tab.

[GAMA_UI_13] < http://gama-platform.googlecode.com/files/firstSteps14_13.png>

- We are now ready to write our first GAMA model!

Next steps...

This was only the very first steps on the long and fascinating way to master GAMA. You can continue our trip by reading the other sections of the GAMA 1.4 documentation.

- We advice you to begin by the [Tutorials14 Tutorials] section to write your first models.
- Do not hesitate to have a look to the [ModelingGuide14 modeling guide] that describes exhaustively the GAML language.
- To explore deeply the possibilities of GAMA, you can have a look to the [InterfaceGuide14 Interface Guide] .

After having mastered the basis of GAMA and its language GAML, you could need to implement your own operators and agent skills or desire to take part to the development of GAMA. The Developing Guide will help you to install the development environment and the source code of GAMA.

Convert GAML 1.3 to 1.4

Introduction

One of the big change in this new version is the language, which is a Domain-specific language (DSL).

The new GAML (GIS & Agent-based Modeling Language) is based on Xtext , an Eclipse Textual Modeling framework.

Video

Some changes

Compare to XML, the Xtext-based GAML is relying to its own specific grammar ,
so there is a set of rules that were not necessarily in XML. For instance, the unary keywords have disappeared, you should now write them as function, ie: length myList to length(myList)

and to cast, you should use instead of myGrid location the following location as myGrid

Interface Guide

The Batch perspective

Introduction

Add your content here.

Details

Add your content here. Format your content with:

- Text in **bold** or *italic*
- Headings, paragraphs, and lists
- Automatic links to other wiki pages

Modeling Guide

Model sections

The XML files that compose a model are structured in several sections.

include

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

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

global

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

```
global {
    var numberBugs type: int init: 100 parameter: 'true';
    var globalMaxConsumption type: float value: 1 parameter: 'true';
    var globalMaxFoodProdRate type: float value: 0.01 parameter: 'true';
    init {
        create species: bug number: numberBugs;
    }
}
```

entities

Definitions of species are placed int this section. Example:

```
entities {
    species bug skills: situated {
        var evol type: float init: 1;
```

```
        var color type: rgb value: rgb [255, 255/evol, 255/evol];
        var maxConsumption type: float value: globalMaxConsumption;
        var myPlace type: stupid_grid value: location as stupid_grid;
        reflex basic_move {
            let destination var: destination value: one_of ((myPlace
neighbours_at 4) where empty(each.agents));
            if condition: destination != nil {
                set location value: destination;
            }
        }
        reflex grow {
            let transfer var: transfer value: min [maxConsumption,
myPlace.food];
            set evol value: evol + transfer;
            set myPlace.food value: myPlace.food - transfer;
        }
        aspect basic {
            draw shape: circle color: color size: 1;
        }
    }
}
```

environment

This section contains definitions of environment. Actually, GAMA supports two kinds of environments: GRID based environment and continuous environment. A model can have several GRID environments and one continuous environment. GAMA platform is responsible for assuring the all the necessary synchronizations between these environments. Example:

```
environment width: 100 height: 100 {
    grid stupid_grid width: 100 height: 100 torus: true {
        var color type: rgb init: rgb('black');
        var maxFoodProdRate type: float value: globalMaxFoodProdRate;
        var maxConsumption type: float value: globalMaxConsumption;
        var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
        var food type: float init: 0.0 value: food + foodProd;
    }
}
```

output

This section defines outputs to display in the simulation mode of GAMA. Several kinds of output are supported.

- display: defines views to display grids, species, charts, texts and images. For each element, it is possible to define a size (defined by a *point*) and a position (also defined by a point). Example of display:

```
display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
}
```

- chart: defines a view to display a chart. There are three types of chart: pie, series and histogram.
  - Pie: defines a view to display the pie-based chart.
  - Series: defines a view to display the series-based chart.
  - Histogram: defines a view to display a histogram-based chart.
- inspect: inspectors the species and agents defined in the model.
  - agent inspect: user clicks on the Grid display or Gis display to select an agent. This view will display all the attributes of the selected agents as well as the change of these attributes over the simulation.
  - species inspect: this inspector displays all the species, the corresponding agents and attributes dedined in the model in a tree-based structure.
- monitor: a monitor is used to observe the change in value of an expression over the simulation.

Example:

```
output {
    display stupid_display {
        grid stupid_grid;
        species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
        chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
            data name: "[0;10]" value: bugs count (each.evol < 10);
            data name: "[10;20]" value: bugs count ((each.evol > 10) and
(each.evol < 20));
            data name: "[20;30]" value: bugs count ((each.evol > 20) and
(each.evol < 30));
            data name: "[30;40]" value: bugs count ((each.evol > 30) and
(each.evol < 40));
            data name: "[40;50]" value: bugs count ((each.evol > 40) and
(each.evol < 50));
            data name: "[50;60]" value: bugs count ((each.evol > 50) and
(each.evol < 60));
```

```
            data name: "[60;70]" value: bugs count ((each.evol > 60) and
(each.evol < 70));
            data name: "[70;80]" value: bugs count ((each.evol > 70) and
(each.evol < 80));
            data name: "[80;90]" value: bugs count ((each.evol > 80) and
(each.evol < 90));
            data name: "[90;100]" value: bugs count ((each.evol > 90) and
(each.evol < 100));
        }
    }
}
```

batch

Please see [Batch this section] for a detailed description of the batch mode.

Species Definition

Introduction

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

Species declaration

The simplest way to declare a species is the following:

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

for example:

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

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

```
let foo_agents type: list of: foo value:[];
create species: foo number: 10 return: foo_agents;
ask target:foo_agents {
    do action: write {
      arg message value: 'my name is: ' + name;
    }
}
```

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

```
species foo {
```

```
    var energy type: float init: rnd (100) min: 0 max: 100 value: energy -
0.001;
}
```

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

```
species foo {
    var energy type: float init: rnd (100) min: 0 max: 100 value: energy -
0.001;
    action eat {
        set energy value: energy + rnd (2);
    }
    action steal {
        let another_agent type:foo value: any ((foo as list) - self);
        if condition: (another_agent != nil) and ((another_agent.energy) > 0){
          set another_agent.energy value: another_agent.energy - 0.01;
          set energy value: energy + 0.01;
        }
    }
}
```

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

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

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

```
species foo {
    var energy type: float init: rnd (100) min: 0 max: 100 value: energy -
0.001;
    action eat {
        set energy value: energy + rnd (2);
    }
```

```
    action steal {
        let another_agent type:foo value: any ((foo as list) - self);
        if condition: (another_agent != nil) and ((another_agent.energy) > 0){
          set another_agent.energy value: another_agent.energy - 0.01;
          set energy value: energy + 0.01;
        }
    }
    reflex {
        if condition:flip (0.1) {
          do action:eat;
        }
        if condition: flip (0.1) {
          do action: steal;
        }
    }
}
```

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

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

they behave by their own.

Skills: behavioral plug-ins

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

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

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

Aspects: display properties

The aspect section allows to define the display of of the agents. it is possible to define different displays (i.e. different aspect sections) for a same species. In this context, the user will be able to change the display drawn during the simulation execution. The

command **draw** allows to draw a shape (line, circle or square), a icon, a text or the agent geometry. this command has several facets:

- shape: optional, can be either "line", "circle", "square" or "geometry (in this case, the geometry of the agent will be drawn).
- geometry: any arbitrary geometry, that will only be affected by the color facet.
- text: string, optional, the text to draw.
- image: string, optional, path of the icon to draw (JPEG, PNG, GIF).
- color: rgb, optional, the color to use to display the shape/text/icon/geometry.
- size: float, size of the shape/text/icon (not use in the context of the drawing of a geometry).
- at: point, location where the shape/text/icon is drawn (not use in the context of the drawing of a geometry).
- to: point, terminal location of the line (only use in the in the context of the drawing of a line).
- rotate: int, orientation of the shape/text/icon (not use in the context of the drawing of a geometry).

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

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

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

Parent: inheritance of species

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

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

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

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

Scheduling description

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

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

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

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

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

Topology description

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

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

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

Nesting species

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

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

- "A" and "B" are micro-species of "world" species.

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

Control: behavioral architecture

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

Base: Java foundation

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

Behaviors

Common behaviors

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

reflex

Attributes

- : a boolean expression

Definition

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

```
reflex { //Executed every time step
    do action: write {
    arg message value: 'Executing the unconditional reflex';
    }
}
```

```
reflex when: flip (0.5){ //Only executed when flip returns true
    do action: write {
    arg message value: 'Executing the conditional reflex';
    }
}
```

init

Attributes

- when: a boolean expression

Definition

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

EMF-based behaviors

EMF (Etho Modeling Framework) is task based behavior model. Species can define any number of tasks within their body. At any given time, only one task having the highest priority value is executed.

task

Sub elements

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

- priority: Mandatory. The activation level of the task.
- durationMin: Optional. The minimal duration of the task. Until it is reached, the task cannot be interrupted.
- durationMax: Optional. The maximal duration of the task. Beyond this duration, the task is automatically halted.
- whileCondition: Optional. The "while" watchdog : a condition tested every step to ensure that the task can still be executed. If false, the task is stopped (except if the min duration is not reached)
- untilCondition: Optional. The "until" watchdog, a condition tested every step to ensure that the task can still be executed. If true, the task is stopped (except if the min duration is not reached)

- endAction: Optional. The action to execute in case of failure/interruption of the task. No repeat is allowed.

Definition

Like reflex, a task is a sequence of commands that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler choses a task to execute basing on its current priority value. For an example of task, please have a look the definition of "foodCarrier" species in this Task exemple.

FSM-based behaviors

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

state

Attributes

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

Sub elements

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

Definition

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

Types

Table of Contents

Primitive built-in types

bool

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

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

Top of the page

float

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

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

Top of the page

int

- **Definition:** primitive datatype holding integer values comprised between -231 and 231 - 1

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

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

Top of the page

string

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

Top of the page

Complex built-in types

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

```
let nom_var type: complex_type init: init_var;
let attr_var type: type_attr value: nom_var.attr_name;
```

For example:

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

agent

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

Top of the page

container

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

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

Top of the page

file

- **Definition:** a datatype that represents a file.
- **Built-in attributes:**
  - name (type = string): the name of the represented file (with its extension)
  - extension(type = string): the extension of the file
  - path (type = string): the absolute path of the file
  - readable (type = bool, read-only): a flag expressing whether the file is readable
  - writable (type = bool, read-only): a flag expressing whether the file is writable
  - exists (type = bool, read-only): a flag expressing whether the file exists
  - is_folder (type = bool, read-only): a flag expressing whether the file is folder
  - contents (type = container): a container storing the content of the file
- **Comments:** a variable with the file type can handle any kind of file (text, image or shape files...). The type of the content attribute will depend on the kind of file. Note that the allowed kinds of file are the followings:

    - text files: files with the extensions .txt, .data, .csv, .text, .tsv, .asc. The content is by default a list of string.
    - image files: files with the extensions .pgm, .tif, .tiff, .jpg, .jpeg, .png, .gif, .pict, .bmp. The content is by default a matrix of int.
    - shapefiles: files with the extension .shp. The content is by default a list of geometry.
    - properties files: files with the extension .properties. The content is by default a map of string::string .
    - folders. The content is by default a list of string.
- **Remark:** Files are also a particular kind of container and can thus be read, written or iterated using the container operators and commands.
- **See also:** [Operators_14 File operators]
- **Declaration:** a file can be created using the generic file (that opens a file in read only mode and tries to determine its contents), folder or the new_folder (to open an existing folder or create a new one) unary operators. But things can be specialized with the combination of the read / write and image / text / shapefile / properties unary operators.

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

Top of the page

geometry

- **Definition:** a datatype that represents a vector geometry, i.e. a list of georeferenced points.
- **Built-in attributes:**
    - location (type = point): the centroid of the geometry
    - area (type = float): the area of the geometry
    - perimeter (type = float): the perimeter of the geometry
    - holes (type = list of geometry): the list of the hole inside the given geometry
    - contour (type = geometry): the exterior ring of the given geometry and of his holes
    - envelope (type = geometry): the geometry bounding box
    - width (type = float): the width of the bounding box

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

```
var varGeom type: geometry init: circle(5);
var polygonGeom type: geometry init: polygon([{3,5}, {5,6},{1,4}]);
```

Top of the page

graph

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

the minimum spanning tree, otherwise it computes the minimum spanning forest.

- **See also:** [Operators_14 Graph operators]
- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create species: road from: shape_file_road;
let the_graph type: graph value: as_edge_graph(list(road));
```

Top of the page

list

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

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

Top of the page

map

- **Definition:** a composite datatype holding an ordered collection of pairs (a key, and its associated value).
- **Built-in attributes:**
  - keys (type = list): the list of all keys
  - values (type = list): the list of all values

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

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

Top of the page

matrix

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

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

    -> c11;c21
      c12;c22
      c13;c23
```

Top of the page

pair

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

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

Top of the page

path

- **Definition:** a datatype representing a path (i.e. a polyline) linking two agents or geometries in a graph or more generally two points
- **Built-in attributes:**
  - source (type = point): the source point, i.e. the first point of the path
  - target (type = point): the target point, i.e. the last point of the path
  - graph (type = graph): the current topology (in the case it is a spatial graph), null otherwise
  - segments (type = list of geometry): the list of the geometries composing the path
- **Comments:** the path created between two agents/geometries or locations will strongly depend on the topology in which it is created.
- **Remark:** paths are particular cases of geometries. Thus they have also all the built-in attributes of the geometry datatype and can be used with every kind of operator or command admitting geometry.
- **Remark:** a path is **immutable** , i.e. it can not be modified after it is created.
- **Declaration:** paths are very barely defined litterally. We can nevertheless use the path unary operator on a list of points to build a path. Operators dedicated to the

computation of paths (such as path_to or path_between) are often used to build a path.

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

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

Top of the page

point

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

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

Top of the page

rgb

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

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

Top of the page

species

- Definition: a generic datatype that represents a species
- **Built-in attributes:**
  - topology (type=topology): the topology is which lives the population of agents
- Comments: this datatype is actually a "meta-type". It allows to manipulate (in a rather limited fashion, however) the species themselves as any other values.

- Litteral declaration: the name of a declared species is already a litteral declaration of species.
- Other declarations: the species casting operator, or its variant called species_of can be applied to an agent in order to get its species.

Top of the page

Species names as types

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

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

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

```
global {
    init {
        create species: human number: 10;
        ask target:list (human) {
              set friend value:one_of (list (human) - self);
        }
    }
}
entities {
    species human {
        var friend type:human init: nil;
    }
}
```

Top of the page

topology

- **Definition:** a topology is basically on neighbourhoods, distance,... structures in which agents evolves. It is the environment or the context in which all these values are computed. It also provides the access to the spatial index shared by all the

agents. And it maintains a (eventually dynamic) link with the 'environment' which is a geometrical border.

- **Built-in attributes:**
  - places(type = container): the collection of places (geometry) defined by this topology.
  - environment(type = geometry): the environment of this topology (i.e. the geometry that defines its boundaries)
- **Comments:** the attributes places depends on the kind of the considered topolopy. For continuous topologies, it is a list with their environment. For discrete topologies, it can be any of the container supporting the inclusion of geometries (list, graph, map, matrix)
- **Remark:** There exist various kinds of topology: continous topology and discrete topology (e.g. grid, graph...)
- **See also:** [Operators_14 Topology operators]
- **Declaration:** To create a topology, we can use the topology unary casting operator applied to:
  - an agent: returns a continuous topology built from the agent's geometry
  - a species name: returns the topology defined for this species population
  - a geometry: returns a continuous topology built on this geometry
  - a geometry container (list, map, shapefile): returns an half-discrete (with corresponding places), half-continuous topology (to compute distances...)
  - a geometry matrix (i.e. a grid): returns a grid topology which computes specifically neighbourhood and distances
  - a geometry graph: returns a graph topology which computes specifically neighbourhood and distances

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

Defining custom types

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

```
species bottle {
    var volume type: float init: 0.0 max:1 min:0.0;
```

```
    var is_empty type: bool value:volume = 0.0;
    action fill {
        set volume value: 1;
    }
}
```

How to use this species to declare new bottles :

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

Top of the page

Commands

Table of Contents

General syntax

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

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

If the command encloses other commands, they are declared between curly brakets, as in:

```
command_keyword1 expression1 attribute2: expression2... {
    command_keyword2 expression1 attribute2: expression2...;
    command_keyword3 expression1 attribute2: expression2...;
}
```

add

Attributes

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

Definition

Allows the agent to add, i.e. to insert, a new element in a container (a list, matrix, map, ...). The new element can be added either at the end of the container or at a

particular position. **Incorrect use:** The addition of a new element at a position out of the bounds of the container will produce a warning and let the container unmodified.

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

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

- Case of a **list**

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

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

- Case of a **matrix**

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

- Case of a **map**

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

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

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

```

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

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

```

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

Top of the page

ask

Attributes

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

Definition

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

```
ask receiver_agent(s) {
    [commands]
}
Or
ask target: receiver_agent(s) {
    [commands]
}
```

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

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

Alternative forms for this casting are :

- if there is only a single receiver agent:

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

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

```
ask receiver_agents of_species species_name {
    [commands]
```

```
}
```

Any command can be declared in the block commands . All the commands 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 commands , it has to use the keyword *myself* .

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

Notice

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

capture

Attributes

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

- **as** : the species that the captured agent(s) will become, this is a micro-species of the calling agent's species.
- returns: a list of the newly captured agent(s).

Definition

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

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

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

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

Top of the page

create

Attributes

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

Definition

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

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

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

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

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

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

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

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

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

Two types of discretization exist:

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

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

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

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

The agents created are initialized following the rules of their species. If one wants to refer to them after the command 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 command creates 0 to 4 agents of the same species as the sender, and puts them in the temporary variable children for later use.

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

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

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

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

Top of the page

do

Attributes

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

Enclosed tags

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

Definition

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

```
do name_of_action_or_primitive;
or
do action: name_of_action_or_primitive;
```

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

```
let result value: self name_of_action_or_primitive [];
or
let result value: name_of_action_or_primitive(self, []);
```

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

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

which is equivalent to:

```
do name_of_action_or_primitive {
    arg arg1 value: expression1;
    arg arg2 value: expression2;
    ...
}
```

or to:

```
do name_of_action_or_primitive with: [arg1::expression1, arg2::expression2];
```

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

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

Top of the page

if

Attributes

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

Following tags

- else : encloses alternative commands

Definition

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

```
if bool_expr {
    [commands]
}
or
if condition: bool_expr {
    [commands]
}
```

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

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

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

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

```
else {
        [commands]
    }
```

Top of the page

let

Attributes

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

Definition

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

```
let temp_var1 type: a_datatype value: an_expression;
```

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

loop

Attributes

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

- step: an int expression

Definition

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

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

Or:

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

Or:

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

Or:

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

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

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

The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to attributes take a a integer expression as arguments, with

the first (resp. the last) specifying the beginning (resp. end) of the interval. The step is assumed equal to 1.

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

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

put

Attributes

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

Definition

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

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

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

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

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

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

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

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

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

Top of the page

release

Attributes

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

Definition

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

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

```
  }
  species D {
  }
}
```

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

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

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

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

remove

Attributes

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

Definition

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

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

- Case of **list** .

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

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

- Case of **matrix**

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

- In the case of **map**

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

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

Top of the page

return

Attributes

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

Definition

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

```
action foo {
    return 'foo';
}
...
reflex {
```

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

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

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

Top of the page

save

Attributes

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

Definition

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

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

Top of the page

set

Attributes

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

Definition

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

```
set my_var value: expression;
set temp_var value:expression;
set global_var value:expression;
```

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

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

Top of the page

switch

Attributes

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

Embedded tags

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

Definition

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

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

Example:

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

Top of the page

Operators

Table of Contents

Definition

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

```
unary_operator (operand)
```

Most of binary operators can use two notations:

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

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

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

```
op1 ? op2 : op3
```

In addition to performing operations, operators are functional, i.e. they return a value. The return value and its type depend on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers - the result of the arithmetic operation. Moreover, operators are strictly functional, i.e. they have no side effects on their operands. For instance, the shuffle operator, which randomizes the positions of

elements in a list, does not modify its list operand but returns a new shuffled list. Top of the page

Operators by categories

Boolean operators

- ! , not
- and , or

Casting operators

- agent or agent name , bool , container , file , float , geometry , graph , int , list , map , matrix , pair , point , rgb , sort , sort_by , species , species_of , string , topology , to_gaml , to_java , unknown
- is , as , as_map , as_matrix

Comparison operators

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

Container operators (list, map, matrix)

- any , collate , empty , first , last , length , max , min , mul , one_of , product , remove_duplicates , reverse , sum ,
- @ , at , among , contains , contains_all , contains_any , copy_between , copy , count , first_with , group_by , in , index_of , inter , last_index_of , last_with , max_of , of_generic_species , of_species , min_of , select , with_max_of , with_min_of , where

File-related operators

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

Graph-related operators

- as_edge_graph , directed , edges , path , undirected , vertices

List-related operators

- accumulate list shuffle at + - collect with_max_of with_min_of of_species starts_with where

Mathematics operators

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

Matrix-related operators

- columns_list , rows_list

Random operators

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

String-related operators

- empty , first , is_number , last , length , reverse
- max min shuffle at + - among collect copy_between count first_with last_with in index_of last_index_of max_of min_of with_max_of with_min_of sort_by starts_with where select > < > = < = tokenize

Spatial operators

- Creation:
  - cricle , cone , line , link norm , point , polygon , polyline , rectangle , square , triangle
  - around
- Operators:

    - union
    - + , - , inter , intersection , masked_by , split_at , union
- Properties:
    - <-> , crosses , disjoint_from , intersects , overlaps , partially_overlaps , touches
- Transformations
    - clean , convex_hull , solid , #split_lines , skeletonize , triangulate , #without_holes
    - * , + , - , as_4_grid , as_grid , as_matrix , at_location , buffer , enlarged_by , reduced_by , rotated_by , scaled_by , simplification , transformed_by , translated_by , translated_to
- Queries:
    - agent_closest_to , agents_inside , agents_overlapping
    - closest_point_to , towards , distance_to , neighbours_at
- Points:
    - any_point_in , any_location_in

Statistical operators

frequency_of , geometric_mean , harmonic_mean , mean , mean_deviation , median , standard_deviation , variance

System

- copy , dead , eval_gaml, eval_java, every

Top of the page

Unary operators

Unary operators take one and only one operand. The syntax is : operator (operand). The priority between operators is determined from right to left (i.e. in op1 op2 op3 operand, op3 is evaluated before op2 and op1).

!

- Operand: a boolean expression.
- Result: opposite boolean value.
- Return type: bool.
- Special cases: if the parameter is not boolean, it is casted to a boolean value.

- see also: bool

```
! (true) # false.
```

Top of the page

- (unary)

- Operand: an int or a float.
- Result: opposite int or float value.
- Return type: int or float (depending on the type of the operand).

```
- (-56) # 56.
```

Top of the page

acos

- Operand: an int or a float.
- Result: the arccos of the operand (which has to be expressed in decimal degrees).
- Return type: float.
- see also: asin , atan .

```
acos (90) # 0.
```

Top of the page

abs

- Operand: an int or a float.
- Result: the absolute value of the operand.
- Return type: a positive int or float depending on the type of the operand.

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

Top of the page

agent (or species name)

- Operand: an int, agent, point or string.
- Result: casting of the operand to an agent (if **species name** is used, casting to an instance of **species name**).

- Return type: agent (or **species name**).
- Special cases:
  - if the operand is a point, returns the closest agent (resp. closest instance of **species name**) to that point (computed in the topology of the calling agent);
  - if the operand is a string, returns the agent (resp. instance of **species name**) with this name;
  - if the operand is an agent, returns the agent (resp. tries to cast this agent to **species name** and returns nil if the agent is instance of another species)
  - if the operand is an int, returns the agent (resp. instance of **species name**) with this unique index;
- see also: of_species , species

Top of the page

agent_closest_to

- Operand: a geometry or an agent.
- Result: the closest agent to the operand
- Return type: agent.
- 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.

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

Top of the page

agents_inside

- Operand: a geometry, an agent, a species, a list of geometries, a list of agents.
- Result: the list of agents covered by the operand (casted as a geometry)
- Return type: list of agents.
- see also: agents_overlapping , geometry

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

Top of the page

agents_overlapping

- Operand: a geometry, an agent, a species, a list of geometries, a list of agents.

