Structure of a model/program

Program myFirstModel	model myFirstModel
global Defines all global variables, model initialization and global behaviors.	global { // global variables declaration // initialization of the model // global behaviors }
species mySpecies1 Defines variables, behaviors and aspects of agents of the species.	species mySpecies1 { // attributes, initialization, behaviors and aspects of a species }
experiment expName Defines the way the model will be executed Includes the type of the execution, which global parameters can be modified, and what will be displayed during simulation	experiment expName { // Defines the way the model is executed, the parameters and the outputs. }

Comments

Block comments	/* A block comment starts with the an opening symbol. The comment runs until the closing symbol below. */
Inline comments	// This is an inline comment. // The // symbol have to be repeated before each line.

Use of an external model

Use a model (i.e. its species and global variables and behaviors) defined in another file.	// this should be after the model statement import "path_to_model/model2.gaml"

Primitive types

Integer number *# value between -2147483648 and 2147483647*	int
Real number *# absolute value between 4.9*10^-324 and 1.8*10^308*	float
String *# explicit value: "double quotes" or 'simples quotes'*	string
Boolean value *# 2 values:* ***true, false***	bool

Other types

pair #with the two elements of undefined types **pair** #with two elements of types type1 and type2 #explicit value using :: symbol: e.g. 1::"one"	pair pair<type1, type2>
color #explicit value: rgb(255,0,0) for red. (3 components: Red, Green, Blue)	rgb
point #explicit value: {1.0, 3} or {1.0, 3, 6}. #Internal representation with 3 coordinates.	point

Variable or constant declaration, affectation

Declaration of a global variable or an attribute # Global variables and species attributes can be declared with or without initial value. **Declaration of a local variable** ***# explicit declaration of the type*** *# (if the type of the affected value is different, this value is automatically casted to the declared type)* **Declaration of a global variable or an attribute with a dynamic value** *# value computed at each simulation step* *# value computed each time the variable is used.* **Declaration of a global variable or an attribute with additional options** *# a variable with a minimum and maximum value (if the variable is assigned with a value greater than the max, it is set to the maximum value)* *# a variable with only some possible values.*	// Global variables or species attributes **int an_int;** **string a_string <- "my string";** // Local variables **float** a_float **<-** 10.0; // Global variables or species attributes with dynamic value // inc_int is incremented by 1 at each simulation step **int** inc_int **<-** 0 **update: inc_int + 1;** // random_int has a new random value each time it is used: **int** random_int **-> { rnd(100) };** // a_proba can only take value between 0.0 and 1.0 with a step of 0.1 **float** a_proba <- 0.5 min: 0.0 max: 1.0 step: 0.1; // a_str can only take 3 values “blue”, “red”, “green” **string** a_str **<-** "blue" **among**: ["blue", "red", "green"];
Definition of a constant	float pi <- 3.14 const: true;
Affectation of a value to a variable Variable ← value or computed expression	// Affectation of a value to an existing variable an_int <- 0;

Display variables

Display ("Text: ", Expression) **Display** Expression :- Expression Value	// Expression will be implicitly casted to a string // the + symbol is the string concatenation operator write "Text: " + Expression ; write sample(Expression);

Conditionals

<table>
<tr><td>

If Condition1 **then**
| actions
└─

If Condition1 **then**
| action1
Else
| other actions
└─

If Condition1 **then**
| action1
Else If Condition2 **then**
| action2
Else
| other actions
└─

composition of Boolean expressions

</td><td>

```
if (expressionBoolean = true) {
        // block of statements
}

if (expressionBoolean = true) {
        // block 1 of statements
} else {
        // block 2 of statements
}

if (expressionBoolean = true) {
        // block 1 of statements
} else if (expressionBoolean2 != false) {
        // block 2 of statements
} else {
        // block 3 of statements
}
// equal: = ; not equal: != (e.g. (var1 != 3) )
// Comparison: <, <=, >, >= (e.g. (var2 >= 5.0) )
// logic operators : not (or !), and, or (e.g. (cond1 and not(cond2)) )
```

</td></tr>
<tr><td>

Conditional affectation
affectation depending of the condition value (if true, affects the value before the : symbol)

</td><td>

```
string s <- (expressBoolean = true) ? "is true" : "is false";
```

</td></tr>
<tr><td>

Switch statement is a more advanced conditional. It be used with any type of data.
switch expression

  match an_expression
    | actions
    └─

  match_one a_list_expression
    | actions
    └─

  match_between a_list_expression
    | actions
    └─

  match_regex a_string_expression
    | actions
    └─

  default
    | actions
    └─
└─

*# All the match and default lines are tested, until reaching a break statement (**break** or **return**)*

</td><td>

```
switch res {
// match to test the equality
        match 0 {
                // block of statements
        }

// match_between for a test on a range of numerical value
        match_between [-#infinity,0] {
                // block of statements
        }

// match_one for at least one equality
        match_