- Result: the list of agents intersecting the operand (casted as a geometry)
- Return type: list of agents.
- see also: agents_inside , geometry

```
agents_overlapping(self) # return the agents that intersect the shape of the
agent applying the operator.
```

Top of the page

any

same signification as one_of operator. Top of the page

any_location_in

- Operand: a geometry.
- Result: a point of this geometry chosen randomly.
- Return type: point.
- see also: any_point_in

```
let my_geometry type: geometry value: square(5);
let one_point type: point value: any_location_in([my_geometry]) # one_point
will be a point of my_geometry, for example : {3,4.6}
```

Top of the page

any_point_in

same signification as any_location_in operator. Top of the page

as_edge_graph

- Operand: a list or a map
- Result: creates a graph from the list/map of edges given as operand
- Return type: a graph
- Special cases:
  - if the operand is a list, the graph will be built with elements of the list as vertices
  - if the operand is a map, the graph will be built by creating edges from pairs of the map
- see also: as_intersection_graph , as_distance_graph

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

Top of the page

asin

- Operand: an int or a float.
- Result: the arcsin of the operand (which has to be expressed in decimal degrees).
- Return type: float.
- see also: acos , atan .

```
asin (90) # 1.
```

Top of the page

atan

- Operand: an int or a float.
- Result: the arctan of the operand (which has to be expressed in decimal degrees).
- Return type: float.
- see also: acos , asin .

```
atan (45) # 1.
```

Top of the page

binomial

- Operand: a point {int,float}.
- Result: a value from a random variable following a binomial distribution (with the number of experiments n in the left member as the operand and the success probability p in the right one).
- Return type: int.
- Comment: The binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p, cf. Binomial distribution on Wikipedia .
- see also: poisson , gauss

```
poisson({15,0.6}) # a random positive integer
```

Top of the page

bool

- Operand: any type.
- Result: casting of the operand to a boolean value.
- Return type: bool.
- Special cases:
  - if the operand is null, returns false
  - if the operand is an agent, returns true if the agent is not dead
  - if the operand is an int or a float, returns true if it is not equal to 0 (or 0.0).
  - if the operand is a file, bool is formally equivalent to exists .
  - if the operand is a container, bool is formally equivalent to not empty (a la Lisp).
  - if the operand is a string, returns true is the operand is "true"
  - Otherwise, returns false.

```
bool (0)         # false;
bool(-0.4)       # true;
bool ([1, 2, 3]) # true;
```

Top of the page

circle

- Operand: a float
- Result: building of a circle geometry which radius is equal to the operand
- Return type: geometry
- Comment: the centre of the circle is by default the location of the current in which has been called this operator
- Special cases: returns a point if the operand is lower or equal to 0

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

Top of the page

clean

- Operand: a geometry
- Result: removes pinch-offs, merges overlapping polygons
- Return type: geometry

```
clean(my_geometry) # returns a cleansed version of my_geometry
```

Top of the page

closest_point_to

- Operand: an agent or a point.
- Result: the point, closest to the operand, that can be reached by the agent.
- Return type: point.
- Special cases: returns nil if the argument cannot be reached by the agent, or if the agent is neither localized nor able to move. Uses the path finder if the environment is a GIS.
- see also: next_point_towards, towards.

```
closest_point_to ({0,0}) # returns the point, closest to the origin, that can
be reached by the agent.
```

Top of the page

collate

- Operand: a list
- Result: a new list containing interleaved elements of the operand
- Return type: list
- Comment: the operand list should be a list of list of elements. The result is a list of elements.

```
collate([["el11","el12","el13"],["el21","el22","el23"],
["el31","el32","el33"]])
        # ["el11","el21","el31","el12","el22","el32","el13","el23","el33"]
```

Top of the page

columns_list

- Operand: a matrix
- Result: returns a list of the columns of the matrix, with each column as a list of elements
- Return type: list
- see also rows_list

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

Top of the page

cone

- Operand: a point
- Result: building of a cone geometry which min and max angles are given by the operand
- Return type: geometry
- Comment: the centre of the cone is by default the location of the current in which this operator has been called
- Special cases: returns nil if the operand is nil

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

Top of the page

container

- Operand: any type
- Result: casting of the operand to a container
- Return type: container
- Special cases:
  - if the operand is a container, returns itself
  - otherwise, returns the operand casted to a list
- see also: list

```
container ("string") # [string]
container (45.6)     # [45.6]
```

Top of the page

convex_hull

- Operand: a geometry
- Result: convex hull of the operand
- Return type: geometry

```
convex_hull(my_geometry) # returns the convex hull of my_geometry
```

Top of the page

copy

- Operand: any
- Result: a copy of the operand
- Return type: operand type

Top of the page

cos

- Operand: an int or a float.
- Result: the cosinus of the operand (in decimal degrees).
- Return type: float.
- Special cases: the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- see also: sin , tan .

```
cos (0) # 1.
```

Top of the page

dead

- Operand: an agent.
- Result: true if the agent is dead, false otherwise.
- Return type: bool.

```
dead(agent_A) # true or false
```

Top of the page

directed

- Operand: a graph
- Result: the operand graph becomes a directed graph
- Return type: graph
- Comment: the operator alters the operand graph, it does not create a new one.
- see also: undirected

Top of the page

edges

- Operand: a path.
- Result: return the list of the edges in the path
- Return type: list.
- Special case: if the operand is null, edges returns an empty list
- see also: path , vertices

```
edges(path([{1,5},{12,45},{34,56}])) # returns the list of the two polylines
composing this path
```

Top of the page

even

- Operand: an int .
- Result: true if the operand is even and false if it is odd.
- Return type: a boolean.
- Special cases: if the operand is equal to 0, it returns true .

```
even (3) # false.
even (-12) # true.
```

Top of the page

every

- Operand: an int.
- Result: true every operand time step, false otherwise
- Return type: a boolean.
- Comment: the value of the every operator depends deeply on the time step. It can be used to do something not every step.

```
reflex text_every {
    if condition: every(2) {
        do action: write with: [message::"the time step is even"];
    else {
        do action: write with: [message::"the time step is odd"];
    }
    }
}
```

Top of the page

empty

- Operand: a container or a string.
- Result: true if the operand is empty, false otherwise.
- Return type: bool.
- Special cases: the empty operator behavior depends on the nature of the operand,
  - if it is a color or a point, empty returns the false
  - if it is a list, empty returns true if there is no element in the list, and false otherwise
  - if it is a map, empty returns true if the map contains no key-value mappings, and false otherwise
  - if it is a pair, empty returns true if both the key and the value are null, 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 graph, empty returns true if it contains no vertex and no edge, and false otherwise
  - if it is a matrix,
    - for a matrix of int, float or object, it will return true if all elements are respectively 0, 0.0 or null, and false otherwise
    - for a matrix of geometry, it will return true if the matrix contains no cell, and false otherwise
  - if it is a string, empty returns true if the string does not contain any character, and false otherwise

```
empty ([]) # true;
empty ('abced') # false;
```

Top of the page

exp

- Operand: an int or a float.
- Result: returns Euler's number e raised to the power of the operand.
- Return type: float.
- Special cases: the operand is casted to a float before being evaluated.
- see also: ln .

```
exp (0) # 1.
```

Top of the page

fact

- Operand: an int .
- Result: the factorial of the operand.
- Return type: an int.
- Special cases: if the operand is less than 0, fact returns 0.

```
fact (4) # 24.
```

Top of the page

file

- Operand: a string.
- Result: opens a file in read only mode and tries to determine and store it in the contents attribute.
- Return type: file.
- Comment: The file should have a supported extension, cf. Types14 for supported file extensions.
- Special cases: If the specified string does not refer to an existing file, an exception is risen when the variable is used.
- see also: folder , new_folder .

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

Top of the page

first

- Operand: a container, a string or a species.
- Result: the first element of the operand
- Return type: any
- Special cases: the first operator behavior depends on the nature of the operand,
  - if it is a color, first returns the red component
  - if it is a list, first returns the first element of the list, or nil if the list is empty
  - if it is a map, first returns nil (the map do not keep track of the order of elements)
  - if it is a pair, first returns the key of the pair key::value

    - if it is a point, first returns the x-coordinate of a point
    - 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 graph, first returns the first element in the list of vertexes
    - if it is a matrix, first returns the element at {0,0} in the matrix
      - for a matrix of int or float, it will return 0 if the matrix is empty
      - for a matrix of object or geometry, it will return null if the matrix is empty
    - if it is a string, first returns a string composed of its first character.
    - if it is a species, first returns the first element of the list of the agents in the species
- see also: last .

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

Top of the page

flip

- Operand: an int or a float (between 0 and 1).
- Result: true or false given the probability represented by the operand.
- Return type: bool.
- Special cases: flip 0 always returns false, flip 1 true.
- see also: rnd .

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

Top of the page

float

- Operand: any type.
- Result: casting of the operand to a floating point value.
- Return type: float.
- Special cases:
    - if the operand is numarical value, returns its value as a floating point value
    - if the operand is a string, tries to convert its content to a floating point value;
    - if the operand is a boolean, returns 1.0 for true and 0.0 for false;
    - otherwise, returns 0.0
- see also: int .

```
float ('234.0') # 234.0;
```

```
float ({12,23}) # 0.0;
float (true) # 1.0;
float (rgb ('black')) # 0.0;
```

Top of the page

folder

- Operand: a string.
- Result: opens an existing repository
- Return type: file.
- Special cases: If the specified string does not refer to an existing repository, an exception is risen.
- see also: new_folder , file .

```
let dirT type: file value: folder("../includes/");

// dirT represents the repository "../includes/"

// dirT.contents here contains the list of the names of included files
```

Top of the page

gauss

- Operand: a point {mean, standardDeviation}.
- 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.
- Return type: float
- Special cases: when standardDeviation value is 0.0, it always returns the mean value.
- see also: truncated_gauss , poisson .

```
gauss({0,0.3}) # 0.22354;
gauss({0,0.3}) # -0.1357.
```

Top of the page

geometric_mean

- Operand: a list.
- Result: the geometric mean of the elements of the operand.

- Return type: float.
- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- Comment: see Geometric mean for more details
- see also: mean , median , harmonic_mean .

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

Top of the page

geometry

- Operand: any type.
- Result: casting of the operand to a geometry value.
- Return type: geometry .
- Special cases:
  - if the operand is a point, returns geometry composed by this point
  - if the operand is a species name, returns the union of the geometry of each element of the species
  - if the operand is pair, returns a geometry representing a link between each element of the pair casted as geometry
  - if the operand is a container,
    - if the elements of the container are all points, returns the polygon composed by these points
    - otherwise returns the union of the elements casted as geometry

Top of the page

harmonic_mean

- Operand: a list.
- Result: the harmonic mean of the elements of the operand.
- Return type: float.
- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- Comment: see Harmonic mean for more details
- see also: mean , median , geometric_mean .

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

Top of the page

image

- Operand: a string.
- Result: opens a file that is a kind of image.
- Return type: file.
- Comment: The file should have an image extension, cf. Types14 for supported file extensions.
- Special cases: If the specified string does not refer to an existing image file, an exception is risen.
- see also: file , shapefile , properties , text .

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

Top of the page

int

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

```
int ('234.3') # 234;
int ({12,23}) # 12;
int (true) # 1;
int (rgb ('black')) # 0;
```

Top of the page

intensity (deprecated)

- Operand: a string.
- Result: the intensity of the signal denoted by the operand at the agent's location.
- Return type: float.

- Special cases: returns 0.0 if the agent is not controlled by EMF, if the agent is not localized or if no grids have been defined as an environment.

```
intensity 'pheromone' # returns the intensity of the 'pheromone' signal on the
grid cell where the agent is located.
```

Top of the page

is_image

- Operand: a string.
- Result: the operator tests whether the operand represents the name of a supported image file
- Return type: bool.
- Comment: cf. Types14 for supported (espacially image) file extensions.
- see also: is_text , is_properties , is_shape .

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

Top of the page

is_number

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

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

Top of the page

is_properties

- Operand: a string.
- Result: the operator tests whether the operand represents the name of a supported properties file
- Return type: bool.
- Comment: cf. Types14 for supported (espacially properties) file extensions.
- see also: is_text , is_image , is_shape .

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

Top of the page

is_text

- Operand: a string.
- Result: the operator tests whether the operand represents the name of a supported text file
- Return type: bool.
- Comment: cf. Types14 for supported (espacially text) file extensions.
- see also: is_properties , is_image , is_shape .

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

Top of the page

is_shape

- Operand: a string.
- Result: the operator tests whether the operand represents the name of a supported shapefile
- Return type: bool.
- Comment: cf. Types14 for supported (espacially shapefile) file extensions.
- see also: is_properties , is_image , is_text .

```
is_shape("../includes/Stupid_Cell.Data")    #  false;
is_shape("../includes/test.png")            #  false;
```

```
is_shape("../includes/test.properties")     #  false;
is_shape("../includes/test.shp")            #  true;
```

Top of the page

last

- Operand: a container, a string or a species.
- Result: the last element of the operand
- Return type: any
- Special cases: the last operator behavior depends on the nature of the operand,
  - if it is a color, last returns the blue component
  - if it is a list, last returns the last element of the list, or nil if the list is empty
  - if it is a map, last returns nil (the map do not keep track of the order of elements)
  - if it is a pair, last returns the value of the pair key::value
  - if it is a point, last returns the y-coordinate of a point
  - 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 graph, last returns the last element in the list of vertexes
  - if it is a matrix, last returns the element at {length-1,length-1} in the matrix
    - for a matrix of int or float, it will return 0 if the matrix is empty
    - for a matrix of object or geometry, it will return null if the matrix is empty
  - if it is a string, last returns a string composed of its last character, or an empty string if the operand is empty
  - if it is a species, last returns the last element of the list of the agents in the species
- see also: first .

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

Top of the page

length

- Operand: a container, a string or a species.
- Result: the number of elements contained in the operand
- Return type: int.
- Special cases: the length operator behavior depends on the nature of the operand,
  - if it is a color, length always returns 3.
  - if it is a list or a map, length returns the number of elements in the list

- if it is a point or a pair, length always return 2
- if it is a graph, last returns the number of vertexes or of edges (depending on the way it was created)
- if it is a matrix, length returns the number of cells
- if it is a string, length returns the number of characters
- if it is a species, length returns the number of elements in the list of the agents in the species

```
length ('I am an agent') # 13;
length ([12,13]) # 2.
```

Top of the page

line

same signification as polyline operator. Top of the page

link

- Operand: a pair
- Result: create a link between the 2 elements of the pair
- Return type: geometry
- Comment: This operator returns a geometry representing a link between two geometries. The geometry of this link is the intersection of the two geometries when they intersect, and a line between their centroids when they do not.
- Special cases:
  - if the operand is null, link returns a point {0,0}
  - if one of the elements of the pair is a list of geometries or a species, link will consider the union of the geometries or of the geometry of each agent of the species

```
link geom1::geom2
```

Top of the page

list

- Operand: any type.
- Result: casting of the operand to a list.
- Return type: list.
- Special cases:

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

```
list ('abc') # ['a', 'b', 'c'];
list (123) # '123';
list (['a', 23]) # ['a', 23];
list (rgb ('black')) # [0, 0, 0];
```

Top of the page

ln

- Operand: an int or a float (greater than 0).
- Result: Returns the natural logarithm (base e) of the operand.
- Return type: float.
- Special cases: an exception is raised if the operand is less than zero.
- see also: exp .

```
ln(1) # 0.0
```

Top of the page

map

- Operand: any type.
- Result: casting of the operand to a map.
- Return type: map.
- Special cases:
    - if the operand is a color RRGGBB, returns a map with the three elements: "r"::RR , "g"::GG , "b"::BB

- if the operand is a point, returns a map with two elements: "x":: x -ccordinate and "y":: y -coordinate
- if the operand is pair, returns a map with this only element
- if the operand is a species name, returns the map containing all the agents of the species as a pair nom_agent::agent
- if the operand is a agent, returns a map containing all the attributes as a pair attribute_name::attribute_value
- if the operand is a list, returns a map containing either elements of the list if it is a list of pairs, or pairs list.get(i) ::list .get(i+1)
- if the operand is a file, returns the content casted to map
- if the operand is a graph, returns the a map with pairs edge_source::edge_target
- otherwise returns a map containing only the pair operand::operand .

```
map (123) # [123::123];
map (rgb ('black')) # ["g"::0, "b"::0, "r"::0];
map({4,6})  #  ["x"::4.0,"y"::6.0]
```

Top of the page

matrix

- Operand: any type.
- Result: casting of the operand to a matrix.
- Return type: matrix.
- Special cases:
  - if the operand is a string, returns a matrix of string, each each line of the matrix corresponding to a line of the string; each line is tokenized and each token is put into a cell of the matrix
  - if the operand is a color, returns a matrix containing on the first row three cells containing the three components of the color
  - if the operand is a point or a key, returns a matrix containing on the first row two cells containing the two elements of the point (coordinates) or the pair (the key and the value)
  - if the operand is a file, returns its contents as a matrix
  - if the operand is a list, returns a flat matrix with elements of the list
  - otherwise returns a matrix containing the operand.

```
matrix ('abc rer r') # [['abc', 'rer', 'r']];
matrix (123) # [[123]];
matrix (['a', 23]) # [['a', 23]];
matrix (rgb ('black')) # [[0, 0, 0]];
```

Top of the page

max

- Operand: a container.
- Result: the maximum element found in the operand.
- Return type: float, int or point (depending on the operand).
- Special cases: the max operator behavior depends on the nature of the operand,
  - if it is a color, max returns the maximum of the three components
  - if it is a list,
    - a list of int or float: max returns the maximum of all the elements
    - 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.)
    - otherwise: max transforms all elements into float 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 pair, max returns the maximum of the two elements of the pair if the first is comparable
  - if it is a point, max returns the maximum of the two coordinates
  - 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,
    - a matrix of int, float or object, max returns the maximum of all the numerical elements (thus all elements for integer and float matrices)
    - a matrix of geometry, max returns the maximum of the list of the geometries
- see also: min .

```
max ([100, 23.2, 34.5]) # 100.0.
```

Top of the page

mul

- Operand: a container.
- Result: the product of all the elements of the operand.
- Return type: float.
- Special cases: the mul operator behavior depends on the nature of the operand,
  - if it is a color, mul returns the product of the three components
  - if it is a list,
    - a list of int or float: mul returns the product of all the elements

    - 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)
    - otherwise: mul transforms all elements into float and multiplies them
  - if it is a map, mul returns the product of the value of all elements
  - if it is a pair, mul returns the product of the two elements of the pair if they are integer or float, 0.0 otherwise
  - if it is a point, mul returns the product of the two coordinates
  - 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,
    - a matrix of int, float or object, mul returns the product of all the numerical elements (thus all elements for integer and float matrices)
    - a matrix of geometry, mul returns the product of the list of the geometries
- see also: sum .

```
mul ([100, 23.2, 34.5]) # 100.0.
```

Top of the page

mean

- Operand: a container.
- Result: the mean of all the elements of the operand.
- Return type: float or point.
- Comment: if the container contains points, the result will be a point.
- Special cases: 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.
- see also: sum .

```
mean ([4.5, 3.5, 5.5, 7.0]) # 5.125
```

Top of the page

mean_deviation

- Operand: a list.
- Result: the deviation from the mean of all the elements of the operand.
- Return type: float.

- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- Comment: cf. Mean deviation for more details
- see also: mean , standard_deviation

```
mean_deviation ([4.5, 3.5, 5.5, 7.0]) # 1.125
```

Top of the page

median

- Operand: a list.
- Result: the median of all the elements of the operand.
- Return type: float.
- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- see also: mean .

```
median ([4.5, 3.5, 5.5, 7.0]) # 5.0
```

Top of the page

min

- Operand: a container.
- Result: the minimum element found in the operand.
- Return type: float, int or point (depending on the operand).
- Special cases: the min operator behavior depends on the nature of the operand,
  - if it is a color, min returns the minimum of the three components
  - if it is a list,
    - a list of int or float: min returns the minimum of all the elements
    - 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.)
    - otherwise: min transforms all elements into float 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 pair, min returns the minimum of the two elements of the pair if the first is comparable
  - if it is a point, min returns the minimum of the two coordinates

    - 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,
      - a matrix of int, float or object, min returns the minimum of all the numerical elements (thus all elements for integer and float matrices)
      - a matrix of geometry, min returns the minimum of the list of the geometries
- see also: max .

```
min ([100, 23.2, 34.5]) # 23.2.
```

Top of the page

new_folder

- Operand: a string.
- Result: opens an existing repository or create a new folder if it does not exist.
- Return type: file.
- Special cases: If the specified string does not refer to an existing repository, the repository is created. If the string refers to an existing file, an exception is risen.
- see also: folder , file .

```
let dirNewT type: file value: new_folder("../incl/");   // dirNewT represents
the repository "../incl/"
                                                        // eventually creates
the directory ../incl
```

Top of the page

norm

- Operand: point.
- Result: the norm of the vector with the coordinnates of the point operand.
- Return type: float.

```
norm({3,4})   #  5.0
```

Top of the page

not

same signification as ! operator. Top of the page

one_of

- Operand: a container or species.
- Result: a random element from the list.
- Return type: any.
- Special cases: returns nil if the list is empty.
- Comment: In the case of a species, the operand is casted to a list before the expression is evaluated. Therefore, if foo is the name of a species, any foo will return a random agent from this species (see list).

```
any ([1,2,3]) # 1, 2, or 3.
one_of ([1,2,3]) # 1, 2, or 3.
// The species `bug` has previously be defined
one_of (bug) # bug3
let mat3 type:matrix value: matrix([["c11","c12","c13"],["c21","c22","c23"]]);
one_of(mat3) # "c11","c12","c13", "c21","c22" or "c23"
```

Top of the page

pair

- Operand: any type.
- Result: casting of the operand to a pair value.
- Return type: pair.
- Special cases:
  - if the operand is null, returns null
  - if the operand is a point, returns the pair x- coordinate::y -coordinate
  - if the operand is a particular kind of geometry, a link between geometry, returns the pair formed with these two geoemtries
  - if the operand is a map, returns the pair where the first element is the list of all the keys of the map and the second element is the list of all the values of the map
  - if the operand is a list, returns a pair with the two first element of the list used to built the pair
  - Otherwise, returns the pair string(operand) ::operand

```
pair({3,5})   #  3::5
pair([6,7,8]) #  6::7
pair([2::6,5::8,12::45]) # [12,5,2]::[45,8,6]
```

Top of the page

path

- Operand: list
- Result: creates a path between all the elements of the operand
- Return type: path
- Comment: The path is created after transformation of each element of the list into the corresponding point.

```
path([{1,5},{12,45},{34,56}])  # a path from {1,5} to {12,45} through {34,56}
                    #  ([polygon ([{1.0,5.0},{12.0,45.0}]),polygon
([{12.0,45.0},{34.0,56.0}])]) as path
```

Top of the page

point

- Operand: any
- Result: casting of the operand to a point value.
- Return type: point.
- Special cases:
  - if the operand is null, returns null
  - if the operand is an agent, returns its location
  - if the operand is a geometry, returns its centroid
  - if the operand is a list with at least two elements, returns a point with the two first elements of the list (casted to float)
  - if the operand is a map, returns the point with values associated respectively with keys "x" and "y"
  - if the operand is a pair, returns a point with the two elements of the pair (casted to float)
  - otherwise, returns a point {val,val} where val is the float value of the operand

```
point([12,123.45]) # {12.0, 123.45}
point(2)           # {2.0, 2.0}
point(["t"::4,"x"::5,"y"::6])  # {5.0,6.0}
```

Top of the page

poisson

- Operand: a float.
- Result: a value from a random variable following a Poisson distribution (with the positive expected number of occurence lambda as operand).

- Return type: int.
- Comment: The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event, cf. Poisson distribution on Wikipedia .
- see also: binomial , gauss

```
poisson(3.5) # a random positive integer
```

Top of the page

polygon

- Operand: a list of points
- Result: building of a polygon geometry from the given points
- Return type: geometry
- 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

```
polygon([{0,0}, {0,10}, {10,10}, {10,0}]) # returns a polygon geometry
composed of the 4 points.
```

Top of the page

polyline

- Operand: a list of points
- Result: building of a polyline geometry from the given points
- Return type: geometry
- 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.

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

Top of the page

product

same signification as mul operator. Top of the page

properties

- Operand: a string.
- Result: opens a file that is a kind of properties.
- Return type: file.
- Comment: The file should have a properties extension, cf. Types14 for supported file extensions.
- Special cases: If the specified string does not refer to an existing propserites file, an exception is risen.
- see also: file , shapefile , image , text .

```
let fileT type: file value: properties("../includes/
testProperties.properties");  // fileT represents the properties file "../
includes/testProperties.properties"
```

Top of the page

read

- Operand: a file.
- Result: marks the file so that only read operations are allowed.
- Return type: file.
- Comment: A file is created by default in read-only mode. The operator write can change the mode.
- see also: file , write .

```
read(shapefile("../images/point_eau.shp"))  #  returns a file in read-only
mode representing "../images/point_eau.shp"
```

Top of the page

rectangle

- Operand: a point
- Result: building of a rectangle geometry which side sizes are given by the operand
- Return type: geometry
- Comment: the centre of the rectangle is by default the location of the current in which has been called this operator
- Special cases: returns a point if the operand is nil
- See also: circle , square

```
rectangle({10, 5}) # returns a geometry as a rectangle with width = 10 and
heigh = 5.
```

Top of the page

remove_duplicates

- Operand: a list
- Result: a list corresponding to the operand without the duplicate elements.
- Return type: list

```
remove_duplicates ([10,12,10,14]) # [10, 12, 14].
```

Top of the page

reverse

- Operand: a container or a string.
- Result: the operand elements in the reversed order in a copy of the operand.
- Return type: a container of the same type of the operand or a string.
- Comment: the operand is not modified.
- Special cases: the reverse operator behavior depends on the nature of the operand,
  - if it is a color (say RRGGBB), reverse returns a new color with the colors in the reversed order (BBGGRR)
  - if it is a list, reverse return a copy of the operand list with elements in the reversed order
  - if it is a map, reverse returns a copy of the operand map with each pair in the reversed order (i.e. all keys become values and values become keys)
  - if it is a pair, reverse returns a new pair in the reversed order (the old key is the new value and the old value is the new key)
  - if it is a point, reverse returns a new point with reversed coordonates
  - if it is a file, reverse returns a copy of the file with a reversed content
  - if it is a graph, reverse returns a copy of the graph (with all edges and vertexes), with all of the edges reversed
  - if it is a matrix, reverse returns a new matrix containing the transpose of the operand.
  - if it is a string, reverse returns a new string with caracters in the reversed order.

```
reverse ('abcd') # 'dcba';
reverse ({10,13}) # {13, 10};
reverse ([10,12,14]) # [14, 12, 10].
```

```
reverse ([k1::44, k2::32, k3::12])  # [12::k3,  32::k2, 44::k1]
```

Top of the page

rgb

- Operand: any type.
- Result: casting of the operand to a rgb color.
- Return type: rgb.
- Special cases:
  - if the operand is nil, returns white;
  - if the operand is a string, the allowed color names are the constants defined in the java.awt.Color class, i.e.: black, blue, cyan, darkGray, lightGray, gray, green, magenta, orange, pink, red, white, yellow. Otherwise tries to cast the string to an int and returns this color
  - if the operand is a list, the integer value associated to the three first elements of the list are used to define the three red (element 0 of the list), green (element 1 of the list) and blue (element 2 of the list) components of the color.
  - if the operand is a map, the red, green, blue compoenents take the value associated to the keys "r", "g", "b" in the map.
  - if the operand is a matrix, return the color of the matrix casted as a list
  - if the operand is a boolean, returns black for true and white for false;
  - if the operand is an integer value, the decimal integer is translated into a hexadecimal value: OxRRGGBB. The red (resp. green, blue) component of the color take the value RR (resp. GG, BB) translated in decimal.

```
rgb ('white') # white color;
rgb (#FFFFFF) # white color;
rgb ([0, 255,0]) # green color;
rgb ('255') # blue color;
```

Top of the page

rnd

- Operand: an int or a float.
- Result: a random integer in the interval [0, operand] .
- Return type: int.
- Special cases: the argument is casted to an int before being evaluated.
- see also: flip .
- Comments : to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision;

```
rnd (2) # 0, 1 or 2
rnd (1000) / 1000 # a float between 0 and 1 with a precision of 0.001
```

Top of the page

round

- Operand: an int or a float.
- Result: the rounded value of the operand.
- Return type: int.
- Special cases: the argument is casted to a float before round is evaluated.
- see also: int .

```
round (0.51) # 1;
round (100.2) # 100.
```

Top of the page

rows_list

- Operand: a matrix
- Result: returns a list of the rows of the matrix, with each row as a list of elements
- Return type: list
- see also: columns_list

```
rows_list(matrix([["el11","el12","el13"],["el21","el22","el23"],
["el31","el32","el33"]])
            #  [["el11","el21","el31"],["el12","el22","el32"],
["el13","el23","el33"]]
```

Top of the page

shapefile

- Operand: a string.
- Result: opens a file that a is a kind of shapefile.
- Return type: file.
- Comment: The file should have a shapefile extension, cf. Types14 for supported file extensions.
- Special cases: If the specified string does not refer to an existing shapefile file, an exception is risen.
- see also: file , properties , image , text .

```
let fileT type: file value: shapefile("../includes/testProperties.shp");
                // fileT represents the shapefile file "../includes/
testProperties.shp"
```

Top of the page

shuffle

- Operand: a list, string, matrix or species.
- Result: the elements of the operand in random order.
- Return type: list, string or matrix.
- Special cases: if the operand is empty, returns an empty list (or string, matrix);
- see also: reverse .

```
shuffle ([12, 13, 14]) # [14,12,13];
shuffle ('abc') # 'bac'.
shuffle ([["c11","c12","c13"],["c21","c22","c23"]]) # [["c12","c21","c11"],
["c13","c22","c23"]]
shuffle (bug) # shuffle the list of all agent of the `bug` species
```

Top of the page

sin

- Operand: an int or a float.
- Result: the sinus of the operand (in decimal degrees).
- Return type: float.
- Special cases: the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- see also: cos , tan .

```
sin (0) # 0.
```

Top of the page

skeletonize

- Operand: a geometry (polygon)
- Result: the skeleton of the geometry
- Return type: list of geometries
- Comment: A skeleton of a geometry is a thin version of that geometry that is equidistant to its boundaries, cf. Skeleton for more details about skeleton.

```
let my_geometry type: geometry value: square(5);
let skeleton type: list of: geometry value: skeletonize(my_geometry);  #
skeleton will be the skeleton of my_geometry
```

Top of the page

solid

- Operand: a geometry (polygon)
- Result: the geometry without its holes
- Return type: geometry
- see also: without_holes

```
let my_geometry type: geometry value: square(5) - square(3);
let solid_geom type: geometry value: solid(my_geometry);
                    # solid_geometry will be a rectangle of side size equal to
5 (without holes)
```

Top of the page

species

- Operand: any.
- Result: casting of the operand to a species.
- Return type: species.
- Special cases:
  - if the operand is nil, returns nil;
  - if the operand is an agent, returns its species;
  - if the operand is a string, returns the species with this name (nil if not found);
  - otherwise, returns nil

```
species (self) # the species of the agent;
species ('world') # the species named 'world';
```

Top of the page

species_of

same signification as species operator. Top of the page

split_lines

- Operand: a list of geometries (lines)
- Result: computes the resulting after cutting the lines at their intersections
- Return type: list of geometry

```
let line1 type: geometry value: line([{0,10}, {20,10}]);
let line2 type: geometry value: line([{10,0}, {10,20}]);
let lines type: list of: geometry value: split_lines([line1, line2]);
            # lines will be a list of four line geometries: line([{0,10},
{10,10}]), line([{10,10}, {20,10}]), line([{10,0}, {10,10}]) and
line([{10,10}, {10,20}]).
```

Top of the page

sqrt

- Operand: an int or a float (greater than 0).
- Result: Returns the square root of the operand.
- Return type: float.
- Special cases: an exception is raised if the operand is less than zero.

```
sqrt(4) # 2.0
```

Top of the page

square

- Operand: a float
- Result: building of a square geometry which side size is equal to the operand
- Return type: geometry
- Comment: the centre of the square is by default the location of the current in which this operator has been called
- Special cases: returns a point if the operand is lower or equal than 0

```
square(10) # returns a geometry as a square of side size 10.
```

Top of the page

standard_deviation

- Operand: a list.

- Result: the standard deviation on the elements of the operand.
- Return type: float.
- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- Comment: cf. Standard deviation for more details
- see also: mean , mean_deviation

```
standard_deviation ([4.5, 3.5, 5.5, 7.0]) # 1.2930100540985752
```

Top of the page

string

- Operand: any type.
- Result: casting of the operand to a string.
- Return type: string.
- Special cases:
  - if the operand is nil, returns 'nil';
  - if the operand is an agent, returns its name;
  - if the operand is a string, returns the operand;
  - if the operand is an int or a float, returns their string representation (as in Java);
  - if the operand is a boolean, returns 'true' or 'false';
  - if the operand is a species, returns its name;
  - if the operand is a color, returns its litteral value if it has been created with one (i.e. 'black', 'green', etc.) or the string representation of its hexadecimal value.
  - if the operand is a container, returns its string representation;

```
string (rgb ('black')) # 'black';
string (12.34) # '12.34';
string (world) # 'world' if world is a species.
```

Top of the page

sum

- Operand: a container.
- Result: the sum of all the elements of the operand.
- Return type: float.
- Special cases: the sum operator behavior depends on the nature of the operand:
  - if it is a color, sum returns the sum of the three components
  - if it is a list:
    - a list of int or float: sum returns the sum of all the elements

    - 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)
    - otherwise: 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 pair, sum returns the sum of the two elements of the pair if they are integer or float, 0.0 otherwise
  - if it is a point, sum returns the sum of the two coordinates
  - if it is a file, sum returns the sum of the content of the file (that is also a container)
  - if it is a graph, sum returns the sum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
  - if it is a matrix:
    - 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)
    - a matrix of geometry, sum returns the sum of the list of the geometries
- see also: mul .

```
sum ([12,10, 3]) # 25.0.
```

Top of the page

tan

- Operand: an int or a float.
- Result: the trigonometic tangent of the operand (expressed in decimal degrees).
- Return type: float.
- Special cases: the argument is casted to an int before being evaluated. Integers outside the range [0-359] are normalized.
- see also: sin , cos .

```
tan (180) # 0.
```

Top of the page

text

- Operand: a string.
- Result: opens a file that a is a kind of text.
- Return type: file.
- Comment: The file should have a text extension, cf. Types14 for supported file extensions.

- Special cases: If the specified string does not refer to an existing text file, an exception is risen.
- see also: file , properties , image , shapefile .

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

Top of the page

TGauss

same signification as truncated_gauss operator. Top of the page

to_java

- Operand: any
- Result: returns the GAML code associated to the operand
- Return type: string
- see also: to_gaml

Top of the page

to_gaml

- Operand: any
- Result: returns the Java code associated to the operand
- Return type: string
- see also: to_java

Top of the page

topology

- Operand: any
- Result: casting of the operand to a topology.
- Return type: topology
- Special cases:
  - if the operand is nil, returns nil
  - if the operand is a population, returns the topology of this population

    - if the operand is a species, returns the topology of the agents of this species (expression evaluated in the scope of the calling agent)
    - if the operand is a geometry, returns a continuous topology built from this geometry (with the the shape of the geometry)
    - if the operand is a container, returns a topology built from this topology
      - if the operand is a graph, returns a graph topolgy built on this graph
      - if the operand is a matrix of geometries, returns a grid topology from this matrix
      - otherwise, built a multiple topology
    - otherwise, returns the topology built from this operand casted to a geometry
- see also geometry

Top of the page

towards

- Operand: an agent or a point.
- Result: the direction (in degrees) that the agent must follow to face the operand.
- Return type: int.
- Special cases: if the agent is not localized, returns 0;
- see also: . next_point_towards

```
towards ({0, 0}) # an int between 0 and 359.
```

Top of the page

triangle

- Operand: a float
- Result: building of a triangle geometry which side size is given by the operand
- Return type: geometry
- Comment: the centre of the triangle is by default the location of the current in which this operator has been called
- Special cases: returns a point if the operand is nil

```
triangle(5) # returns a geometry as a triangle with side_size = 5.
```

Top of the page

triangulate

- Operand: a geometry (polygon)

- Result: the Delaunay triangulation of the geometry
- Return type: list of geometries
- Comments: see Delaunay Triangulation for more details

```
let my_geometry type: geometry value: square(5);
let skeleton type: list of: geometry value: triangulate(my_geometry);  #
list of triangle geometries resulting from the Delaunay triangulation of
my_geometry
```

Top of the page

truncated_gauss

- Operand: a list or a point {mean, standardDeviation}.
- Result: a random value from a normally distributed random variable in the interval]mean - standardDeviation; mean + standardDeviation[.
- Return type: float.
- Special cases: when truncated_gauss is called with an list of only one element [mean] , it will always return 0.0.
- see also: gauss .

```
truncated_gauss ({0, 0.3}) # an float between -0.3 and 0.3.
truncated_gauss ([0.5, 0.0]) # 0.5 (always).
```

Top of the page

undirected

- Operand: a graph
- Result: the operand graph becomes an undirected graph
- Return type: graph
- Comment: the operator alters the operand graph, it does not create a new one.
- see also: directed

Top of the page

unknown

- Operand: any
- Result: the operand
- Return type: any
- Comment: returns the operand without any casting

Top of the page

union

- Operand: a list of geometry or a species.
- Result: the geometry composed of the union of all the geometries of the list or of the geometries of all the agents of the species.
- Return type: geometry.
- Special cases:
  - when the operand is an empty list, it returns a null geometry
  - when the operand is a list of points and it forms a closed linestring, the corresponding polygon is built

```
union([{2,5},{7,21},{56,12},{2,5}])   # build the polygon from the 3 points
```

Top of the page

variance

- Operand: a list.
- Result: the variance of the elements of the operand.
- Return type: float.
- Special cases: The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- Comment: see Variance for more details
- see also: mean , median .

```
variance ([4.5, 3.5, 5.5, 7.0]) # 1.671875
```

Top of the page

without_holes

same signification as solid operator. Top of the page

write

- Operand: a file.
- Result: marks the file so that read and write operations are allowed.
- Return type: file.
- Comment: A file is created by default in read-only mode.

- see also: file , read .

```
write(shapefile("../images/point_eau.shp"))  #  returns a file in read-write
mode representing "../images/point_eau.shp"
```

Top of the page

Binary operators

=

- Left-hand operand: any expression.
- Right-hand operand: any expression.
- Result: true if both operands are equal, false otherwise.
- Comments: this operator will return true if the two operands are identical (i.e., the same object) or equal. Comparisons between nil values are permitted. Note that floating point and integer values are always considered as different (i.e., 0.0 is not the same as 0).
- see also: !=.

```
[1, 2, 3.0] = [1,2,3.0] # true;
int (3.0) = 3 # true;
rgb ('black') = rgb (#000000) # true;
float (1) = 1.0 # true;
```

Top of the page

!=

same signification as <> operator. Top of the page

<>

- Left-hand operand: any expression.
- Right-hand operand: any expression.
- Result: true if both operands are different, false otherwise.
- Comments: this operator will return false if the two operands are identical (i.e., the same object) or equal. Comparisons between nil values are permitted. Note that floating point and integer values are always considered as different (i.e., 0.0 is not the same as 0).
- see also: =.

```
[1, 2, 3.0] != [1,2,3.0] # false;
int (3.0) != 3 # false;
rgb ('black') != rgb (#000000) # false;
float (1) != 1.0 # false;
```

Top of the page

> >= < <=

- Left-hand operand: any expression.
- Right-hand operand: any expression.
- Result: true if the left-hand operand is respectively greater than, less than, greater than or equal to, less than or equal to the right-hand operand. false otherwise.
- Comments: these operator work with numbers (int and float) as well as strings, for which a lexicographic comparison is performed.

```
12 > 11.0 # true;
'zzz' > 'abc' # true;
int (12.0) < 12 # false;
```

Top of the page

@

same signification as **at** operator. Top of the page

+

- Left-hand operand: a list, string, matrix, point, int, float or a geometry.
- Right-hand operand: a list, string, matrix, point, int, float or a geometry.
- Result: the sum, union or concatenation of the two operands.
- Special cases:
  - if left-operand is a geometry
    - if the right-operand is a geometry, see union
    - if the right-operand is a point, if the geometry is a point, returns the polyline with these two points, otherwise add the point at the end of the list of points of the geometry (if the new geometry is valid, in the case of a polygon)

```
geometry({10,10}) + {15,15}              # polyline([{10.0;10.0},{15.0;15.0}])
polyline([{10,10},{20,20}]) + {15,15}   # polyline([{10.0;10.0},{20.0;20.0},
{15.0;15.0}])
polygon([{10,10},{10,20},{20,20},{20,10}]) + {25,25}    #
polygon([{10.0;10.0},{10.0;20.0},{20.0;20.0},{20.0;10.0},{10.0;10.0}])
```

```
polygon([{10,10},{10,20},{20,20},{20,10}]) + {5,5})   # polygon([{10.0;10.0},
{10.0;20.0},{20.0;20.0},{20.0;10.0},{5.0;5.0},{10.0;10.0}])
```

- if the right-operand is a number, see buffer and enlarged_by
- If both operands are numbers, performs a normal arithmetic sum and returns a float if one of them is a float.

```
1 + 1 # 2
1.0 + 1 # 2.0
```

If the left-hand operand is an int and the right-hand operand is not a number, casts the right-hand operand to an int.

```
1 + '34' # 35;
```

If the left-hand operand is a float and the right-hand operand is not a number, casts the right-hand operand to a float.

```
1.0 + '34' # 35.0;
```

If the left-hand operand is a string, casts the right-hand operand to a string and returns their concatenation.

```
'abc' + 'def' # 'abcdef';
```

If both operands are lists, returns their union.

```
[1, 2, 3] + [4, 5] # [1, 2, 3, 4, 5];
list ('abc') + list ('def') # ['a','b','c','d','e','f'];
```

If both operands are points, returns their sum.

```
{1, 2} + {4, 5} # {5, 7};
```

If the left-hand operand is a list and the right-hand operand is not a list, returns a new list with the right-hand operand added to the end of the left-hand operand. Matrices, points and strings are not considered as lists and will be added as individual elements.

```
[1,2,3] + 4 # [1,2,3,4];
['a','b','c'] + 'def' # ['a', 'b', 'c', 'def'];
```

- Comments: Sum of matrices is not yet implemented but should be available sometime in the future with the same syntax.
- see also: union

Top of the page

-

- Left-hand operand: a geometry, list, matrix, point, int or float.
- Right-hand operand: a geometry, list, matrix, point, int or float.
- Result: the difference of the two operands.
- Special cases:
  - if the left-operand is a geometry
    - if the right-operand is either a geometry, a list of agents or a species, returns the geometry resulting from de difference between both geometries (or between the left-operand geometry and the geometry of each agent of the list or of the species)
    - if the right-operand is a number, see reduced_by
  - If both operands are numbers, performs a normal arithmetic difference and returns a float if one of them is a float.

```
1 - 1 # 0;
1.0 - 1 # 0.0;
```

If the left-hand operand is an int and the right-hand operand is not a number, casts the right-hand operand to an int.

```
1 - '34' # -33;
```

If the left-hand operand is a float and the right-hand operand is not a number, casts the right-hand operand to a float.

```
1.0 - '34' # -33.0;
```

If both operands are lists, returns their difference (removes all the elements of the right-hand operand from the left-hand operand).

```
[1, 2, 3] - [3, 4, 5] # [1, 2];
list ('abc') - list ('abk') # ['c'];
```

If both operands are points, returns their difference.

```
{10, 20} - {4, 5} # {6, 15};
```

If the left-hand operand is a list and the right-hand operand is not a list, returns a new list with the right-hand operand removed from the left-hand operand.

```
[1,2,3] - 3 # [1,2];
```

- Comments: Difference of matrices is not yet implemented but should be available sometime in the future with the same syntax. Top of the page

*

- Left-hand operand: a geometry, an int or float.
- Right-hand operand: a geometry, an int or float.
- Result: a float, equal to the product of the two operands or a geometry.
- Comments: Product of matrices is not yet implemented but should be available sometime in the future with the same syntax.
- Special cases:
  - if the left-operand is a geometry, see scaled_by

Top of the page

/

- Left-hand operand: an int or float.
- Right-hand operand: an int or float.
- Result: a float, equal to the division of the left-hand operand by the rigth-hand operand.
- Special cases: if the right-hand operand is equal to zero, raises no exception and returns the maximum value of float instead.
- Comments: Division of matrices is not yet implemented but should be available sometime in the future with the same syntax.

Top of the page

^

- Left-hand operand: a int or float expression
- Right-hand operand: a int or float expression
- Result: an int or float value, equal the left-hand operand raised to the power of the right-hand operand.
- Special cases: if the right-hand operand is equal to 0, returns 1; if it is equal to 1, returns the left-hand operand.

Top of the page

<->

same signification as disjoint_from operator. Top of the page

around

- Left-hand operand: a float.
- Right-hand operand: any type (the operand will be casted in a geometry)
- Result: 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
- Return type: geometry
- Special cases: returns a circle geometry of radius right-operand if the left-operand is nil
- see also: geometry , - , circle , buffer

```
let circle_geom type: geometry value: circle(5);
let ring_geom type: geometry value:  10 around circle_geom # returns a the
ring geometry between 5 and 10.
```

Top of the page

at

- Left-hand operand: a list, string, matrix or point.
- Right-hand operand: an integer or a point.
- Result: the element of the left-hand operand located at the index specified by the right-hand operand.
- Comments:
  o strings, lists, and matrices are zero-based implementations, which means that the first index is 0 and the last one length-1. o at can be used in the left member of an assignment to assign a new value to one of the positions of a list, a matrix, a string, a point...

```
let my_list type: list of: int value: 12, 13, 14, 15;
let my_int type: int value: (my_list at 2) + 18; # my_int is equal to 32.
```

```
# the element on the first column and second line is now 42. (the rest of the
matrix is empty)
```

- Special cases:
  o if the left-hand operand is a string, a list, a point or a one-dimension matrix, and the index is less than 0 or greater than its length, returns nil. o if the left-hand operand is a two-dimensions matrix and the index is a point, the same rule applies for both dimensions. o if the left-hand operand is a one-dimension matrix, a list, a point or a string, and the index is a point, the x-coordinate of the point is used for the index. o if the left-hand operand is a string, returns a one-character string.

```
[1, 10, 3.0] @ 2 # 3.0;
'abcdef' at 0 # 'a';
```

Top of the page

div

- Left-hand operand: a int or float.
- Right-hand operand: a int or float.
- Result: an int, equal to the truncation of the division of the left-hand operand by the rigth-hand operand.
- Special cases: if the right-hand operand is equal to zero, raises no exception and returns the maximum value of int instead.

Top of the page

mod

- Left-hand operand: a int or float.
- Right-hand operand: a int or float.
- Result: an int, equal to the remainder of the integer division of the left-hand operand by the rigth-hand operand.
- Special cases: if the right-hand operand is equal to zero, raises no exception and returns the maximum value of int instead.

Top of the page

and

- Left-hand operand: a boolean expression
- Right-hand operand: a boolean expression
- Result: a bool value, equal to the logical and between the left-hand operand and the rigth-hand operand.
- Comments: both operands are always casted to bool before applying the operator. Thus, an expression like 1 and 0 is accepted and returns false.
- see also: bool , or .

Top of the page

or

- Left-hand operand: a boolean expression
- Right-hand operand: a boolean expression
- Result: a bool value, equal to the logical or between the left-hand operand and the rigth-hand operand.
- Comments: both operands are always casted to bool before applying the operator. Thus, an expression like 1 or 0 is accepted and returns true.
- see also: bool , and .

Top of the page

accumulate

- Left-hand operand: a list, point or string expression
- Right-hand operand: any.
- Result: a list containing all elements of left-hand expression and right-hand expression.

```
[1,2,3] accumulate [4,5,6] # [1,2,3,4,5,6]
```

Top of the page

among

- Left-hand operand: a int expression
- Right-hand operand: a list, string, point or matrix.
- Result: a list of length the value of the left-hand operand, containing random elements from the right-hand operand.
- Special cases: if the right-hand operand is empty, returns an empty list; if the left-hand operand is greater than the length of the right-hand operand, returns the right-hand operand.

```
3 among [1,2,3,4,5,6,7,8] # [3, 6, 7]
```

Top of the page

as_matrix

TO DO

at_location

same signification as translated_to operator. Top of the page

buffer

TO DO

closest_point_to

TO DO

closest_points_with

TO DO

closest_to

TO DO

collect

- Left-hand operand: a list, matrix, string or point.
- Right-hand operand: an arbitrary expression.
- Result: a list containing the result of the right-hand expressions applied to each of the elements of the left-hand operand.
- Comments: the order of the elements is kept. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4] collect (each + 10) # [11, 12, 13, 14];
'abcd' collect (each > 'a') # [false, true, true, true];
```

Top of the page

contains_all

TO DO

contains_any

TO DO

contour_points_every

TO DO

copy_between

- Left-hand operand: a list or a string
- Right-hand operand: a point
- Result: a string or a list containing the elements of the left-hand operand between the indexes provided by the x-coordinate (inclusive) and the y-coordinate (exclusive) of the right-hand operand.
- Comments: the length of the resulting list or string is (second index - first index).
- Special cases: if the first index is less than the second, returns an empty list or string; if the first index is less than 0, it is set to 0; if the second index is greater than the length of the left-hand operand, it is set to this length.

```
[1,2,3,4,5,6,7,8] copy_between {1,3} # [2,3];
'abcdefgh' between {2,5} # 'cde';
```

Top of the page

count

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: a boolean expression
- Result: an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] count (each > 3) # 5;
(list (species (self))) count ((first (each.location))> 30)  # all the agents
of the same species whose x-coordinate is greater than 30.
```

Top of the page

crosses

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: a boolean, equal to true if the agent geometry crosses the geometry of the localized entity passed in parameter
- Result type: boolean
- Special cases:
  - if one of the operand is null, returns false
  - if one operand is a point, returns false
- see also: intersects , <->

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

Top of the page

disjoint_from

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: a boolean, equal to true if the left-operand is disjoint from the right-operand
- Result type: boolean
- Special cases:
  - if one of the operand is null, returns true
  - if one operand is a point, returns false if the point is included in the geometry
- see also: intersects , <>

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

Top of the page

distance_to

- Left-hand operand: a point or an agent
- Right-hand operand: a point or an agent
- Result: a float, equal to the distance (in meters) between the two operands.
- Comments: in case a GIS environment is part of the model, the pathfinder is used to compute the distance; otherwise the distance in straight line is returned.
- Special cases: if one of the agents is not localized, returns 0.0.

```
((list (species (self))) - self) collect (each distance_to self) # collects
the distances between self and all the agents of the same species.
```

Top of the page

enlarged_by

TO DO Top of the page

first_with

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: a boolean expression
- Result: any value, equal to the first element of the left-hand operand that make the right-hand operand evaluate to true.
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] first_with (each > 3) # 4;
(list species self) first_with ((first (each.location))> 30)  # the first
agent of the same species whose x-coordinate is greater than 30.
```

Top of the page

group_by ***

TO DO

in

- Left-hand operand: an expression or a list
- Right-hand operand: a list, string, matrix or point

- Result: if the left_hand operand is not a list, true if the left-hand operand is equal to one of the elements of the right-hand operand, otherwise false; if it is a list, returns true if all its elements are present in the right-hand operand, otherwise returns false;

```
3 in [1,2,3,4,5,6,7,8] # true;
[3, 10] in [1,2,3,4,5,6,7,8] # false;
'bc' in 'abcded' # true;
['a','b','cd'] in 'abcdef' # true;
```

Top of the page

index_of

- see: last_index_of.

Top of the page

inside

TO DO

inter

- Left-hand operand: a geometry
- Right-hand operand: a geometry, a point
- Result: a geometry, equals the intersection between the two operands, or null if the intersection is empty.
- Result type: geometry
- Special cases:
  - if the right-hand operand or the left-hand operand is null, returns null.
  - if the right-hand operator is a point, returns a geometry containing this point if the left operator geometry covers the point, null otherwise.

```
polygon([{10,10},{10,20},{20,20},{20,10}]) inter polygon([{15,15},{15,25},
{25,25},{25,15}]))
                                # polygon([{15.0;20.0},{20.0;20.0},{20.0;15.0},
{15.0;15.0},{15.0;20.0}])
polygon([{10,10},{10,20},{20,20},{20,10}]) inter {15,15}   #
geometry({15.0;15.0})
```

Top of the page

intersection

same signification as inter operator. Top of the page

intersects

- Left-hand operand: a geometry
- Right-hand operand: a geometry, a point
- Result: a boolean, equal to true if the left-operand intersects the right-operand
- Result type: boolean
- Comments: if the right-hand operand or the left-operand is not present, returns false.

```
square(5) intersects {10,10} # false
```

Top of the page

last_index_of

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: an expression
- Result: an int, equal to the index of the first (resp. last) occurence of the right-hand operand in the left-hand operand.
- Comments: if the right-hand operand is not present, returns -1.

```
[1,2,3,4,5,6,7,8] index_of 3 # 2;
'abcdefgh' index_of 'bcf' # -1;
```

Top of the page

masked_by

- Left-hand operand: any type (the operand will be casted in a geometry)
- Right-hand operand: any type (the operand will be casted in a geometry)
- Result: geometry representing the part of the right operand visible from the point of view of the agent using the operator while considering the obstacles defined by the left operand
- Return type: geometry

```
perception_geom masked_by obstacle_geom # returns the geometry representing
the part of perception_geom visible from the agent position considering the
obstacle obstacle_geom
```

Top of the page

max_of

- see: min_of .

Top of the page

min_of

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: an in or float expression
- Result: an int or float, equal to the maximum (resp. minimum) value of the right-hand expression evaluated on each of the elements of the left-hand operand
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,10,4,7,6,7,8] max_of (each * 100) # 1000;
(list (species (self))) min_of (first (each.location))  # the smallest x-coordinate of the agents of the same species as self.
```

Top of the page

neighbours_at

- Left-hand operand: an agent
- Right-hand operand: an int or float (expressing a distance in meters)
- Result: a list, containing all the agents located in the circle of radius equal to the right-hand operand, and center the location of the left-hand operand.
- Comments: if the left-hand operand is a grid cell, returns only grid cells of the same environment; if it is a regular agent, returns all the agents, whatever their species, save for grid cells.
- Special cases: if the left-hand operand is a not localized agent, returns an empty list.

```
(self neighbours_at (10)) of_species (species (self)) # all the agents of the same species situated at a distance greater or equal to 10 of self.
```

Top of the page

of_generic_species

TO DO

of_species

- Left-hand operand: a list (of agents)
- Right-hand operand: a species expression
- Result: a list, containing the agents of the left-hand operand whose species is that denoted by the right-hand operand.
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements. The expression agents of_species (species self) is equivalent to agents where (species each = species self); however, the advantage of using the first syntax is that the resulting list is correctly typed with the right species, whereas, in the second syntax, the parser cannot determine the species of the agents within the list (resulting in the need to cast it explicitely if it is to be used in an ask statement, for instance).

```
(self neighbours_at 10) of_species (species (self)) # all the neighbouring
agents of the same species.
```

Top of the page

overlapping

TO DO

overlaps

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: a boolean, equal to true if the left-operand is not disjoint from the right-operand
- Result type: boolean
- Special cases:
  - if one of the operand is null, returns false
  - if one operand is a point, returns true if the point is included in the geometry
- see also: intersects , <> , disjoint_from

```
polyline([{10,10},{20,20}]) overlaps polyline([{15,15},{25,25}])
                  # true
```

```
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},
{25,25},{25,15}])  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps geometry({25,25})
                  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{35,35},{35,45},
{45,45},{45,35}])  # false
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polyline([{10,10},
{20,20}])                  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps geometry({15,15})
                  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{0,0},{0,30},
{30,30}, {30,0}])     # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{15,15},{15,25},
{25,25},{25,15}])  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps polygon([{10,20},{20,20},
{20,30},{10,30}])  # true
```

Top of the page

partially_overlaps

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: a boolean, returns true if the left-operand geometry partially overlaps the right-operand geometry
- Result type: boolean
- Special case: if the right-hand operand is null, returns false.
- Comments: if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).

```
polyline([{10,10},{20,20}]) partially_overlaps polyline([{15,15},{25,25}])
                            # true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
polygon([{15,15},{15,25},{25,25},{25,15}])  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
geometry({25,25})                           # true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
polygon([{35,35},{35,45},{45,45},{45,35}])  # false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
polyline([{10,10},{20,20}])                 # false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
geometry({15,15})                           # false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps polygon([{0,0},
{0,30},{30,30}, {30,0}])     # false
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
polygon([{15,15},{15,25},{25,25},{25,15}])  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps
polygon([{10,20},{20,20},{20,30},{10,30}])  # false
```

Top of the page

points_at

TO DO

reduced_by

TO DO

rotated_by

TO DO

select

- see: where.

scaled_by

TO DO Top of the page

simple_clustering_by_distance

TO DO

simple_clustering_by_envelope_distance

TO DO

simplification

TO DO

sort_by

- Left-hand operand: a list, string, point, or matrix

- Right-hand operand: an int, float or string expression.
- Result: a list, containing the agents of the left-hand operand sorted in ascending order by the value of the right-hand operand when it is evaluated on them.
- Comments: the left-hand operand is casted to a list before applying the operator. Therefore, a species value can be used directly to represent all the agents of this species. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
(self neighbours_at 10) sort_by (first (each.location)) # all the neighbouring
agents, sorted by their x-coordinate.
```

Top of the page

starts_with

- Left-hand operand: a string
- Right-hand operand: a string
- Result: true is the left-hand operand starts with the right-end operand.
- Comments: equivalent to (left-hand_operand [[[#index_of] right-hand_operand) = 0, but faster.

Top of the page

split_at

- Left-hand operand: a geometry
- Right-hand operand: a point
- Result: a list of geometry,
  - if the left-operand is a ploryline, the list contains two polylines, the two part of the left-operand slit at thea given right-operand point
  - if the right-operand is a multi-polyline, the list contains the two polylines resulting from the spliting of the polyline the closest to the point
- Special cases:
  - if the left-operand is a point or a polygon, returns an empty list
- Result type: list of geometry

```
polyline([{1,2},{4,6}]) split_at {7,6}  #  geometry([polyline([{1.0;2.0},
{7.0;6.0}]),polyline([{7.0;6.0},{4.0;6.0}])])
circle(4) split_at {5,5}]  #  []
```

Top of the page

split_using

- see: tokenize .

Top of the page

tokenize

- Left-hand operand: a string
- Right-hand operand: a string (delimiters)
- Result: a list, containing the sub-strings (tokens) of the left-hand operand delimited by each of the characters of the right-hand operand.
- Comments: delimiters themselves are excluded from the resulting list.

```
'to be or not to be, that is the question' split_using ' ,' #
['to','be','or','not,'to','be','that','is','the','question'].
'to be or not to be, that is the question' tokenize ' ,' #
['to','be','or','not,'to','be','that','is','the','question'].
```

Top of the page

touches

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: returns true if the left-operand geometry touches the right-operand geometry
- Result type: boolean
- 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.

```
polyline([{10,10},{20,20}]) touches geometry({15,15})  # false
polyline([{10,10},{20,20}]) touches geometry({10,10})  # true
geometry({15,15}) touches geometry({15,15})            # false
polyline([{10,10},{20,20}]) touches polyline([{10,10},{5,5}])    # true
polyline([{10,10},{20,20}]) touches polyline([{5,5},{15,15}])    # false
polyline([{10,10},{20,20}]) touches polyline([{15,15},{25,25}])  # false
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{15,15},{15,25},
{25,25},{25,15}])  # false
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,20},{20,20},
{20,30},{10,30}])  # true
polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{10,10},{0,10},
{0,0},{10,0}])       # true
polygon([{10,10},{10,20},{20,20},{20,10}]) touches geometry({15,15})   # false
```

```
polygon([{10,10},{10,20},{20,20},{20,10}]) touches geometry({10,15})   # true
```

Top of the page

transformed_by

TO DO

translated_by

- Left-hand operand: a geometry
- Right-hand operand: a point
- Result: applies a translation operation (vector (dx, dy)) to the geometry and returns the translated geoemtry
- Return type: geometry.
- see also: translated_to

```
geometry({10,10}) translated_by {5,5}                              #
geometry({15.0;15.0})
polyline([{10,10},{20,20}]) translated_by {5,5}                   #
polyline([{15.0;15.0},{25.0;25.0}])
polygon([{10,10},{10,20},{20,20},{20,10}]) translated_by {5,5} #
polygon([{15.0;15.0},{15.0;25.0},{25.0;25.0},{25.0;15.0}])
```

Top of the page

translated_to

TO DO

union (binary)

- Left-hand operand: a geometry
- Right-hand operand: a geometry
- Result: the geometry composed of the union of the two operands.
- Return type: geometry.
- Special cases:
  - if both operands are null, returns null
  - if one of the two operands is null, return the not null operand
- see also: union , +

```
polygon([{10,10},{10,20},{20,20},{20,10}]) union polygon([{15,15},{15,25},
{25,25},{25,15}])
    # polygon([{10.0;10.0},{10.0;20.0},{15.0;20.0},{15.0;25.0},{25.0;25.0},
{25.0;15.0},{20.0;15.0},{20.0;10.0},{10.0;10.0}])
```

Top of the page

where

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: a boolean expression
- Result: a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] select (each > 3) # [4, 5, 6, 7, 8];
(list species self) where ((first each.location)> 30)  # all the agents of the
same species whose x-coordinate is greater than 30;
'abcdef' select (each < 'd') # ['a', 'b','c'].
```

Top of the page

with_max_of

- see: with_min_of.

Top of the page

with_min_of

- Left-hand operand: a list, string, matrix or point
- Right-hand operand: an int, float or string expression
- Result: a list containing the elements of the left-hand operand that maximize (resp. minimize) the value of the right-hand operand.
- Comments: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
(list species self) with_max_of (each.var)  # all the agents of the same
species whose variable var is equal to the current maximum value of var
(equivalent to (list species self) where (each.var = max (list species self
collect (each.var))) -- only faster and easier to write);
['abc', 'bcde', 'fgh', 'fffde'] with_min_of (each.length) # ['abc','fgh'].
```

Top of the page

with_precision

- Left-hand operand: an int or float.
- Right-hand operand: an int.
- Result: round off the value of left-hand operand to the precision given by the value of right-hand operand.

```
12345.78943 with_precision 2 # 12345.79
123 with_precision 2 # 123.00
```

Top of the page

Ternary operators

? :

- Left-hand operand: a boolean expression
- Middle operand: an expression
- Right-hand operand: an expression
- Result: if the left-hand operand evaluates to true, returns the value of the middle operand, otherwise that of the right-hand operand

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

These functional tests can be combined together (here, the color variable takes three values with respect to the value of food, above 5, between 2 and 5, and below 2):

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

Top of the page

Variables

Declaration

base

Except temporary variables, which are declared with their own syntax within behaviors or actions, all other variables are declared using the following one:

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

This declaration can be simplified by replacing the keyword var by the datatype:

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

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

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

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

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

init or <-

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

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

which is equivalent to:

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

and to:

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

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

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

const

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

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

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

update

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

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

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

- Automatically evolving variables:

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

- Sticky variables:

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

- Conditionally evolving variables:

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

Special attributes

function

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

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

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

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

parameter

This attribute can only be used in the context of global variables, i.e. variables declared in the world species in the global section. Indicates that the value of the variable will (or can) be defined by an external input : either a file or an optimization process (in the case of batch simulations), or the user (in the case of interactive simulations with a user interface). Makes **const** turns to false if it has been defined. For example, declaring:

```
var max_energy type: int init: 300 parameter:true;
```

will translate to this in the user interface:

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

```
var max_energy type: int init: 300 parameter: "Maximum energy for the agents";
```

max, min

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

```
var max_energy type: int init: 300 min: 100 max: 3000;
```

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

of

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

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

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

Naming variables

Reserved Keywords

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

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

Naming conventions

Alphanumeric characters can be use to variable name. No space is accepted.

Accessing variables

Direct access

Global variables, species variables and temporary variables declared in the same scope can be accessed directly by agents. For instance:

```
species animal {
  var energy type: float init: 1000 min: 0 max: 2000 value: energy - 0.001;
  var age_in_years type: int init: 1 value: age_in_years + int (time / 365);
  action eat {
      arg amount default: 0;
      let gain type: float value: amount / age_in_years;
      set energy value: energy + gain;
  }
}
```

In this declaration, we are able to directly name time, the global built-in variable, energy and age_in_years, which are defined as species variables, amount, which is an argument to the action eat and gain, a temporary variable within the action.

Remote access

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

```
remote_agent.variable
```

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

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

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

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

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

Keywords

constants

nil

Represents the null value (or undefined value). It is the default value of variables of type agent, point or species when they are not initialized.

true, false

Represent the two possible values of boolean variables or expressions. (see bool).

pseudo-variables

Pseudo-variables are special variables whose value cannot be changed by agents (but can change depending on the context of execution).

self

self is a pseudo-variable (can be read, but not written) that always holds a reference to the executing agent.

- Example (sets the friend field of another random agent to self and reversely) :

```
let potential_friend value: one_of (list (species(self)) - self);
if condition: potential_friend != nil {
    set potential_friend.friend value: self;
    set friend value: potential_friend;
}
```

myself

myself is the same as self, except that it needs to be used instead of self in the definition of remotely-executed code (ask and |create commands). myself represents the sender agent when the code is executed by the target agent.

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

```
ask target: first (list (species (self))){
    set color value: myself.color;
}
```

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

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

each

each is a pseudo-variable only used in filter expressions following operators such as where or first_with. It represents, in turn, each of the elements of the target datatype (a list, string, point or matrix, usually).

units

Units can be used to qualify the values of numeric variables. By default, unqualified values are considered as:

- meters for distances, lengths...
- seconds for durations
- cubic meters for volumes
- kilograms for masses

So, an expression like

```
let foo type: float value: 1;
```

will be considered as 1 meter if foo is a distance, or 1 second if it is a duration, or 1 meter/second if it is a speed. If one wants to specify the unit, it can be done very simply by adding the unit name after the numeric value, like:

```
let foo type: float value: 1 centimeter;
```

In that case, the numeric value of foo will be automatically translated to 0.01 (meter). It is recommended to always use float as the type of the variables that might be qualified by units (otherwise, for example in the previous case, they might be truncated to 0). Several units names are allowed as qualifiers of numeric variables. As they can be used in expressions directly, they are considered as reserved keywords (and therefore cannot be used for naming variables and species). Their complete list is:

length

meter (default), meters, m, centimeters, centimeter, cm, millimeter, millimeters, mm, decimeter, decimeter, dm, kilometer, kilometers, km, mile, miles, yard, yards, inch, inches, foot, feet, ft.

time

second (default), seconds, sec, s, minute, minutes, mn, hour, hours, h, day, days, d, month, months, year, years, y, millisecond, milliseconds, msec.

mass

kilogram (default),kilogram, kilo, kg, ton, tons, t, gram, grams, g, ounce, ounces, oz, pound, pounds, lb, lbm.

surface

square_meter (default), m2, square_meters, square_mile, square_miles, sqmi, square_foot, square_feet, sqft.

volume

m3 (default), liter, liters, l, centiliter, centiliters, cl, deciliter, deciliters, dl, hectoliter, hectoliters, hl. These represent the basic metric and US units. Composed and derived

units (like velocity, acceleration, special volumes or surfaces) can be obtained by combining these units using the **and / operators. For instance:**

```
var one_kmh type: float init: 1 km / h const: true;
var one_microsecond type: float init: 1 sec / 1000;
var one_cubic_inch type: float init: 1 sqin * 1 inch;
... etc ...
```

Skills

Overview

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

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

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

- moving: for agents that need to move.
- carrying: for agents that need to carry other agents.
- communicating: for agents that need to communicate using the FIPA protocols.
- exploring: for agents that need to explore a region of space.

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

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

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

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

Or, to obtain ever growing agents:

```
var speed type:float max:100 min:0 value: area * 1.01;
```

Or, to change their capacity in a behavior:

```
if condition: capacity = 5 {
    set capacity value: 10;
}
```

moving

variables

speed

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

heading

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

destination

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

actions

follow

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

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

```
do action: follow {
    arg speed value: speed * 2;
    arg path value: road_path;
}
```

goto

moves the agent towards the target passed in the arguments.

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

```
do action: goto{
    arg target value: one_of (list (species (self)));
    arg speed value: speed * 2;
    arg on value: road_network;
}
```

move

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

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

```
do action: move {
    arg speed value: speed - 10;
    arg heading value: heading + rnd (30);
    arg bounds value: agentA;
}
```

wander

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

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

```
do action: wander{
    arg speed value: speed - 10;
    arg amplitude value: 120;
    arg bounds value: agentA;
}
```

carrying

variables

capacity

int, the maximum number of agents that can be carried

contents

list, read-only, the current list of agents carried.

actions

drop

makes the agent leave, at its location, all or some of the agents it was carrying.

- → agents: list, optional, the list of agents to drop. If not specified, all the agents carried are dropped.
- ← return: list, the list of agents actually dropped.

```
do action: drop
    arg agents value: contents of_species foo;
}
```

load

makes the agent pick & carry all or some of the agents specified by the arguments.

- → agents: list, optional, the list of agents to load. If not specified, the action is ignored.
- → number: int, optional, the number of agents to load from the list. If not specified, all the agents are loaded until the capacity is reached.
- ← return: list, the list of agents actually loaded.

```
do action: load {
    arg agents value: (agents_overlapping (self)) of_species foo;
    arg number value: 10;
}
```

communicating

This skill implements the major FIPA communication and negotiation protocols and enable the agents to communicate using them. It brings along two new datatypes : message and conversation. GAMA implements the following interaction protocols of FIPA : FIPA Brokering, FIPA Contract Net, FIPA Propose, FIPA Query, FIPA Request, FIPA Request When, FIPA Subscribe. Besides the standard interaction protocols of FIPA, GAMA support a special interaction protocol named "no-protocol".

variables

conversations

list, read-only, the conversation in which the agent is currently engaged.

messages

list, read-only, the messages received by the agent.

accept_proposals, agrees, cancels, cfps, failures, informs, proposes, queries, refuses, reject_proposals, requests, requestWhens, subscribes

list, read-only, the messages received by the agent, broken down by performative.

actions

send

allows the agent to send an existing message, or to create one and send it immediately. Either use the argument:

- → message: message, optional, the message to send.

Or a combination of the following arguments:

- → receivers: list, mandatory, the list of agents to send the message to.
- → content: list, mandatory, the content of the message. Can be anything.
- → performative: string, mandatory. The performative of the message (see this page)
- → protocol: string, mandatory, the protocol followed by this message (see this page)
- → conversation: conversation, optional, the conversation in which the message should be inserted. If nil, creates a new conversation.
- ← return: message, the message actually sent or nil if an error has occurred.

```
let mes type: list of: message value: [];
create species: message number: 1 return: mes {
    set receivers value: (agents_overlapping (self));
    set content value:['Hello !'];
    set protocol value:'no-protocol';
    set performative value:'inform';
}
do action: send {
    arg message value: first(mes);
}
```

```
do action: send {
    set receivers value: (agents_overlapping (self));
    set content value:['Hello !'];
    set protocol value:'no-protocol';
    set performative value:'inform';
```

```
}
```

end, failure, inform, propose, query, refuse, reject-proposal, request, subscribe

replies a list of messages with the corresponding performatives "end", "failure", "inform", "propose", "query", "refuse", "reject-proposal", "request", "subscribe". The replied messages will be filled automatically with the corresponding performatives.

- $\rightarrow$ message: list, mandatory, messages to reply to.
- $\rightarrow$ content: list, optional, the content of the replied messages.

```
let noProtocolInformMsgs type: list value:(informs) where: ((species
(each.sender) = foodCarrier) and (each.protocol = 'no-protocol'));
do action: end {
    arg message value: noProtocolInformMsgs;
}
```

data types

message

An instance of message represents a piece of information exchanged between agents. It has the following attributes :

- sender : agent, the sender of the message.
- receivers : a list of agents, contains all the receivers of the message.
- performative : string, the performative of the corresponding message. For more details concerning the performative, please refer to the FIPA Communicative Act Library Specification.
- content : list, contains the content of the corresponding message. The modeller can put whatever content in this list.

conversation

An instance of conversation represents a FIPA interaction protocol. It helps ensure that a conversation between agents respects the specification of a corresponding protocol defined by FIPA. It has the following attributes :

- messages : list, the currently unread messages of the conversation.

- protocol : string, the name of the interaction protocol that this conversation respects.
- initiator : agent, the agent which initiates the conversation/the interaction protocol.
- participants : a list of agents, all the participants of the conversation (except for the initiator).
- ended : bool, helps determine if the corresponding conversation/interaction protocol has already finished or not yet.

Notes

GAMA supports a special interaction protocol named "no-protocol". If the modeller finds that no supported FIPA Interaction Protocols fits his modelling case, he can use the "no-protocol". Uppon declaring a conversation following this procotol, the modeller is free to send messages having whatever performative between agents. But it 's upto the modeller to finish up the corresponding conversation by using the "end" primitive.

exploring

This skill implements the patrolling capability of agents, enables the agents to go on patrol in a list of given points.

variables

points

list, the list of points to patrol. This list must not be empty before the agents can begin its patrolling activity.

patrolling

bool, determines if the corresponding agent is going on patrol or not.

actions

patrol

do the patrolling activity if the list of points is not empty.

- → speed: float, optional, the speed of the corresponding agent.

```
task name: patrol_the_terrain while: patrolTheTerrain {
  priority if: patrolTheTerrain value: 10 else: 0;
  duration max: 2 hours;
  repeat action: patrol {
    arg speed value: maxSpeed;
  }
}
```

Built-in Agents

Introduction

It is possible to use in the models a set of built-in agents. These agents allow to directly use some advance features like clustering, multi-criteria analysis, etc. The creation of these agents are similar as for other kinds of agents:

```
create species: my_built_in_agent return: the_agent;
```

The list of available built-in agents in GAMA is:

- cluster_builder: allows to use clustering techniques on a set of agents.
- multicriteria_analyzer: allows to use multi-criteria analysis methods.

So, for instance, to be able to use clustering techniques in the model:

```
create species: cluster_builder return: clusterer;
```

cluster_builder

The **cluster_builder** agent allows to divide a set of agents into different clusters according to the values of some of their attributes. This agents is built from the weka library : most of the clustering algorithms are directlty based on their weka implementation.

actions

simple_clustering_by_distance

Returns groups of agents using hierarchical clustering. The distance between agents are directly computed from the agent locations (euclidean distance). The distance between two groups of agents corresponds to the min of distances between the agents of each group.

- $\rightarrow$ agents: list, the list of Entities to be divided into groups.

- → dist_min: float, minimal distance between two groups. By default, num_clusters = -1.
- → distance_geom: bool, optional, if true, uses the distance between the agent geometry; otherwise uses the distance between the agent centroid (more optimize). By default, distance_geom = false.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let groups type: list value: self.simple_clustering_by_distance [agents::ags,
dist_min::10.0];
```

clustering_cobweb

Returns groups of agents using the Cobweb and Classit clustering algorithms (Weka implementation). For more information see: D. Fisher (1987). Knowledge acquisition via incremental conceptual clustering. Machine Learning. 2(2):139-172; J. H. Gennari, P. Langley, D. Fisher (1990). Models of incremental concept formation. Artificial Intelligence. 40:11-61.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → acuity: float, optional, minimum standard deviation for numeric attributes. By default, acuity = 1.0.
- → cutt_off: float, optional, set the category utility threshold by which to prune nodes. By default, cutoff = 0.0028209479177387815.
- → seed, int, optional, The random number seed to be used. By default, seed = 42.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
let groups type: list value: self.clustering_cobweb [agents::ags,
attributes::attribs];
```

clustering_DBScan

Returns groups of agents using the DBScan algorithm (Weka implementation). For more information see: Martin Ester, Hans-Peter Kriegel, Joerg Sander, Xiaowei Xu: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Second International Conference on Knowledge Discovery and Data Mining, 226-231, 1996.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → distance_f: string, optional, The distance function to use : 2 possible distance functions: (by default) euclidean; and 'manhattan'.
- → epsilon: float, optional, radius of the epsilon-range-queries. By default, epsilon = 0.9.
- → min_points: int, optional, minimun number of [DataObjects] required in an epsilon-range-query. By default, min_points = 6.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
let groups type: list value: self.clustering_DBScan [agents::ags,
attributes::attribs];
```

clustering_em

Returns groups of agents using the EM (expectation maximisation) algorithm (Weka implementation).

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 100.
- → num_clusters: int, optional, set number of clusters. if num_clusters equals -1, EM decides how many clusters to create by cross validation. By default, num_clusters = -1.
- → min_std_dev: float, optional, set minimum allowable standard deviation. By default, min_std_dev = 1.0E-6.
- → seed, int, optional, The random number seed to be used. By default, seed = 100.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
let groups type: list value: self.clustering_em [agents::ags,
attributes::attribs];
```

clustering_farthestFirst

Returns groups of agents using the farthestFirst algorithm (Weka implementation). For more information see: Hochbaum, Shmoys (1985). A best possible heuristic for the k-center problem. Mathematics of Operations Research. 10(2):180-184.

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → num_clusters: int, optional, set number of clusters. By default, num_clusters = 2.
- → seed, int, optional, The random number seed to be used. By default, seed = 1.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
let groups type: list value: self.clustering_farthestFirst [agents::ags,
attributes::attribs];
```

clustering_simple_kmeans

Returns groups of agents using the K-Means algorithm (Weka implementation).

- → agents: list, the list of Entities to be divided into groups.
- → attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- → distance_f: string, optional, The distance function to use : 4 possible distance functions: (by default) euclidean; otherwise, 'chebyshev', 'manhattan' and 'levenshtein'.
- → dont_replace_missing_values: bool, optional, Replace missing values globally with mean/mode. By default, dont_replace_missing_values = false.
- → max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 500.
- → num_clusters: int, optional, set number of clusters. By default, num_clusters = 2.
- → preserve_instances_order: bool, optional, Preserve order of instances. By default, preserve_instances_order = false.
- → seed, int, optional, The random number seed to be used. By default, seed = 10.
- ← return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
```

```
let groups type: list value: self.clustering_simple_kmeans [agents::ags,
attributes::attribs];
```

clustering_xmeans

Returns groups of agents using the X-Means algorithm (Weka implementation). "X-Means is K-Means extended by an Improve-Structure part. In this part of the algorithm the centers are attempted to be split in its region. The decision between the children of each center and itself is done comparing the BIC-values of the two structures. For more information see: Dan Pelleg, Andrew W. Moore: X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: Seventeenth International Conference on Machine Learning, 727-734, 2000." (Weka documentation).

- $\rightarrow$ agents: list, the list of Entities to be divided into groups.
- $\rightarrow$ attributes: list, the list of the agent attributes (string: the attribute names) used to divide the agents into groups. For the moment, only numeric attributes are considered.
- $\rightarrow$ bin_value: float, optional, Set the value that represents true in the new attributes. By default, bin_value = 1.0.
- $\rightarrow$ cut_off_factor: float, optional, the cut-off factor to use. By default, cut_off_factor = 0.5.
- $\rightarrow$ distance_f: string, optional, The distance function to use : 4 possible distance functions: (by default) euclidean; otherwise, 'chebyshev', 'manhattan' and 'levenshtein'
- $\rightarrow$ max_iterations: int, optional, the maximum number of iterations to perform. By default, max_iterations = 1.
- $\rightarrow$ max_kmeans: int, optional, the maximum number of iterations to perform in KMeans. By default, max_iterations = 1000.
- $\rightarrow$ max_kmeans_for_children: int, optional, the maximum number of iterations KMeans that is performed on the child centers. By default, max_kmeans_for_children = 1000.
- $\rightarrow$ max_num_clusters: int, optional, set maximum number of clusters. By default, max_num_clusters = 4.
- $\rightarrow$ min_num_clusters: int, optional, set minimum number of clusters. By default, min_num_clusters = 2.
- $\rightarrow$ seed, int, optional, The random number seed to be used. By default, seed = 10.
- $\leftarrow$ return: list, a list of groups; each group is a list of agents.

```
let ags type: list of: agent value: [agentA, agentB, agentC, agentD, agentE];
let attribs type: list of: string value:
['area','food_quantity','proximity_to_roads'];
let groups type: list value: self.clustering_xmeans [agents::ags,
attributes::attribs];
```

multicriteria_analyzer

The **multicriteria_analyzer** agent allows to make a decision from a set of candidate solutions according to a set of criteria.

actions

weighted_means_DM

Returns the index of the candidate with the highest utility. The utility of each candidate is computed by a weighted means.

- $\rightarrow$ criteria: list, the list of criteria. A criterion is a map that contains two elements: a name, and a weight.
- $\rightarrow$ candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- $\leftarrow$ return: int, the index of the selected candidate.

```
let crits type: list value: [[name::'proximity', weight::1], [name::'quality',
weight::3], [name::'usefulness', weight::2]];
let cands type: list value: [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],
[0.9, 0.2, 0.4],[0.6, 0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int value: self.weighted_means_DM [criteria::crits,
candidates::cands];
```

promethee_DM

Returns the index of the *best* candidate according to the Promethee II method. This method is based on a comparison per pair of possible candidates along each criterion: all candidates are compared to each other by pair and ranked. More information about this method can be found in Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research(2009)

- $\rightarrow$ criteria: list, the list of criteria. A criterion is a map that contains fours elements: a name, a weight, a preference value (p) and an indifference value (q). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant.

- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list value: [[name::'proximity', weight::1, p::0.9, q::0.1],
[name::'quality', weight::3, p::1.0, q::0.0],[name::'usefulness', weight::2,
p::0.8, q::0.2]];
let cands type: list value: [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],
[0.9, 0.2, 0.4],[0.6, 0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int value: self.promethee_DM [criteria::crits,
candidates::cands];
```

electre_DM

Returns the index of the *best* candidate according to a method based on the ELECTRE methods. The principle of the ELECTRE methods is to compare the possible candidates by pair. These methods analyses the possible outranking relation existing between two candidates. An candidate outranks another if this one is at least as good as the other one. The ELECTRE methods are based on two concepts: the concordance and the discordance. The concordance characterises the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterises the fact that, for an outranking relation to be validated, none of the criteria in the minority should oppose too strongly this assertion. These two conditions must be true for validating the outranking assertion. More information about the ELECTRE methods can be found in Figueira, J., Mousseau, V., Roy, B.: ELECTRE Methods. In: Figueira, J., Greco, S., and Ehrgott, M., (Eds.), Multiple Criteria Decision Analysis: State of the Art Surveys, Springer, New York, 133--162 (2005)

- → criteria: list, the list of criteria. A criterion is a map that contains fives elements: a name, a weight, a preference value (p), an indifference value (q) and a veto value (v). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant. The veto value represents the threshold from which the difference between two criterion values disqualifies the candidate that obtained the smaller value.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list value: [[name::'proximity', weight::1, p::0.9, q::0.1,
v::0.95], [name::'quality', weight::3, p::0.8, q::0.0, v:1.0],
    [name::'usefulness', weight::2, p::0.8, q::0.2, v::0.9]];
```

```
let cands type: list value: [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],
[0.9, 0.2, 0.4],[0.6, 0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int value: self.electre_DM [criteria::crits,
candidates::cands];
```

evidence_theory_DM

Returns the index of the *best* candidate according to a method based on the Evidence theory. This theory, which was proposed by Shafer (Shafer G (1976) A mathematical theory of evidence, Princeton University Press), is based on the work of Dempster (Dempster A (1967) Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, vol. 38, pp. 325--339) on lower and upper probability distributions.

- → criteria: list, the list of criteria. A criterion is a map that contains seven elements: a name, a first threshold s1, a second threshold s2, a value for the assertion "this candidate is the best" at threshold s1 (v1p), a value for the assertion "this candidate is the best" at threshold s2 (v2p), a value for the assertion "this candidate is not the best" at threshold s1 (v1c), a value for the assertion "this candidate is not the best" at threshold s2 (v2c). v1p, v2p, v1c and v2c have to been defined in order that: v1p + v1c < = 1.0; v2p + v2c < = 1.0.
- → candidates: list, the list of candidates. A candidate is a list of floats; each float representing the value of a criterion for this candidate.
- ← return: int, the index of the selected candidate.

```
let crits type: list value: [[name::'proximity', s1::0.1, s2::0.8, v1p::0,
v2p::0.3, v1c::0.5, v2c::0], [name::'quality',s1::0.0, s2::0.8, v1p::0.05,
v2p::0.5, v1c::0.6, v2c::0], [name::'usefulness', s1::0.0, s2::0.8, v1p::0,
v2p::0.2, v1c::0.9, v2c::0]];
let cands type: list value: [[0.8, 0.1, 0.3],[0.5, 0.7, 0.5],[0.1, 0.5, 0.9],
[0.9, 0.2, 0.4],[0.6, 0.5, 0.5],[0.7, 0.4, 0.3]];
let index type: int value: self.evidence_theory_DM [criteria::crits,
candidates::cands];
```

Built-in Items

Introduction

Species, when declared in a model, inherit some of their attributes (variables, actions) from the Java class that back them behind the scene. For "regular" agents, the number of these attributes is voluntarily limited. In a sense, that class acts exactly like a basic skill, and gives the agents a (very) limited number of common capabilities. However, some agents in the model are based on more specialized Java classes: this is the case of the world, which provides some critical global knowledge and functions to the modeller and the agents. In this section, we start by examining the built-in variables and actions common to all the agents. We then describe the global ones defined on the world.

name

string. Each agent has a default name (concatenation of its species name and unique index), which can be changed at will to something more useful for the modeler (if needed).

Note

other "built-in" read-only attributes can be accessed through special operators:

- species or species_of, which return the species of the agent,
- int, which returns its index;

or keywords:

- self and myself, which return the agent itself.

Built-in actions

Three built-in actions are provided to the agents.

write

makes the agent output an arbitrary message in the console.

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

```
do action: write {
    arg message value: 'This is a message from ' + self;
}
```

debug

makes the agent output an arbitrary message in the console. The message is automatically prefixed with the cycle of the simulation and followed by a carriage return and postfixed with information concerning the agent that called this action.

→ message: string, mandatory, the message to display.

```
do action: debug {
    arg message value: 'This is a message from ' + self;
}
```

error

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

→ message: string, mandatory, the message to display.

```
do action: error {
    arg message value: 'This is an error raised by ' + self;
}
```

tell

makes the agent output a dialog (if the simulation contains a user interface). The simulation goes on, but its interface is not accessible until the dialog is closed (use with caution, as it may prevent the user from accessing the interface).

→ message: string, mandatory, the message to display.

```
do action: tell {
    arg message value: 'This is a message dialog raised by ' + self;
}
```

Global built-in variables

Global variables can be accessed by the world and every other agent in the model.

time

int, read-only, represents the current simulated time in seconds (the default unit). Begins at zero. Although time is a read-only variable, it is possible to control its maximum value by redeclaring it. When the maximum is reached during a simulation, this simulation is automatically stopped.

```
global {
...
    var time type: int max: 1000;
...
}
```

step

int, represents the time step between two executions of the set of agents, in seconds. Its default value is 1. Each turn, the value of time is incremented by the value of step. The definition of step must be coherent with that of the agents' variables like speed.

```
global {
...
    var step type: int max: 10;
...
}
```

seed

float, represents the seed used in the computation of random numbers. Keeping the same seed between two runs of the same model ensures that the sequence of events will remain the same, which can be useful when debugging a model. Declaring it as a parameter allows the user or an external process (batch, for instance) to modify it.

```
global {
...
```

```
    var seed type: int value: 354 parameter: true;
...
}
```

agents

list, read-only, returns a list of all the agents of the model that are considered as "active" (i.e. all the agents with behaviors, excluding the places). Note that obtaining this list can be quite time consuming, as the world has to go through all the species and get their agents before assembling the result. For instance, instead of writing something like:

```
ask target: agents of_species my_species {
...
}
```

one would prefer to write (which is much faster):

```
ask target: list (my_species) {
...
}
```

Global built-in actions

halt

stops the simulation.

```
global {
    ...
    reflex when: length (agents) <= 0 {
            do action: halt;
    }
}
```

pause

pauses the simulation, which can then be continued by the user.

```
global {
    ...
    reflex when: time = 100 {
            do action: pause;
```

```
    }
}
```

Built-in Variables

Built-in Agent Variables

Add your content here.

Details

=== location ===

- type: Point
- description:
- deafault value:

- type: string
- description:
- deafault value:

- type: geometry
- description:
- deafault value:

- type: list
- description:
- deafault value:

- type: list
- description:
- deafault value:

- type: boolean
- description:
- deafault value:

- type: list
- description:
- deafault value:

- type: list
- description:
- deafault value:

Batch

Introduction

Add your content here.

Details

Add your content here. Format your content with:

- Text in **bold** or *italic*
- Headings, paragraphs, and lists
- Automatic links to other wiki pages

Developer Guide

Introduction to the Developer Guide

This guide is dedicated to the learning of the first steps required to develop extensions to the GAMA platform.

git quick presentation:

http://bit.ly/gamagit here is a Git clone of the current GAMA branch (unofficial at the moment):

http://eclipselabs.org/p/gama

Global view of GAMA architecture

The contents of this page has been moved to [G__GamaArchitecture]

_Xtext

Summary

Introduction

I will try to explain how to test the Xtext project here.

First, you should have a look at the Xtext on eclipse.org

"_Xtext is a framework for development of programming languages_"

And if you want to go deeper, see the nice Xtext latest doc

Then msi.gama.lang.gaml and msi.gama.lang.gaml.ui (our Xtext project)

Install Xtext

See: http://www.eclipse.org/Xtext/download

if you want a ready-to-use Xtext packaged in Eclipse, see xtext.itemis.com

(I'm using eclipse-SDK-3.6-xtext-1.0.0-win32.zip + XML Editor (WST) for GAMA)

Test GAMA Xtext project

So, Xtext let you define a grammar in an ANTLR style (not fully ANTLR, but close to it)

```
Model :
  (elements += Entity)+;
Entity:
  'entity' name=ID value=INT;
```

for example here, the += let you define a list (org.eclipse.emf.common.util.EList)

(see Gaml.xtext) You should checkout the SVN now to get the two projects msi.gama.lang.gaml and msi.gama.lang.gaml.ui Once you defined a valid grammar, you generate some code through the GenerateGaml.mwe2 file

This step will generate (in src-gen folders)

- a set of Java class representing the meta-model of your Xtext grammar,
- the ANTLR grammar,
- the ANTLR parser
- and last but not least, the code of a ready-to-use IDE (as an Eclipse plugin).

Then you simply launch a new Eclipse Application from your Xtext project, and you'll be abble to play with your new language ! *easy*

And you'll get something like:

—
If you enjoyed, you can have a look at the feature of the next release

Logic of our language

Here we define a very simple grammar in order to let us update the language dynamically.

```
// a model is made of multiple lines of code
model:
  (modeldef+=syntaxbase)*;
syntaxbase:
  (kw=keyword (expr=Expression (facets+=facet)*)?) ((b=block)|';');
facet:
  kw=facetkw expr=Expression;
block:
  '{' (body+=syntaxbase)* '}';
// different class here for highlighting, warning and proposal
keyword:
  value=ID;
facetkw:
  value=ID;
```

And for the definition of Expression , we used JavaFX ANTLR definition that we adapted for Xtext.

- http://openjfx.java.sun.com/current-build/doc/reference/ch06s03.html
- http://kenai.com/projects/openjfx-compiler/sources/soma-patch/show/doc/reference
- http://kenai.com/projects/openjfx-compiler/sources/soma-patch/show/src/share/classes/com/sun/tools/javafx/antlr

Have a look at the grammar here (msi.gama.lang.gaml/**/Gaml.xtext)

Highlighting

- Xtext documentation
- msi.gama.lang.gaml.ui/**/highlight/GamlSemanticHighlightingCalculator.java
- msi.gama.lang.gaml.ui/**/GamlUiModule.java

The AbstractGamlUiModule class is used _to register components to be used within the IDE_, here the Highlighter. In the GamlSemanticHighlightingCalculator we implement provideHighlightingFor from ISemanticHighlightingCalculator , which receive XtextResource where we'll find a reference to all nodes of the model. Then for each node, we generate a specific style via acceptor.addPosition regarding on the node's type (instanceof).

```
// for each node, if this node is a keyword (ID) apply bold font
if (abstractNode.getElement() instanceof keyword) {
    acceptor.addPosition(abstractNode.getOffset(), abstractNode.getLength(),
            DefaultLexicalHighlightingConfiguration.KEYWORD_ID);
}
```

Label provider

- Xtext documentation
- msi.gama.lang.gaml.ui/**/labeling/GamlLabelProvider.java

Here we can easily associate a type of the meta-model to the Outline UI (associated text and icons).

```
// syntaxbase : keyword.value
String text(syntaxbase ele) {
    return ele.getKw().getValue();
}
// keyword : value
```

```
String image(keyword ele) {
    return "_"+ele.getValue()+".png";
}
```

Icons should be in the msi.gama.lang.gaml.ui/icons folder.

Dynamic keywords

- msi.gama.lang.gaml/**/GamaKeywords.java

In order to give us the possibility to upgrade the language (by adding methods and data-models), we want to differentiate language's keywords from the grammar, so for all the user interface development tools we'll need to link both... This class aim to manage this link in a proper way. Somehow is a part of the MVC pattern, making link between the *Model* and the *View* .

Validation

- Xtext documentation
- msi.gama.lang.gaml/**/validation/GamlJavaValidator.java

Here, we can define different *Check* in order to validate the code (syntax analysis).

```
@Check
public void checkKeywordExists(keyword kw) {
    // kw: keyword, kw.eContainer(): syntaxbase
    if (!GamaKeywords.getInstance().isKeyword(kw.getValue(),
getContextList(kw.eContainer()))) {
        warning("unknown keyword '"+kw.getValue()+"'", GamlPackage.KEYWORD);
    }
}
```

We check if we know the keyword, and if not, we produce a warning (which could as well be an error())

We call the singleton [GamaKeywords] for testing this, passing the context list to it.

The context-list is made like:

```
private List getContextList(EObject container) {
    List lcontext = new ArrayList();
    while (container instanceof syntaxbase) {
        syntaxbase s = (syntaxbase) container;
```

```
        lcontext.add(s.getKw().getValue());
        // get the container of the parent of this syntax
        container = container.eContainer().eContainer();
    }
    return lcontext;
}
```

As we work with a graph, each node know his children and his parents, which is quite convenient to navigate through the model.

Proposal provider

- Xtext documentation
- msi.gama.lang.gaml.ui/**/contentassist/GamlProposalProvider.java

Here we add proposals for specific types depending on where we are in the code (the context).

—

—

—
_ **TODO** list_

Scope provider

- Xtext documentation
- msi.gama.lang.gaml/**/scoping/GamlScopeProvider.java

You can read on this blog post more about scoping.

"So where other frameworks create a tree , Xtext also takes care of the cross-links, hence creates a graph (a.k.a model)." ^1^

"Xtext can read and write textual models into Ecore meta models be they dynamic or static." ^2^

```
TermLiteral
    : {VarRef} value=[VarRef|ID] // here we should specify that we can get
either a ref or a new var
    | ...
```

```
    ;
```

OR

```
TermExpression
    : VarRef
    | ...
    ;
VarRef:
    value=[syntaxbase|ID];
```

warning(200): ../msi.gama.lang.gaml/src-gen/msi/gama/lang/gaml/parser/antlr/internal/InternalGaml.g:211:3: Decision can match input such as "{RULE_STRING..RULE_ID, RULE_INT..RU

As a result, alternative(s) 2 were disabled for that input

see Xtext FAQ **TODO** Work on this http://www.eclipse.org/forums/index.php?t=msg&th=167921&start=0&

Outline

- Xtext documentation
- msi.gama.lang.gaml.ui/**/outline/GamlTransformer.java

TODO Work on this

Value converter

- Xtext documentation
- msi.gama.lang.gaml/**/convert/GamlValueConverters.java
- msi.gama.lang.gaml/**/GamlRuntimeModule.java

terminal COLOR returns java.awt.Color we should specify that this returns something else than String or we'll get cast exception in ColorLiteralImpl.eSet(int, Object) : 124 "setColor((String)newValue);" **TODO** find a way to put Color in Ecore, or find another class to fit the cast cf "terminal COLOR returns java.awt.Color" Gaml.xtext:164 [http://code.google.com/p/gama-platform/source/browse/branches/msi.gama.lang.gaml/src/msi/gama/lang/gaml/Gaml.xtext#169 Gaml.xtext:169]

Formater

- Xtext documentation
- msi.gama.lang.gaml/**/formatting/GamlFormatter.java

TODO See why it's not working / Debug (CTRL+SHIFT+F)

Project wizard

- Xtext documentation

TODO Work on this / read more / code it

Launching Framework Eclipse (not Xtext)

- Launching Framework at Eclipse.org
- msi.gama.lang.gaml.ui/**/launch/GamaLaunchConfigurationDelegate.java
- org.eclipse.debug.core.model.LaunchConfigurationDelegate

Maybe interesting to reuse the Eclipse launching / debug framework... **TODO** Work on this / read more / code it

Dynamic Languages Toolkit (DLTK)

- http://www.eclipse.org/dltk/
- http://wiki.eclipse.org/A_guide_to_building_a_DLTK-based_language_IDE

TODO Read about it http://www.eclipse.org/forums/?t=msg&th=168162

—

—

—

- GIS & Agent based Modelling Architecture
- Eclipse TMF Xtext

http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.dltk/?root=Technology_Project
http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.tmf/?root=Modeling_Project
see ANTLR predicate https://wincent.com/wiki/ANTLR_predicates
see Xtext ANTLR backtracking http://wiki.eclipse.org/Xtext/FAQ#OK.2C_but_I_didn.27t_get_these_warnings_in_oAW_Xtext.C2.A0.21 http://www.eclipse.org/forums/?t=msg&th=168162 http://wiki.eclipse.org/index.php?title=Xtext/planning_0.8.0&oldid=166900#Xtext_Grammar_2

- _add support for semantic predicates_
- _add support for syntactic predicates_

TODO: http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/org.eclipse.emf.mwe/plugins/org.eclipse.emf.mwe2.language/src/org/eclipse/emf/mwe2/language/Mwe2.xtext?root=Modeling_Project&view=markup

```
DeclaredProperty:
  'var' (type=[types::JvmType|FQN])? name=FQN ('=' default=Value)?;
```

voir si [types::JvmType] pourrait etre une liste dynamic et pas poser de pb pour la gen Xtext.... http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.tmf/org.eclipse.xtext/plugins/org.eclipse.xtext.common.types/model/JavaVMTypes.ecore?revision=1.7&root=Modeling_Project&view=markup http://www.antlr.org/grammar/1153358328744/C.g

```
type_id
    :   {isTypeName(input.LT(1).getText())}? IDENTIFIER
//        {System.out.println($IDENTIFIER.text+" is a type");}
    ;
[...]
@members {
    boolean isTypeName(String name) {
        for (int i = Symbols_stack.size()-1; i>=0; i--) {
            Symbols_scope scope = (Symbols_scope)Symbols_stack.get(i);
            if ( scope.types.contains(name) ) {
                return true;
            }
        }
        return false;
    }
}
```

backtrack Xtext 1.0 : http://www.eclipse.org/forums/?t=msg&th=164984

```
import de.itemis.xtext.antlr.*
...
            fragment = XtextAntlrGeneratorFragment {
                options = auto-inject {
                    backtrack = true
                }
            }
...
            fragment = XtextAntlrUiGeneratorFragment {
                options = auto-inject {
                    backtrack = true
                }
            }
```

Develop a new plug-in

Introduction

This page details how to create a new plug-in, in order to extend the GAML language with new skills (and thus actions) and new operators. We consider that the developper version of GAMA has been installed (as detailled in [InstallProcedure14 this tutorial]).

Details

Here are detailled steps to create and configure a new plug-in.

- File > New > Project > plug-in project
- In the "New plug-in Project" / "Plug-in project" window:
  - Choose as **name** « name_of_the_plugin » (or anything else)
  - Check "Use défaut location"
  - Check "Create a Java Project"
  - The project should be targeted to run with Eclipse
  - working set (if any, in my case I had not one)
  - Click on "Next"
- In the "New plug-in Project" / "Content" window:
  - Id : could contain the name of your institution and/or your project, e.g. « irit.maelia.gaml.additions »
  - version 1.0.0
- Name « Additions to GAML from Maelia project »
- Uncheck "Generate an activator, a Java class that controls the plug-in's life cycle" ,
- Uncheck "This plug-in will make contributions to the UI"
- Check "No" when its asks "Would you like to create a rich client application ?"
- Click on "Next"

- In the "New plug-in Project" / "Templates" window:
  - Uncheck "Create a plug-in using one of the templates"
  - Click on "Finish"

Your plug-in has been created.

- Edit the file "Manifest.MF":

    - Overview pane: check « This plug-in is a singleton »
    - Dependencies pane: add (at least minimum) the two plug-ins "msi.gama.core" and "msi.gama.processor" in the "Required Plug-ins". When you click on "Add", a new window will appear without any plug-in. Just write the beginning of the plug-in name in the text field under "Select a plu-in".
    - Runtime pane:
        - In exported Packages: nothing (but when you will have implented new packages in the plug-in you should add them their)
        - Add in the classpath all the additional libraries (.jar files) used in the project.
    - Extensions pane: "Add" "gaml.grammar.addition"
    - Save the file. This should create a "plugin.xml" file.
- Select the project and in menu Project > Properties:
    - Java Compiler > Annotation Processing: check "Enable project specific settings", then in "Generated Source Directory", change ".apt_generated" in "gaml",
    - Java Compiler > Annotation Processing > Factory path: check "Enable project specific settings", then "Add Jars" and choose "msi.gama.processor/processor/plugins/mis.gama.processor.1.4.0.jar"

The plug-in is ready to accept any addition to the GAML language, e.g. skills, actions, operators To test the plug-in, 2 possibilities:

- Open the existing "gama1.4.product" and add the new plug-in
- Create a new .product file in msi.gama.application that contains all the necessary plug-ins and the one just developped.

Do not forget to add the created packages that could be used by "clients", especially all packages containing new GAML primitives in the plugin.xml of the new project.

Develop a new skill

Introduction

A new skill adds new features (attributes) and new capabilities (actions) to an agent.

How to define a new skill

A Skill is basically a Java class that:

- extends the abstract class Skill ,
- begins by the annotation @skill("name_of_the_skill_in_gaml") .

How to define new attributes

To add new attributes to agents with this skill, developpers have to define them before the class thanks to @vars and @var annotations. The @vars annotation contains a set of @var elements. In a @var element will define the name, the type and the init by default value of the parameter. For example in the [MovingSkill]:

```
@vars({
  @var(name = IKeyword.SPEED, type = IType.FLOAT_STR, init = "1.0"),
  @var(name = IKeyword.HEADING, type = IType.INT_STR, init = "rnd 359")
})
```

In order to access to these new attributes a an easy way in the GAML model, developpers have to define a getter (using @getter) and a setter (using @setter) method. For example:

```
@getter(var = IKeyword.SPEED)
public double getSpeed(final IAgent agent) {
    return (Double) agent.getAttribute(IKeyword.SPEED);
}
@setter(IKeyword.SPEED)
public void setSpeed(final IAgent agent, final double s) {
    agent.setAttribute(IKeyword.SPEED, s);
}
```

How to define a new action

An action is basically a Java method that can be used from the GAML language.
The method should by annotated by @action and the name of the action as it will be available in GAML. The developper can also define parameters for this action using the annotation @args will a set of parameters names. For example, the action goto from the [MovingSkill] is defined as follows:

```
@action("goto")
@args({ "target", "speed", "on" })
public IPath primGoto(final IScope scope) throws GamaRuntimeException {
...
}
```

It is called in GAMA models with:

```
do action: goto {
  arg target value: the_target ;
  arg on value: the_graph;
}
```

or

```
let path_followed type: path value: self.goto [target::the_target,
on::the_graph];
```

Access to parameters

To get parameters value in the Java code, two methods can be usefull:

- scope.hasArg("name_of_parametre") returns a boolean value testing whether the attribute "name_of_parameter" has been used by the modeler,
- getArg("name_param", IType) , getFloatArg("name_param_of_float") or getIntArg("name_param_of_int") return the value of the given parameter.

Warnings

Developpers should notice that:

- the method associated with an action has to return a non-void object.
- such a method can only throws GamaRuntimeException . Other exceptions should be catch in the method.

Annotations

@skill

@vars and @var

@action

@args

@getter

@setter

Tutorial 1: The Stupid model

Stupid Model GAMA 1.4

Introduction

This tutorial is based on the article: StupidModel and Extensions: A template and teaching tool for agent-based modeling platforms by Railsback, Lytinen and Grimm. It is particularly well fitted for this purpose as the complexity increase smoothly and it allows to compare GAMA to other well known platforms such as Mason, Netlogo, Swarm (java & objective-C): (see here for implementations).

Preliminary notes

Any uncertainity that have appeared within the formulation of the previously cited paper have been dealt using the netlogo implementation.

Stupid model: models list

Here the list of the different models that you will build while following the tutorial. This list is the same as in the original paper though some intermediary steps have been added in some cases. For each model we will present its purpose and an explicit formulation (from the original authors) and possibly some disambiguation or speficities due to the GAMA platform.

1. Basic stupidModel
2. Bug growth
3. Habitat cells and ressource
4. Cell and bug probes
5. Parameters and parameters displays
6. Histogram output
7. Stopping the model
8. File output
9. Randomized agent actions
10. Sorted agent actions
11. Optimal movement
12. Bug mortality and reproduction

13. Population abundance graph
14. Random normal initial size
15. Habitat data frome file input
16. [StupidTutorialModel16v14 Predators]

1. Basic stupidModel

Purpose

This is the basic Stupid Model, an extremely simple individual-based model used as a starting point for learning GAMA (or other IBM platforms).

Formulation

- The space is a two-dimensional grid of dimensions 100 x 100. The space is toroidal, meaning that if bugs move off one edge of the grid they appear on the opposite edge.
- 100 bug agents are created. They have one behavior: moving to a randomly chosen grid location within +/- 4 cells of their current location, in both the X and Y directions. If there already is a bug at the location (including the moving bug itself—bugs are not allowed to stay at their current location unless none of the neighborhood cells are vacant), then another new location is chosen. This action is executed once per time step.
- The bugs are displayed on the space. Bugs are drawn as red circles. The display is updated at the end of each time step.

Models

Model 0 : the minimal set

On the user interface of GAMA, we begin by creating a new GAMA project in our "OWN PROJECTS" repository and name it [StupidModelTutorial]. We then create a new model and name it [StupidModel1].gaml:

```
model StupidModel1
```

Now we have a good starting point for the stupid model which we will incrementally develop, by adding GAML code, in responding to the model formulation.

Model 1.1 : the environment

We are going to defined a 100x100 grid toroidal environment.

```
environment {
    grid stupid_grid width: 100 height: 100 torus: true;
}
```

Model 1.2 : Defining the agents

Here we have to define the structure of the bug agents then their behaviour:

- What is a bug agent?
  - A bug is by default situated in the space.
  - It has a red color and a circle shape defined using the **aspect** section.

```
entities {
    species bug {
    aspect basic {
        draw shape: circle color: rgb('red') size: 1;
    }
        ...
    }
}
```

Here we add an **entities** section containing the definition of species (the prototype of an agent).

- What is the behaviour of a bug?
  - It selects a destination cell or 'place' within a distance of 4 cells where there is no agent.
  - It stays at the same cell only if there is no neighbour place empty.

```
entities {
    species bug {
        ...
        reflex basic_move {
            let place value: location as stupid_grid;
            let destination value: one_of ((place neighbours_at 4) where
empty(each.agents));
            if condition: destination != nil {
            set location value: destination;
            }
        }
    }
}
```

We add a **reflex** within the **species** section which declares the behavior of the bug. This **reflex** will be executed at each time step by the agent. First, we declare a temporary variable, using the **let** command, to hold the current cell of the agent calling it place . To do so, we cast explicitly our location, built-in variable, into a cell by using the name of the environment stupid_grid . This cast is quite powerful as it translates a coordinate (location) into a cell, you will discover several powerful cast like this later on. Then, we declare the destination as a cell which is one of the empty neighbor place. Finally we check that the destination variable is not null and the agent moves. If destination is null (which means that all neighbors are already full), it stays where it is.

Model 1.3 : Instantiating bugs

- How to instantiate the 100 bugs?
  - As we have no information they will be placed randomly by the system.
  - We introduce here the **global** section which is responsible to hold global variables and process global action.

```
global {
    init {
    create species: bug number: 100;
    }
}
```

We add a **global** section which contains an **init** subsection where we call the **create** command. The **init** subsection will be executed upon the creation of the entity. Here the entity is the system itself, we call it the "world". Consequently, the bugs will be created before the start of the simulation and will be placed randomly on the default environment (the stupid_grid).

Model 1.4: Display output

We are going to define a display output to display the environment in the user interface, listing species to display.

```
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect:basic;
    }
}
```

Nota bene

In GAMA we cannot choose when to draw the agent thus the "The display is updated at the end of each time step." statement is of no interest (though it is the case).

Complete model 1

```
model StupidModel1
import "platform:/plugin/msi.gama.application/generated/std.gaml"
global {
    init {
    create species: bug number: 100;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true;
}
entities {
    species bug {
    reflex basic_move {
        let place value: location as stupid_grid;
        let destination value: one_of ((place neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }
    aspect basic {
        draw shape: circle color: rgb('red') size: 1;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
}
```

2. Bug growth

Purpose

Addition of instance variables and methods to the agents.

Formulation

- Add a second bug action: grow . Each time step, a bug grows by a #xed amount (0.1). So bugs need an instance variable for their size, which is initialized to 1.0. This action is scheduled after the move action.
- The bugs' color on the display is shaded to re#ect their size. Bug colors shade from white when size is zero to red when size is 10 or greater.

Models

Shading color

We saw previously that we can cast a string like red into a color variable.

```
var color type: rgb init: rgb('red');
```

It is also possible use the hexadecimal value, which always starts with the '#' character.

```
var color type: rgb value: #FF0000;
```

Unfortunately it is still uneasy to scale smoothly the color from white to red. Fortunately, it is possible to define one by one the three RGB components of the color using a list of three elements. In our case, we would do as follow:

```
var color type: rgb value: rgb [255, 0, 0];
```

Model 2.1: Growing using a reflex

Similarly to the first reflex we defined, you have to create a reflex that will increase a size variable by 0.1 (used after in the **aspect** section).

```
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    ...
    reflex grow {
        set size value: size + 0.1;
    }
    ...
        aspect basic {
            draw shape: circle color: color size: size;
    }
}
```

Model 2.2: Growing using a variable automatic update

In GAMA, it is possible to define an automatic variable update, for example:

```
entities {
    species bug {
    var size type: float init: 1 value: size+0.1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    ...
    }
}
```

Indeed the variable parameter **value** is evaluated at each time step while the **init** parameter is evaluated once only (upon the creation of the holding entity). The use of the _automatic update_ is convenient when the bugs growth only depend of one parameter. As soon as more parameters are involved, the implementation based on a reflex is more suitable.

Nota bene

- As you can see, it is possible to declare a list by using the brackets and the comma as separator: [elt_1, elt_2, elt_3, etc...] .

- We added a color definition within the grid section to make the grid black because bugs can be white and so invisible on a white grid. It is interesting because it shows how to define cells variable. Indeed the grid section is a bit similar to the species one. It defines the prototype of instance (cell in the case of the grid). It is possible to define here not only variables but also reflexes in the very same way as species.

Complete model 2

```
model StupidModel2
global {
    init {
    create species: bug number: 100;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];

        reflex basic_move {
        let place value: location as stupid_grid;
        let destination value: one_of ((place neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }

        reflex grow {
        set size value: size + 0.1;
    }

        aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
}
```

3. Habitat cells and resource

Purpose

Giving a behavior to the grid cells. It also illustrates how agents and cells interact.

Formulation

- The grid cells have now instance variables for their food availability and maximum food production rate. Cells also have a variable for the bug at their location.
- Food availability is initialized to 0.0, and maximum food production rate is initialized to 0.01. Each time step, food availability is increased by food production. Food production is a random #oating point number between zero and the maximum food production.
- The bug growth corresponds to the food consumption. Food consumption is equal to the minimum of (a) the bug’s maximum consumption rate (set to 1.0) and (b) the bug’s cell’s food availability.
- The food consumed by each bug is subtracted from the food availability of its cell.

Models

Giving a behavior to the cells

Here, we have to add the following variables to the cells in the grid section: maximum food production rate , food availability and food production .

```
var maxFoodProdRate type: float value: 0.01;
var food type: float init: 0.0;
var foodProd type: float value: (rnd(1000) / 1000) * 0.01;
```

The random operator works only on integer but allows us to set the precision. In this example, we have a precision of 10^-3 (as we use a range from 0 to 1000) and generate a number in the [0;0.01] range. In GAMA, whenever a species is defined, it is possible to use it as a type and then reference instance of this type. Thus, instead of repeating (stupid_grid location) , we will also add a convenient variable as follow:

```
var myPlace type: stupid_grid value: location as stupid_grid;
```

Increasing cell's food

Since we already defined the required variables, we need now to update them at each time step, either using a reflex or using the automatic update (value parameter).

```
var food type: float init: 0.0 value: food + foodProd;
```

In the value parameter, which is evaluated at each time step, we add the remaining food and the foodProd variable (which is updated at each time step thanks to value parameter too).

Bug growth

Instead of having a constant increase, we now want to have a dynamic one. The increase is defined as the minimum of maxConsumption (bug's variable to define) and food (current cell's variable). Note that this quantity has to be subtracted from the cell... Since many parameters are involved, the growth cannot be implemented using a variable automatic update so we will use a reflex.

```
reflex grow {
    let transfer value: min [maxConsumption, myPlace.food];
    set size value: size + transfer;
    set myPlace.food value: myPlace.food - transfer;
}
```

Nota Bene

As you can guess the executionner of this reflex is a bug but we could imagine that the cell do the work: it would check if there is an agent within it then transfer food (subtract to its food variable and add to the bug's size variable). Please check the data-type section for more explanation on this particular type.

Complete model 3

```
model StupidModel3
global {
    init {
    create species: bug number: 100;
    }
```

```
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: 0.01;
    var foodProd type: float value: (rnd(1000) / 1000) * 0.01;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: 1;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
        let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }

    reflex grow {
          let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }

        aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
}
```

4. Cell and bug probes

Purpose

Making model objects probeable from the display.

Formulation

- Make the bugs, and the cells, so they can be probed via mouse clicks on the display.

Models

We introduce here the inspectors. Their declarations go into the output section. We will use two kinds, one to inspect species and one to inspect the internal status of a selected agent (by clicking on it or by selecting it from the species inspector).

```
output {
    ...
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
}
```

- The **refresh_every** parameter is used to set the refresh rate of the view, here views will be updated every 5 steps. You can also change this rate during the simulation.
- Now you can see that the GAMA interface is "tab-based". You can re-arrange them as you please.
- The species inspector shows the structure of the species (variables, reflexes, etc.) and their populations (instances of the species).
- The agent inspector shows the names and the value of each attributes of a particular instanciated agent.
- You may note that the environment is also a species.

Complete model

We obtain the following model:

```
model StupidModel4
global {
    init {
    create species: bug number: 100;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: 0.01;
    var foodProd type: float value: (rnd(1000) / 1000) * 0.01;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: 1;
    var myPlace type: stupid_grid value: location as stupid_grid;

        reflex basic_move {
          let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }

        reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }

        aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
```

```
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
}
```

5. Parameters and parameters displays

Purpose

Variables as parameters and the parameter settings window.

Formulation

Make these variables into parameters that can be accessed through the settings window:

- Initial number of bugs (a model parameter)
- The maximum daily food consumption (a bug parameter)
- The maximum food production (a cell parameter).

Models

- We introduce the variables setting with this model. To do so, we add the **parameter: 'description'** (or **parameter: true** to show the variable name) state within the variable definition, as follows:

```
var numberBugs type: int init: 100 parameter: 'numberBugs';
```

- Parametrization is only available to the world's variable within the global section. Thus for bugs' and cells' parameters, we have to define global variables that will be used in the their personal variable definition. We add the following statement to the global section:

```
var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
```

- We change the foodProd variable definition within cells like this:

```
var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
```

- We also change the maxConsumption variable definition within bugs like this:

```
var maxConsumption type: float value: globalMaxConsumption;
```

Nota Bene

- It seems useless to declare a maxFoodProdRate variable, but in the case we want heterogeneous values of maxFoodProdRate , it will be very easily done.
- Parameters could not be shown by default. In this case, you have to click on the dedicated button: _Show/Hide parameters editor_.

Complete model 5

We obtain the following model:

```
model StupidModel5
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    init {
    create species: bug number: numberBugs;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
          let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
```

```
            set location value: destination;
        }
    }
    reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }

        aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
}
```

6. Histogram output

Purpose

Illustrate how to add graphs to the display and provide the ability to see the distribution of agent sizes.

Formulation

Add a histogram reflecting the distribution of bugs' size.

Models

We will now add a histogram subsection to the output one. In order to have a useable view, we would define 10 classes within the [0;100] range.

Adding the pie chart

We add the following to the output section.

```
output {
    ...
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
        data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
```

```
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
        }
    }
}
```

The bugs list is added in the **global** section. Please note again the usefulness of the cast operator in GAMA with the added variable bugs which translate a species into a list of its instantiated agents. We have moreover to initiate it after the creation of all the bugs:

```
global {
    ...
    var bugs type: list of: bug value: bug as list;
    init {
        ...
        set bugs value: bug as list;
    }
    ...
}
```

- The chart section can be of three types: **histogram** , **pie** or **series** . In all cases, we can name it and define a background color.
- Within the chart section, we can define several inputs and, for each we define a name and a value.

Nota Bene

- When using this version, you may note that we see, most of the time, only one class representing almost 100% of agents. It would be much more interesting to use adaptive classes. It is possible by changing the value expression by taking into account mean, minimum and maximum value of bugs size. To do so, you have to define the needed variables in the **global** section.
- We will do that in the complete section in order to replace the (list bug) by a global variable that will be computed once a time step.

Complete model 6

We obtain the following model:

```
model StupidModel6
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var bugs type: list of: bug value: bug as list;
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
        let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
        }
        reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }

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

```
output {
    display stupid_display {
        grid stupid_grid;
        species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
        data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
    }
    }
}
```

7. Stopping the model

Purpose

- Show how to cause a model to stop itself upon a certain condition.
- Show how to _clean up_ when a model stops.

Formulation

- The model stops when the largest bug reaches a size of 100.

Models

Model 7.1: using a world reflex

The whole system is what we call the **World** (**global** section). Thus, it makes sense that the world has to stop the execution of the simulation, thanks to the specific **halt** action. We would do that by adding the following statement in the global section of our model:

```
global {
    ...
    reflex shouldHalt when: !(empty (bugs where (each.evol > 100))) {
    do action: halt;
    }
}
```

We can see that:

- The world can have reflexes that are defined in the same way
- It is possible to add a condition to the execution of a reflex using the **when** parameter
- We could have used the **pause** command instead of the **halt** one.

Model 7.2: using bugs reflex

We could also have define a reflex within the bugs and whenever the bug attain the size of '100', it will ask the world to stop. That would be something like this:

```
entities {
    species bug {
    ...
    reflex askToHalt when: (size>=100) {
            ask target: world_species {
        do action: halt;
        }
    }
    }
}
```

- For the first time we see the **ask** command here. As it sounds, it allows an agent to ask (understand to force) an agent to execute something.
- Note that **world_species** represents the species of the world agent.

Complete model 7

We obtain the following model:

```
model StupidModel7
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var bugs type: list of: bug value: bug as list;
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: !(empty (bugs where (each.size > 100))) {
    do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
```

```
    }
}
entities {
    species bug {
        var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
        let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }
    reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }
    aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
        chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
        data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
```

```
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
    }
    }
}
```

8. File output

Purpose

Show how to write results to an output file. Illustrate how to work with list.

Formulation

- Each time step, write the minimum, mean, and maximum bug size on one line of an output #le.

Models

Writing a log file is an output thus we have to add a specific file output. In our case it would as follow:

```
output{
    ...
    file stupid_results type: text data: 'cycle: ' + (time as string)
    + '; minSize: ' + ((bugs min_of each.size) as string)
    + '; maxSize: ' + ((bugs max_of each.size) as string)
        + '; mean: ' + (((sum (bugs collect ((each as bug).size))) /
(length(bugs))) as string);
}
```

We define here:

- A name for the file: stupid_results
- A type, **text** here but it could also be **XML** or **CSV** .
- A data from an expression, here we check the evolution of two variables plus some strings to make the file more readable.

Complete model 8

We obtain the following model:

```
model StupidModel8
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var bugs type: list of: bug value: bug as list;
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: !(empty (bugs where (each.size > 100))) {
    do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
          let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }
      reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }
    aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
```

```
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
        data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
    }
    }
    file stupid_results type: text data: 'cycle: ' + (time as string)
    + '; minSize: ' + (((bug as list) min_of each.size) as string)
    + '; maxSize: ' + (((bug as list) max_of each.size) as string)
        + '; mean: ' + (((sum ((bug as list) collect ((each as bug).size))) /
(length(bug as list))) as string);
}
```

9. Randomized agent actions

Purpose

Show how to randomize the order in which agents execute an action.

Formulation

The bugs' move action is altered so that the order in which bugs execute the action is shuf#ed each time step.

Models

In GAMA, the order of execution of agents is already shuffled at every time step. So, we have nothing to do for this step.

10. Sorted agent actions

Purpose

Show how to sort a list of agents, and cause an agent action to be executed in size order.

Formulation

- The bugs' execution is un-randomized so it is executed in descending size order.

Models

Generally, the schedule with respect to which agents execute their action is randomized by a shuffle each time step. However, GAMA offers the possibility for the modeler to specify the scheduling strategy. In our context, we just have to modify the built-in global variable **scheduling_targets** in order to consider the agent size.

```
global{
    ...
    var scheduling_targets type: list value: bugs sort_by (each as bug).size;
}
```

It is important to note that only the **world** and bug agents will be executed during the simulation with above expression. We thus have to add the grid cells agents to this list in order that they are executed.

```
global{
    ...
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
}
```

We execute here all the grid cells first. Then the bugs ordered by size are executed.

Complete model 10

We obtain the following model:

```
model StupidModel10
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: !(empty (bugs where (each.size > 100))) {
    do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
          let destination value: one_of ((myPlace neighbours_at 4) where
empty(each.agents));
        if condition: destination != nil {
        set location value: destination;
        }
    }
    reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }
    aspect basic {
```

```
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
          data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
    }
    }
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
    + ((bugs min_of each.size) as string) + '; maxSize: '
    + ((bugs max_of each.size) as string) + '; meanSize: ';
    + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
}
```

11. Optimal movement

Purpose

Show how agents can identify and rank neighbor cells. Illustrate how to iterate over a list.

Formulation

- In its move method, a bug identifies a list of all cells that are within a distance of 4 cells but do not have another bug in them. (The bug's current cell is included on this list.)
- The bug iterates over the list and identifies the cell with highest food availability. The bug then moves to that cell.

Models

We have to filter cells in two ways here. First, we will remove all cells containing an agent using the **empty** command, we sort these free cell using the **sort_by** command and finally we select the one with most food using the **last** command (because of ascending order).

```
entities{
  species bug {
    ...
    reflex basic_move {
        let destination value: last (((myPlace neighbours_at 4) where
empty(each.agents)) sort_by ((each as stupid_grid).food));
        ...
    }
    }
}
```

Nota Bene

This is another possibility to get a cell with the maximum food. It is also possible to use the **with_max_of** command. We leave this to you as an exercice.

```
entities {
    species bug {
    ...
    reflex basic_move {
        let destination value: ((myPlace neighbours_at 4) where
empty(each.agents)) with_max_of each.food;
        ...
    }
```

Complete model

We obtain the following model:

```
model StupidModel11
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: !(empty (bugs where (each.size > 100))) {
    do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
```

```
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
          let destination value: last (((myPlace neighbours_at 4) where
empty(each.agents)) sort_by ((each as stupid_grid).food));
        if condition: destination != nil {
        set location value: destination;
        }
    }
    reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }
    aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
          data name: "[0;10]" value: bugs count (each.size < 10);
        data name: "[10;20]" value: bugs count ((each.size > 10) and
(each.size < 20));
        data name: "[20;30]" value: bugs count ((each.size > 20) and
(each.size < 30));
        data name: "[30;40]" value: bugs count ((each.size > 30) and
(each.size < 40));
        data name: "[40;50]" value: bugs count ((each.size > 40) and
(each.size < 50));
        data name: "[50;60]" value: bugs count ((each.size > 50) and
(each.size < 60));
        data name: "[60;70]" value: bugs count ((each.size > 60) and
(each.size < 70));
        data name: "[70;80]" value: bugs count ((each.size > 70) and
(each.size < 80));
        data name: "[80;90]" value: bugs count ((each.size > 80) and
(each.size < 90));
        data name: "[90;100]" value: bugs count ((each.size > 90) and
(each.size < 100));
        }
    }
```

```
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
    + ((bugs min_of each.size) as string) + '; maxSize: '
    + ((bugs max_of each.size) as string) + '; meanSize: '
    + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
}
```

12. Bug mortality and reproduction

Purpose

Show how to “kill” and drop objects from a model, and how to create new objects during a run.

Formulation

- When a bug’s size reaches 10, it reproduces by splitting into 5 new bugs. Each new bug has an initial size of 0.0, and the old bug disappears.
- New bugs are placed at the #rst empty location randomly selected within +/- 3 cells of their parent’s last location. If no location is identi#ed within 5 random draws, then the new bug dies.
- A new bug parameter “survivalProbability” is initialized to 0.95. Each time step, each bug draws a uniform random number, and if it is greater than survivalProbability, the bug dies and is dropped.
- This mortality action is scheduled after the bug moves and grows.
- The model stopping rule is changed: the model stops after 1000 time steps have been executed or when the number of bugs reaches zero.

Models

Multiplying

To make an agent dies and create new agent is pretty easy, it is done with the **create** command and the **die** primitive. We will do it within a new reflex.

```
reflex multiply {
    if condition: size > 10 {
    let possible_nests value: (myPlace neighbours_at 3) where
empty(each.agents);
```

```
    loop times: 5 {
        let nest value: one_of(possible_nests);
        if condition: nest != nil {
        set possible_nests value: possible_nests - nest;
        create species: bug number: 1 return: child;
        ask target: child {
            set location value: nest.location;
        }
        }
    }
    do action: die;
    }
}
```

- Note the use of the **loop** structure.

Introducing survivability

We have to introduce a new variable survivalProbability , we will set it at 0.95. We will also define a reflex that will manage the death of agents using this parameter.

```
reflex shallDie when: ((rnd(100)) / 100.0) > survivalProbability {
    do action: die;
}
```

Changing the stop condition

We have to modify the shouldHalt reflex as follow:

```
reflex shouldHalt when: (time > 1000) or (empty (bug as list)) {
    do action: halt;
}
```

Complete model

We obtain the following model:

```
model StupidModel12
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
```

```
    var survivalProbability type: float init: 0.95 parameter:
'survivalProbability';
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: (time > 1000) or (empty (bug as list)) {
    do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
    var color type: rgb init: rgb('black');
    var maxFoodProdRate type: float value: globalMaxFoodProdRate;
    var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
    var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
    var size type: float init: 1;
    var color type: rgb value: rgb [255, 255/size, 255/size];
    var maxConsumption type: float value: globalMaxConsumption;
    var myPlace type: stupid_grid value: location as stupid_grid;
    reflex basic_move {
        let destination value: last (((myPlace neighbours_at 4) where
empty(each.agents)) sort_by ((each as stupid_grid).food));
        if condition: destination != nil {
        set location value: destination;
        }
    }
    reflex grow {
        let transfer value: min [maxConsumption, myPlace.food];
        set size value: size + transfer;
        set myPlace.food value: myPlace.food - transfer;
    }
    reflex shallDie when: ((rnd(100)) / 100.0) > survivalProbability {
        do action: die;
    }
    reflex multiply {
        if condition: size > 10 {
        let possible_nests value: (myPlace neighbours_at 3) where
empty(each.agents);
            loop times: 5 {
            let nest value: one_of(possible_nests);
            if condition: nest != nil {
                set possible_nests value: possible_nests - nest;
                create species: bug number: 1 return: child;
```

```
                ask target: child {
                set location value: nest.location;
                }
            }
        }
        do action: die;
        }
    }
    aspect basic {
        draw shape: circle color: color size: size;
    }
    }
}
output {
    display stupid_display {
    grid stupid_grid;
    species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
    chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
        data name: "[0;1]" value: bugs count (each.size < 1);
        data name: "[1;2]" value: bugs count ((each.size > 1) and (each.size <
2));
        data name: "[2;3]" value: bugs count ((each.size > 2) and (each.size <
3));
        data name: "[3;4]" value: bugs count ((each.size > 3) and (each.size <
4));
        data name: "[4;5]" value: bugs count ((each.size > 4) and (each.size <
5));
        data name: "[5;6]" value: bugs count ((each.size > 5) and (each.size <
6));
        data name: "[6;7]" value: bugs count ((each.size > 6) and (each.size <
7));
        data name: "[7;8]" value: bugs count ((each.size > 7) and (each.size <
8));
        data name: "[8;9]" value: bugs count ((each.size > 8) and (each.size <
9));
        data name: "[9;10]" value: bugs count ((each.size > 9) and (each.size
< 10));
    }
    }
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
    + ((bugs min_of each.size) as string) + '; maxSize: '
    + ((bugs max_of each.size) as string) + '; meanSize: ';
    + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
}
```

13. Population abundance graph

Purpose

Show how to add a simple time series graph to a model. This graph is important for understanding results now that reproduction and mortality change the abundance of bugs.

Formulation

No change is made to the model formulation. A graph is added to display the number of bugs alive at each time step.

Models

We previously defined a histogram output, the time series one follows the same structure. We simplify change the **type** of chart. Note that we chose here to give a blue color to our serie.

```
output {
    ...
    display series_display {
        chart name: 'Population history' type: series background:
rgb('lightGray') {
            data name: 'Bugs' value: length(bugs) color: 'blue';
        }
    }
}
```

Complete model

We obtain the following model:

```
model StupidModel13
global {
```

```
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var survivalProbability type: float init: 0.95 parameter:
'survivalProbability';
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: (time > 1000) or (empty (bug as list)) {
        do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
        var color type: rgb init: rgb('black');
        var maxFoodProdRate type: float value: globalMaxFoodProdRate;
        var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
        var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
        var size type: float init: 1;
        var color type: rgb value: rgb [255, 255/size, 255/size];
        var maxConsumption type: float value: globalMaxConsumption;
        var myPlace type: stupid_grid value: location as stupid_grid;
        reflex basic_move {
            let destination value: last (((myPlace neighbours_at 4) where
empty(each.agents)) sort_by ((each as stupid_grid).food));
            if condition: destination != nil {
                set location value: destination;
            }
        }
        reflex grow {
            let transfer value: min [maxConsumption, myPlace.food];
            set size value: size + transfer;
            set myPlace.food value: myPlace.food - transfer;
        }
        reflex shallDie when: ((rnd(100)) / 100.0) > survivalProbability {
            do action: die;
        }
        reflex multiply {
            if condition: size > 10 {
                let possible_nests value: (myPlace neighbours_at 3) where
empty(each.agents);
```

```
                loop times: 5 {
                    let nest value: one_of(possible_nests);
                    if condition: nest != nil {
                        set possible_nests value: possible_nests - nest;
                        create species: bug number: 1 return: child;
                        ask target: child {
                            set location value: nest.location;
                        }
                    }
                }
                do action: die;
            }
        }
        aspect basic {
            draw shape: circle color: color size: size;
        }
    }
}
output {
    display stupid_display {
        grid stupid_grid;
        species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
        chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
            data name: "[0;1]" value: bugs count (each.size < 1);
            data name: "[1;2]" value: bugs count ((each.size > 1) and
(each.size < 2));
            data name: "[2;3]" value: bugs count ((each.size > 2) and
(each.size < 3));
            data name: "[3;4]" value: bugs count ((each.size > 3) and
(each.size < 4));
            data name: "[4;5]" value: bugs count ((each.size > 4) and
(each.size < 5));
            data name: "[5;6]" value: bugs count ((each.size > 5) and
(each.size < 6));
            data name: "[6;7]" value: bugs count ((each.size > 6) and
(each.size < 7));
            data name: "[7;8]" value: bugs count ((each.size > 7) and
(each.size < 8));
            data name: "[8;9]" value: bugs count ((each.size > 8) and
(each.size < 9));
            data name: "[9;10]" value: bugs count ((each.size > 9) and
(each.size < 10));
        }
    }
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
```

```
        + ((bugs min_of each.size) as string) + '; maxSize: '
        + ((bugs max_of each.size) as string) + '; meanSize: '
        + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
    display series_display {
        chart name: 'Population history' type: series background:
rgb('lightGray') {
            data name: 'Bugs' value: length(bugs) color: 'blue';
        }
    }
}
```

14. Random normal initial size

Purpose

Illustrate use of random number distributions. A common use of them is to induce variability among initial individuals.

Formulation

- Two new model parameters are added, and put on the parameter settings window: *initialBugSizeMean* and *initialBugSizeSD* . Values of these parameters are 0.1 and 0.03.
- Instead of initializing bug sizes to 1.0 (Sect. 2.2), sizes are drawn from a gaussian (normal) distribution de#ned by *initialBugSizeMean* and *initialBugSizeSD* .
- Negative values are very likely to be drawn from normal distributions such as the one used here. To avoid them, a check is introduced to limit initial bug size to a minimum of zero.

Models

Adding new parameters

You already know how to add variables and make them parameters.

```
var initialBugSizeMean type: float init: 0.1 parameter: 'initialBugSizeMean';
var initialBugSizeSD type: float init: 0.03 parameter: 'initialBugSizeSD';
```

Normal randomization of bug size

In order to use a gaussian distribution, we simply have to use the dedicated operator: **gauss** with values of the mean (initialBugSizeMean) and of the standard deviation (initialBugSizeSD).

```
entities {
```

```
    species bug {
        var size type: float init:
gauss({initialBugSizeMean,initialBugSizeSD});
        ...
    }
}
```

Note that other operators exist taking into account various distributions.

Check and withdraw initial negative values

We add an **init** block to check the initial value of the size attribute of the bug. If it is negative, we remplace it by 0. Note the first use of the **if** structure.

```
entities {
    species bug {
        ....
        init {
            if condition: size<0 {
                set size value: 0;
            }
        }
        ...
    }
}
```

In order to avoid division by zero trouble in the computation of the color of new bugs, we have to test whether the size is 0 before compute the color. Note the use of the triadic operator **... ? ... : ...** .

```
entities {
    species bug {
        ...
        var color type: rgb value: (size > 0) ? rgb [255, 255/size, 255/
size] : rgb [255, 255, 255];
        ...
```

Complete model

We obtain the following model:

```
model StupidModel14
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
```

```
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var survivalProbability type: float init: 0.95 parameter:
'survivalProbability';
    var initialBugSizeMean type: float init: 0.1 parameter:
'initialBugSizeMean';
    var initialBugSizeSD type: float init: 0.03 parameter: 'initialBugSizeSD';
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;
    }
    reflex shouldHalt when: (time > 1000) or (empty (bug as list)) {
        do action: halt;
    }
}
environment {
    grid stupid_grid width: 100 height: 100 torus: true {
        var color type: rgb init: rgb('black');
        var maxFoodProdRate type: float value: globalMaxFoodProdRate;
        var foodProd type: float value: (rnd(1000) / 1000) * maxFoodProdRate;
        var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
        var size type: float init:
gauss({initialBugSizeMean,initialBugSizeSD});
        var color type: rgb value: (size > 0) ? rgb [255, 255/size, 255/
size] : rgb [255, 255, 255];
        var maxConsumption type: float value: globalMaxConsumption;
        var myPlace type: stupid_grid value: location as stupid_grid;
        init {
            if condition: size<0 {
                set size value: 0;
            }
        }
        reflex basic_move {
            let destination value: last (((myPlace neighbours_at 4) where
empty(each.agents)) sort_by ((each as stupid_grid).food));
            if condition: destination != nil {
                set location value: destination;
            }
        }
        reflex grow {
            let transfer value: min [maxConsumption, myPlace.food];
            set size value: size + transfer;
            set myPlace.food value: myPlace.food - transfer;
        }
```

```
        reflex shallDie when: ((rnd(100)) / 100.0) > survivalProbability {
            do action: die;
        }
        reflex multiply {
            if condition: size > 10 {
                let possible_nests value: (myPlace neighbours_at 3) where
empty(each.agents);
                loop times: 5 {
                    let nest value: one_of(possible_nests);
                    if condition: nest != nil {
                        set possible_nests value: possible_nests - nest;
                        create species: bug number: 1 return: child;
                        ask target: child {
                            set location value: nest.location;
                        }
                    }
                }
                do action: die;
            }
        }
        aspect basic {
            draw shape: circle color: color size: size;
        }
    }
}
output {
    display stupid_display {
        grid stupid_grid;
        species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
        chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
            data name: "[0;1]" value: bugs count (each.size < 1);
            data name: "[1;2]" value: bugs count ((each.size > 1) and
(each.size < 2));
            data name: "[2;3]" value: bugs count ((each.size > 2) and
(each.size < 3));
            data name: "[3;4]" value: bugs count ((each.size > 3) and
(each.size < 4));
            data name: "[4;5]" value: bugs count ((each.size > 4) and
(each.size < 5));
            data name: "[5;6]" value: bugs count ((each.size > 5) and
(each.size < 6));
            data name: "[6;7]" value: bugs count ((each.size > 6) and
(each.size < 7));
            data name: "[7;8]" value: bugs count ((each.size > 7) and
(each.size < 8));
```

```
            data name: "[8;9]" value: bugs count ((each.size > 8) and
(each.size < 9));
            data name: "[9;10]" value: bugs count ((each.size > 9) and
(each.size < 10));
        }
    }
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
        + ((bugs min_of each.size) as string) + '; maxSize: '
        + ((bugs max_of each.size) as string) + '; meanSize: '
        + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
    display series_display {
        chart name: 'Population history' type: series background:
rgb('lightGray') {
            data name: 'Bugs' value: length(bugs) color: 'blue';
        }
    }
}
```

15. Habitat data from file input

Purpose

Show how to read spatial data in from a file.

Formulation

- Instead of assuming the space size and assuming cell food production is random (model 3), food production rates are read in from a file. The file also determines the space size.
- The file contains one line per cell, with (a) X coordinate, (b) Y coordinate, and (c) food production rate.
- Food production in a cell is now equal to the production rate read in from the #le, and is no longer random.
- Now, because we are representing real habitat with real data, it no longer makes sense for the space to be toroidal. So the space objects and movement-related methods must be modi#ed so bugs cannot move off the edge of their space.
- The input file is Stupid_Cell.Data . It has X, Y, and food production data for a grid space. X ranges from 0 to 250; Y ranges from 0 to 112. The file starts with three lines of header information that is ignored by the model.
- The cells are now displayed and colored to indicate their current food availability. Cell colors scale from black when cell food availability is zero to green when food availability is 0.5 or higher.
- A change to the bug move method is required to avoid a very strong artifact now that cell food production is no longer random. Near the start of a simulation, many cells will have exactly the same food availability, so a bug simply would move to the #rst cell on its list of neighbor cells. This is always the top-left cell among the neighbors, so bugs move constantly up and left if all the cells available to them have the same food availability. This artifact is removed by randomly shuffling the list of available cells before the bug loops through it to identify the best.

Models

Reading data from a file

GAMA allows to directly read CSV or txt files and to represent them as a matrix using casting operators:

```
var init_data type: matrix init: '../data/Stupid_Cell.Data' as file const:
true;
```

In this model, we have to find the X max (width) and the Y max (height). A way to find them is to use the **max_of** operators:

```
var width type: int init: ((init_data column_at 0) copy_between {3,
length(rows_list(init_data)) - 1}) max_of each const: true;
var height type: int init: ((init_data column_at 1) copy_between
{3,length(rows_list(init_data)) - 1}) max_of each const: true;
```

Remarks that we use the operator **copy_between** to skip the first three lines of header information. In order to set the foodProd attribute of each cell, we just have to loop over the matrix rows (from the fourth row), and set the foodProd attribute of the right cell using an **ask** command.

```
global {
    ...
    init {
        ...
        let i type: int;
        loop from: 3 to: length(rows_list(init_data)) - 1 var: i {
          let ind_i type: int value: init_data at {0,i};
          let ind_j type: int value: init_data at {1,i};
        ask target: (stupid_grid as matrix) at {ind_i,ind_j} {
          set foodProd value: init_data at {2,i};
        }
    }
    }
    ...
}
```

Cell color

In a similar way than for the bugs (model 2), we have to dynamically compute the color of the cell according to the food availability.

```
var color type: rgb value: [0,[255, food * 255 * 2] min_of each, 0] as rgb;
```

We used the **min_of** operator to ensure that the value for the green will be lower or equal to 255.

Modification of the bug behaviour

We have to modify the bug behaviour by randomly shuffling the list of available cells before the bug loops through it to identify the best. The shuffling of a list is very simple in GAMA using the shuffle operator.

```
let destination value: last ((shuffle ((myPlace neighbours_at 4) where
empty(each.agents))) sort_by ((each as stupid_grid).food));
```

Complete model

We obtain the following model:

```
model StupidModel15
global {
    var numberBugs type: int init: 100 parameter: 'numberBugs';
    var globalMaxConsumption type: float init: 1 parameter:
'globalMaxConsumption';
    var globalMaxFoodProdRate type: float init: 0.01 parameter:
'globalMaxFoodProdRate';
    var survivalProbability type: float init: 0.95 parameter:
'survivalProbability';
    var initialBugSizeMean type: float init: 0.1 parameter:
'initialBugSizeMean';
    var initialBugSizeSD type: float init: 0.03 parameter: 'initialBugSizeSD';
    var init_data type: matrix init: '../data/Stupid_Cell.Data' as file const:
true;
    var width type: int init: ((init_data column_at 0) copy_between {3,
length(rows_list(init_data)) - 1}) max_of each const: true;
    var height type: int init: ((init_data column_at 1) copy_between
{3,length(rows_list(init_data)) - 1}) max_of each const: true;
    var bugs type: list of: bug value: bug as list;
    var scheduling_targets type: list value: (stupid_grid as list) + (bugs
sort_by (each as bug).size);
    init {
        create species: bug number: numberBugs;
        set bugs value: bug as list;

        let i type: int value: 0;
        loop from: 3 to: length(rows_list(init_data)) - 1 var: i {
          let ind_i type: int value: init_data at {0,i};
```

```
          let ind_j type: int value: init_data at {1,i};
        ask target: (stupid_grid as matrix) at {ind_i,ind_j} {
          set foodProd value: init_data at {2,i};
        }
    }
    }
    reflex shouldHalt when: (time > 1000) or (empty (bug as list)) {
        do action: halt;
    }
}
environment {
    grid stupid_grid width: width height: height torus: false {
        var color type: rgb init: [0,[255, food * 255 *2] min_of each, 0] as
rgb value: [0,[255, food * 255 * 2] min_of each, 0] as rgb;
        var maxFoodProdRate type: float value: globalMaxFoodProdRate;
        var foodProd type: float;
        var food type: float init: 0.0 value: food + foodProd;
    }
}
entities {
    species bug {
        var size type: float init:
gauss({initialBugSizeMean,initialBugSizeSD});
        var color type: rgb value: (size > 0) ? rgb [255, 255/size, 255/
size] : rgb [255, 255, 255];
        var maxConsumption type: float value: globalMaxConsumption;
        var myPlace type: stupid_grid value: location as stupid_grid;
        init {
            if condition: size<0 {
            set size value: 0;
            }
        }
        reflex basic_move {
            let destination value: last ((shuffle ((myPlace neighbours_at 4)
where empty(each.agents))) sort_by ((each as stupid_grid).food));
            if condition: destination != nil {
                set location value: destination;
            }
        }
        reflex grow {
            let transfer value: min [maxConsumption, myPlace.food];
            set size value: size + transfer;
            set myPlace.food value: myPlace.food - transfer;
        }
        reflex shallDie when: ((rnd(100)) / 100.0) > survivalProbability {
            do action: die;
        }
        reflex multiply {
            if condition: size > 10 {
                let possible_nests value: (myPlace neighbours_at 3) where
empty(each.agents);
```

```
                loop times: 5 {
                    let nest value: one_of(possible_nests);
                    if condition: nest != nil {
                        set possible_nests value: possible_nests - nest;
                        create species: bug number: 1 return: child;
                        ask target: child {
                            set location value: nest.location;
                        }
                    }
                }
                do action: die;
            }
        }
        aspect basic {
            draw shape: circle color: color size: size;
        }
    }
}
output {
    display stupid_display {
        grid stupid_grid;
        species bug aspect: basic;
    }
    inspect name: 'Agents' type: agent refresh_every: 5;
    inspect name: 'Species' type: species refresh_every: 5;
    display histogram_display {
        chart name: 'Size distribution' type: histogram background:
rgb('lightGray') {
            data name: "[0;1]" value: bugs count (each.size < 1);
            data name: "[1;2]" value: bugs count ((each.size > 1) and
(each.size < 2));
            data name: "[2;3]" value: bugs count ((each.size > 2) and
(each.size < 3));
            data name: "[3;4]" value: bugs count ((each.size > 3) and
(each.size < 4));
            data name: "[4;5]" value: bugs count ((each.size > 4) and
(each.size < 5));
            data name: "[5;6]" value: bugs count ((each.size > 5) and
(each.size < 6));
            data name: "[6;7]" value: bugs count ((each.size > 6) and
(each.size < 7));
            data name: "[7;8]" value: bugs count ((each.size > 7) and
(each.size < 8));
            data name: "[8;9]" value: bugs count ((each.size > 8) and
(each.size < 9));
            data name: "[9;10]" value: bugs count ((each.size > 9) and
(each.size < 10));
        }
    }
    file stupid_results type: text data: 'cycle: '+ (time as string) + ';
minSize: '
```

```
        + ((bugs min_of each.size) as string) + '; maxSize: '
        + ((bugs max_of each.size) as string) + '; meanSize: '
        + (((sum (bugs collect ((each as bug).size))) / (length(bugs))) as
string);
    display series_display {
        chart name: 'Population history' type: series background:
rgb('lightGray') {
            data name: 'Bugs' value: length(bugs) color: 'blue';
        }
    }
}
```

16. Predators

Purpose

How to create multiple classes of agents that interact.

Formulation

- 200 predator objects are intialized and randomly distributed as the bugs are. A cell can contain a predator as well as a bug. Predators are created after bugs are.
- Predators have one method: hunt. First, a predator looks through a shuf#ed list of its immediately neighboring cells (including its own cell). As soon as the predator #nds a bug in one of these cells it “kills” the bug and moves into the cell. (However, if the cell already contains a predator, the hunting predator simply quits and remains at its current location.) If these cells contain no bugs, the predator moves randomly to one of them.
- Predator hunting is scheduled after all the bug actions.

Models

Instanciating predators

We did it already in the first model (here) thus will leave to you as an exercise.

Defining predators

We also did it in the first model (here). Although the definition of the hunting action is a bit tricky thus we will define it:

Explanations

- We select cells around of the predator (direct neighbourhood)
- We select the bugs contained inside these cells
- We select one of them randomly.
- If we have a chosen prey.
  - We check it's empty of other predators.
    - If so we go there and kill the bug.
    - If there is another predator already, nothing happens.
- If we have no place with a bug.
  - We move randomly.

Scheduling Predators

You just have to add to the scheduling_targets list (in the global section) the predator agent. Check model number 10 if you do not remember how to do.

Visualization

We just have to add the predator species to the stupid display:

Complete model

We obtain the following model:

```
     <file name="stupid_results" type="text" data="'cycle: '+ (string  time)
       + '; minSize: ' + (string (bugs min_of   each.size))
       + '; maxSize: ' + (string (bugs max_of   each.size))
       + '; meanSize: ' + (string ((sum (bugs collect ((bug   each).size))) /
(length bugs)))"/>
```

Tutorial 2: Epidemiology oriented tutorial

Epidemiology oriented tutorial (for GAMA1.4)

Introduction

This tutorial is oriented towards epidemiologists who wants to learn agent-based modelling and modeller usually working on epidemiological models. Nevertheless, "non-epidemiology-friendly" persons should be able to follow the tutorial. The only requirement to follow this tutorial is to have a basic understanding of the agent meta-model.

Details

This tutorial consists of a series of GAMA models in which a disease propagating among a population (made of explicitly represented individuals). The disease is transmitted through direct contact.

List of models

To come ....

1. Basic model

Introduction

If you are not familiar with the GAMA interface please check it out with the first steps and [InterfaceGuide14 interface guide] pages. The objectives of this first model are: - Description of the structure of a GAML model - Introduction to

Implementing the first model

GAML file organisation

A GAMA model is composed on three sections *global* , *entities* and *output* .

- **Global** : The global model variables, parameters and dynamics are defined here
- **Entities** : Here, you can defined the various entities that will interact in the model (simulation)
  - **Species** : A species is a type of individuals. In the present tutorial there are only "animal" individuals but it could go further by considering two types of animal (cats and dogs for instance).
- **Output** : Visualisation of the model is not mandatory but if you want to define one, here it is.

Conceptual model

A disease is propagated among a population of *animal* through direct contact. An *animal* can be whether *susceptible* , *infected* or *dead* . *Animal* are located within an homogeneous space where they move randomly. The space will be represented as a grid of cells and the random movement will be the relocation of an *animal* to a neighbouring cell of the present one.

Implemented model

Skeleton

First of all, start a new project and create a new model file. You will create a new model file for each step of this tutorial (when you click on the next page). You should obtain something similar as:

```
/**
*  model_name
*  Author: some_author
*  Description:  a brief description of your model

*/
model model_name
global {
    /** Insert the global definitions, parameters and actions here */
}
environment {
    /** Insert the grid or gis environment(s) in which the agents will be
located */
}
entities {
    /** Insert here the definition of the species of agents */
}
output {
    /** Insert here the definition of the different outputs shown during the
simulations */
}
```

You can right click on the code and select _load default_ which will initiate the simulation. As you can see the view is then switched to the simulation one but there is not much to do as the model empty.

Setting up the space

In GAMA, a continuous environment is defined by default. In order, to simplify the current model, we will use a grid. Consequently, we define an grid having the same size as the environment (we will forget the notion of continuous environment until further notice).

```
environment width: 10  height: 10 {
    grid space width: 10 height: 10 neighbours: 8 torus: true{}
}
```

This defines a 10x10 grid. If you reload, the model you do not see more. You have to define the visualisation of the grid. To do so you need to define a grid aspect and add to the *output* section. For example:

```
environment width: 10  height: 10 {
    grid space width: 10 height: 10 neighbours: 8 torus: true{
        aspect default{
                        draw shape: shape  color: rgb('lightGray') ;
                }
    }
}
output {
    display my_display{
    species space aspect:default;
    }
}
```

The **aspect** of a cell or an agent defines how it should be visualised. Here, we simply to define the "default" one as representing a cell shape (a rectangle) and its color as "lightGray". A **display** represents a panel of the simulation interface. In the present case, we register the "space" using its default aspect in order to visualise it.

Setting up the animals

Animal species

As said before, the various model entities are define within the *entities* section as *species* . For example, a basic animal could be defined as having a status, a myPlace (technical variable), a location (technical variable) as characteristics and a movement as single behaviour or dynamics:

```
entities{
    species animal {}
        var status type:string init:'S';
        var myPlace type: space init: one_of (space as list);
        var location type:point init:  myPlace.location;
        reflex move{
            let destination type:space value: one_of(myPlace neighbours_at 1);
            set myPlace value: destination;
            set location value: myPlace.location;
        }
        aspect default {
            draw shape: circle at: location  color:rgb('green') size: 0.5;
            }
```

Here, we can see that an agent variable is defined by two mandatory elements: a name (*status*), its type (*string*) while the *init* facet is not. The dynamics of this agent is an unconditional movement (the *move* reflex which has no condition). Practically, a temporary variable (*destination*) is used to store a random cell selected within the neighbours (of a 1 degree) of the current agent's cell. *myPlace* is the cell where is located the agent and *location* is its exact location within the continuous environment. Similarly to the grid (cells), we have to define an *aspect* to define the visualisation of animals agents. We also have to update the display in order to visualise *animals* , similarly to the grid we had the following line in the output section:

```
species animal aspect:default;
```

Nota : myPlace.location shows that we access to the location variable (or characteristics) of the myPlace cell. More generally the "." operator allows to access a cell or an agent variable.

Creating animals

If you reload the simulation, no animal are visible. Indeed, we have to define how many and when we went to create them. As they cannot create themselves, their creation will be defined in the global section. In addition, the number of animals will be a parameter of the model.

```
global
{
    var dimensions type: int init: 20 max: 400 min: 10 parameter: 'Width and
height of the environment:' category: 'Environment' ;
    var number_of_animal type: int init: 20 min:2 parameter:"initial number of
animals";
    init{
        create species: animal number: number_of_animal;
    }
}
```

- A **parameter** is simply a *global* variable with an additional facet, *parameter* , which defines the name that will be shown to the user during simulation. The *category* facet is accessory and just allow to group parameters together in the GUI. In addition, you can see that we added a variable "dimensions" to define it as a parameter.

- In the global section, **Init** is a reflex that is computed at the beginning of the simulation. Here, we use a built-in action, *create* , to create a given number of animals.

Tune the model

It is now time for you to play a bit with the model by adding variables and parameters or changing the color or shapes of the animals. You should also play with the GUI if you are not familiar with it.

Complete model

Only for this first model the complet sourcecode will be shown here:

```
model epidemio1
global
{
    var dimensions type: int init: 20 max: 400 min: 10 parameter: 'Width and
height of the environment:' category: 'Environment' ;
    var number_of_animal type: int init: 20 min:2 parameter:"initial number of
animals";
    init{
        create species: animal number: number_of_animal;
    }
}
entities{
    species animal {
        var status type:string init:'S';
        var myPlace type: space init: one_of (space as list);
        var location type:point init:  myPlace.location;
        reflex move{
            let destination type:space value: one_of(myPlace neighbours_at 1);
            set myPlace value: destination;
            set location value: myPlace.location;
        }
        aspect default {
            draw shape: circle at: location  color:rgb('green') size: 0.5;
        }
    }
}

environment width: 10  height: 10 {
    grid space width: 10 height: 10 neighbours: 8 torus: true{
        aspect default{
                        draw shape: shape  color: rgb('lightGray') ;
              }
    }
}
output {
    display my_display{
    species space aspect:default;
    species animal aspect:default;
```

```
    }
}
```

Tutorial 3: GIS tutorial

Introduction to the tutorial

summary Tutorial: Road traffic in a city

Introduction

This tutorial has for goal to present the use of GIS data and complex geometries. In particular, this tutorial shows how to load gis data, to agentify them and to use a network of polylines to constraint the movement of agents. All the files related to this tutorial (shapefiles and models) are available here . The model built in this tutorial concerns the study of the road traffic in a small city. Two layers of GIS data are used: a road layer (polylines) and a building layer (polygons). The building GIS data contain an attribute: the 'NATURE' of each building: a building can be either 'Residential' or 'Industrial'. In this model, people agents are moving along the road network. Each morning, they are going to an industrial building to work, and each night they are coming back home. Each time a people agent takes a road, it wears it out. More a road is worn out, more a people agent takes time to go all over it. The town council is able to repair some roads. [Road_traffic_image] < http://gama-platform.googlecode.com/files/road_traffic.png>

Road traffic in a city: model list

This tutorial is composed of 8 models. For each model we will present its purpose, an explicit formulation and the corresponding GAML code.

1. Loading of GIS data (buildings and roads)
2. [RoadTraficModel2v14 Integration of people agents]
3. [RoadTraficModel3v14 Movement of the people agents]
4. [RoadTraficModel4v14 Definition of weight for the road network]
5. [RoadTraficModel5v14 Dynamic update of the road network]
6. [RoadTraficModel6v14 Definition of a chart display]
7. [RoadTraficModel7v14 Automatic repair of roads]
8. [RoadTraficModel8v14 Complex repair of roads]

1. Loading of GIS data (buildings and roads)

Purpose

Illustrate how to load GIS data (shapefiles) and to read attributes from GIS data.

Formulation

- Load, agentify and display two layers of GIS data (building and road)
- Read the 'NATURE' attribute of the building data: the buildings of 'Residential' type will be colored in gray, the buildings of 'Industrial' type will be color in blue.

Model

Entities

In this first model, we have to define two species of agents: the **building** agents and the **road** ones. These agents will not have a particular behaviour. They will be just displayed. Thus, we just have to define an aspect for the agents (see [Species Aspects] in the Modeling guide). In this model, we want to represent the geometry of the agent, we then use the value **geometry** of the facet **shape** for the command **draw** . Concerning the **building** agent, we have to add in addition to the color a second attribute: the type of the building ('Residential' or 'Industrial').

Parameters

GAMA allows to automatically read GIS data that are formatted as shapefiles. In order to let the user chooses his/her shapefiles, we define three parameters (string). One allowing the user to choose the road shapefile, one allowing him/her to choose the

building shapefile, and, at last, one allowing him/her to choose the bounds shapefile. We will come back later on the notion of "bounds" in GAMA.

Agentification of GIS data

In GAMA, the agentification of GIS data is very straightforward: it only requires to use the **create** command with the **from** facet to pass the reference of the shapefile. Each object of the shapefile will be directly used to instantiate an agent of the specified species. The reading of an attribute in a shapefile is also very simple. It only requires to use the **with** facet: the argument of this facet is a dictionary of which the keys are the names of the agent attributes and the value the **read** command followed by the name of the shapefile attribute ('NATURE' in our case).

Environment

Building a GIS environment in GAMA requires nothing special, just to define the bounds of the environment, i.e. the square envelop of the GIS data. It is possible to use a shapefile to automatically define it. In this model, we use a specific shapefile to define it. However, it would been possible to use the road shapefile to define it.

Display

We have to define a diplay for the road and building agents. We use for that the classic **species** markup.

Complete model

How to do

How to do?

Introduction

A set of tutorials on specific points

How To Import Image and Geographic data

Introduction

We present here how to import images and (raster and vector) geographic data in GAMA. The aim of this tutorial is to be able to import georeferenced data from several sources. We will take the example of the Adour-Garonne basin. We want to import in the simulation:

- digital elevation model data from a raster image
- management unit data from vector data
- river data from vector data

Details

We detail various ways to import images and geographic data depending on their nature (raster/vector) and their number:

- Import 1 raster image
- Import vector data
- Import both raster and vector data

Image and Raster data

As Background

To use an image only as a background of the simulation, in a passive way, we only write that we want to display it in the output section of the model:

```
output {
  display HowToImportRaster {
      image name: 'Background' file:'../images/mnt/mntsect211.png';
  }
}
```

In a grid

To use the data contained in the image (in particular the pixel color in a raster image), we need to transform the image in a matrix in order to manipulate it. Afterwards we can create a grid environment using this matrix in order to ease the interactions between other agents of the simulation and the image (via the grid). The transformation from the image to the matrix is made automatically in GAMA with the operator as_matrix . Note that as_matrix takes a Point as right operand; this point specifies the number of rows and columns of the matrix):

```
global {
  var mapColor type: matrix ;

  init {
      set mapColor value: file('../images/mnt/mntsect211.png') as_matrix
{10,10} ;
  }
}
```

The above code transforms the image located at '../includes/mnt/mntsect211.png' in a 10x10 matrix of colors. When a matrix cell includes more than one pixel of the pixel, the average of the pixels color is affected to the cell. Next we create the grid using this matrix to initialize the color of each cell:

```
global {
  var mapColor type: matrix ;
```

```
  init {
      set mapColor value: file('../images/mnt/mntsect211.png')   as_matrix
{10,10} ;
      ask target: cell as list {
        set color value: mapColor at {grid_x,grid_y} ;
      }
    }
}
environment {
  grid cell width: 10 height: 10;
}
```

Note that the grid and the matrix must of course have the same dimensions: each grid cell takes the color of the corresponding matrix cell (representing the image). It is now easy for agents to use image information. For example, if we want an agent to be located on white places of the image, we can write:

```
entities {
  species izard {
      init{
        set location value: one_of(cell as list where ( each.color =
rgb('white')) ) ;
      }
  }
}
```

As agent's aspect

A nice way to represent an agent is to use an image. For this purpose, we need to add an aspect subsection in the species definition.

```
entities {
  species izard {
      init{
        set location value: one_of(cell as list where ( each.color =
rgb('white')) ) ;
      }
      aspect image{
        draw image: '../images/icons/izard.gif';
      }
  }
}
```

We then precize that we want to display the agent with the defined aspect.

```
output {
```

```
  display HowToImportRaster {
      grid cell;
      species izard aspect: image;
  }
}
```

Example

Finally, we present a complete example. We want to import an MNT (Modèle Numérique de Terrain or Digital Elevation Model) data into the simulation and randomly locate some izard agents in the most elevated places of the area (i.e. in the white places of the MNT). We detail in the following the various sections of the model.

```
model HowToImportRaster
```

We first initialize the grid and create the izard agents

- we use the as_matrix operator to transform an image file into a matrix of colors
- we then set the color built-in attribute of the cell to the value of the corresponding matrix cell.

Note that the two constants heightImg and widthImg represent the actual size of the raster image in number of pixels. With the factorDiscret parameter, the user can change the discretization factor to have a more or less fine grid. It is (for the moment) technically impossible to have a factorDiscret of 1 because it would create a grid with more than 30 millions of cells. We create a grid as environment with the same dimensions as the matrix in which we want to store the image. Note that the height (resp. the width) of the grid corresponds to the number of rows (resp. of columns) of the matrix:

- in the creation of a matrix: as_matrix {widthImg/factorDiscret,heightImg/ factorDiscret} ;)
- in the creation of the grid: grid cellule width: widthImg/ factorDiscret height: heightImg/factorDiscret;

```
global {
  // Constants
  const heightImg type: int init: 5587;
  const widthImg type: int init: 6201;

  // Parameters
  var mntImageRaster type: file init: '../images/mnt/mntsect211.png'
parameter: 'MNT file' category: 'MNT' ;
```

```
  var factorDiscret type: int init:20 parameter:'Discretization factor'
category:'Environment';

  var nbIzard type: int init: 25 parameter: 'Nb of Izards' category: 'Izard';
  var izardShape type: file init:'../images/icons/izard.gif' category:
'Izard';

  // Local variable
  var mapColor type: matrix ;

  init {
      set mapColor value: mntImageRaster as_matrix {widthImg/
factorDiscret,heightImg/factorDiscret} ;
      ask target: cell as list {
        set color value: mapColor at {grid_x,grid_y} ;
      }
      create species: izard number: nbIzard;
  }
}
environment {
  grid cell  width: widthImg/factorDiscret height: heightImg/factorDiscret;
}
```

Izard agents that have been created in the global section are located on one random cell among the list of empty cells (empty(each.agents)) and with a white color (each.color = rgb('white')).

```
entities {
  species izard {
      init{
    set location value: one_of(cell as list where (empty(each.agents) and
each.color = rgb('white')) ) ;
      }
      aspect basic{
          draw shape: square color: rgb('orange') size: 1;
      }
      aspect image{
    draw image: izardShape.path;
      }
  }
}
```

We finally display:

- the grid
- the original MNT image as background
- izard agents

We can thus compare the original MNT image and the discretized image in the grid. For cosmetic need, we can choose to not display the grid.

```
output {
  display HowToImportRaster {
      grid cell;
      image name: 'Background' file: mntImageRaster.path;
      species izard aspect: image;
  }
  inspect name: 'Species' type: species refresh_every: 5;
}
```

Vector data

Introduction to vector data

When we produce a shapefile with dedicated software or get such data from external ressource, we often get several files. Here are the files that are mandatory for the importation in GAMA:

- .shp: this file contains the geometric shapes
- .shx: this file contains the shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly
- .dbf: this file contains the attribute table

Note that a .prj file is often provided with the shapefile. This file contains information about the coordinate system and projection of the shapefile. In the context of data expressed in a Geographic Coordinate System (longitude, latitude), GAMA will automatically try to applied the adequate UTM (Universal Transverse Mercator) projection in order to express the coordinate of the object in meters.

How to create agents from the shapefile

In order to import a shapefile and use its data, we will create agents that will be instantiated with the geometry and the attributes of the elements of the shapefile: each point, polyline and polygon of the shapefile will become an agent in GAMA. We consider that each layer of the GIS (i.e. each shapefile) contains elements that can be represented by only one species of agent. For example, if we want to import data of the buildings of a city and of its streets, we will import two different shapefiles: one containing buildings and the other one the streets. In this part, we plan to import the data of the water management units of the Adour-Garonne basin. We thus begin by creating a new agent species:

```
entities {
  species ManagementUnit{
      aspect basic{
        draw shape: geometry;
      }
  }
}
```

We create as many instances of [ManagementUnit] agent as there are geometric elements in the shapefile:

```
global {
  var ManagementUnitFile type: file init: '../images/ug/UGSelect.shp';

  init {
      create species: ManagementUnit from: ManagementUnitFile.path ;
  }
}
```

It is important to note that the environment is by default a 100x100 square. In most of the shapefiles, this space is not adapted to display the shapes. We thus have to fix manually the bounds of the environment. When we only use 1 shapefile, we can use this file to bound the environment: GAMA will consider the minimal bounding rectangle of the data contained in the shapefile and fix the bounds of the environment with its dimensions.

```
environment bounds: ManagementUnitFile.path;
```

We can then display the created agents as usual:

```
output {
  display HowToImportVectorial {
      species uniteDeGestion aspect: basic;
  }
  inspect name: 'Species' type: species refresh_every: 5;
}
```

How to import attributes data from the shapefile

The shape built-in attribute of the agents created using the shapefile ([ManagementUnit] agents in the example) gets automatically the geometry of the corresponding geometric element of the shapefile. But the geometric elements of the shapefile can have additional attributes. To get them in the created agents, we need to:

- Add additional attributes to the agent
- Assign each agent attribute by reading the corresponding attribute in the shapefile.

In the example, we thus add three additional attributes. These attributes must have the same type as the attributes in the shapefile:

```
entities {
  species managementUnit{
      var MUcode type: int;
      var MULabel type: string;
      var pgeSAGE type: string;

      aspect basic{
          draw shape: geometry;
      }
  }
}
```

The reading of these attributes is made during the creation of the agents. 'Code_UG', 'Libelle_UG' and 'PGE_SAGE' are the names of the different attributes in the shapefile.

```
global {
  var ManagementUnitShape type: file init: '../images/ug/UGSelect.shp';

  init {
      create species: managementUnit from: ManagementUnitShape.path
        with: [MUcode::read('Code_UG'), MULabel::read('Libelle_UG'),
pgeSAGE::read('PGE_SAGE')] ;
    }
}
```

How to use shapefile as background

We can also use a shapefile as background of the simulation display. It becomes then a passive image (GAMA does not create any agent). Note that we have to use the gis: keyword to import a shapefile instead of the file: keyword used for raster image.

```
output {
  display HowToImportVectorial {
      image name: 'GISBackground' gis: ManagementUnitShape.path color:
rgb('blue');
  }
}
```

Complete example

```
model HowToImportVectorial
global {
  var ManagementUnitShape type: file init: '../images/ug/UGSelect.shp'
parameter: 'Management unit:' category: 'GIS' ;
```

```
  init {
      create species: managementUnit from: ManagementUnitShape.path
    with: [MUcode::read('Code_UG'), MULabel::read('Libelle_UG'),
pgeSAGE::read('PGE_SAGE')] ;
  }
}
environment bounds: ManagementUnitShape.path {}
entities {
  species managementUnit{
      var MUcode type: int;
      var MULabel type: string;
      var pgeSAGE type: string;

      aspect basic{
          draw shape: geometry;
      }
  }
}
output {
  display HowToImportVectorial {
      image name: 'GISBackground' gis: ManagementUnitShape.path color:
rgb('blue');
      species managementUnit aspect: basic;
  }
  inspect name: 'Species' type: species refresh_every: 5;
}
```

Raster and Vector data

Introduction

We combine the two previous models to import:

- the MNT raster data
- the management unit shapefile
- the river shapefile

We also create izards agents on the higher places of the MNT.

Details

The only difficulty of this model is the definition of environment bounds in order to display correctly the various raster and vectorial data layers. As the raster data are not fully georeferenced, contrarily to vectorial ones, the environment bounds will be designed in order to display correctly the raster data. We will thus hand-made a box shapefile bounding the raster image, georeference it thanks to the .pgw file associated to the .pgn image and use it as bounds. This file is the boundsMNT.shp in our case. GAMA will then automatically display the vectorial data: in particular some translation and homothetic transformations will be done in order to display in the correct location these agents.

Complete example

```
model HowToImportRasterAndVectoriel
global {
  // Constants
  const heightImg type: int init: 5587;
  const widthImg type: int init: 6201;
  const boundsMNT type: file init: '../images/mnt/boundsMNT.shp';

  // Parameters related to the MNT
  var mntImageRaster type: file init: '../images/mnt/mntsect211.png'
parameter: 'MNT file' category: 'MNT' ;
```

```
  var factorDiscret type: int init:20 parameter:'Discretization factor'
category:'MNT';

  // Parameters related to the Management units
  var ManagementUnitShape type: file init: '../images/ug/UGSelect.shp'
parameter: 'Management unit:' category: 'MU' ;

  // Parameters related to the water network
  var waterShape type: file init: '../images/reseauHydro/reseauEau.shp'
parameter: 'Rivers shapefile' category: 'Water';
  // Parameters related to izard agents
  var nbIzard type: int init: 25 parameter: 'Nb of Izards' category: 'Izard';
  var izardShape type: file init:'../images/icons/izard.gif' parameter: 'Izard
Shape' category: 'Izard';
  // Local variable
  var mapColor type: matrix ;

  // Initialization of grid and creation of the izard agents.
  // Creation of managmentUnit and rivers agents from the corresponding
shapefile
  init {
      create species: managementUnit from: ManagementUnitShape.path
    with: [MUcode::read('Code_UG'), MULabel::read('Libelle_UG'),
pgeSAGE::read('PGE_SAGE')] ;

      create species: river from: waterShape.path;

      set mapColor value: mntImageRaster as_matrix {widthImg/
factorDiscret,heightImg/factorDiscret} ;
      ask target: cell as list {
    set color value: mapColor at {grid_x,grid_y} ;
      }
      create species: izard number: nbIzard;
  }
}
// We create a grid as environment with the same dimensions as the matrix in
which we want to store the image
// The environment bounds are defined using the hand-made boundsMNT shapefile.
// This shapefile has been created as a georeferenced bounding box of the MNT
raster image, using information of the .pgw file
environment bounds: boundsMNT.path {
  grid cell  width: widthImg/factorDiscret height: heightImg/factorDiscret;
}
entities {
  species river {
      aspect basic{
    draw shape: geometry color: rgb('blue');
      }
  }

  species managementUnit{
```

```
      var MUcode type: int;
      var MULabel type: string;
      var pgeSAGE type: string;

      aspect basic{
        draw shape: geometry;
      }
  }
  species izard {
      init{
    set location value: one_of(cell as list where (empty(each.agents) and
each.color = rgb('white')) ) ;
      }
      aspect basic{
        draw shape: square color: rgb('orange') size: 5000;
      }
      aspect image{
        draw image: izardShape.path size: 5000;
      }
  }
}
// We display:
// - the original MNT image as background
// - the grid representing the MNT
// - izard agents
// - the management unit shapefile
// - the river shapefile
// We can thus compare the original MNT image and the discretized image in the
grid.
// For cosmetic need, we can choose to not display the grid.
output {
  display HowToImportVectorial {
      image name: 'Background' file: mntImageRaster.path;
      grid cell;
      species managementUnit aspect: basic transparency: 0.5;
      species river aspect: basic;
      species izard aspect: image;
  }
  inspect name: 'Species' type: species refresh_every: 5;
}
```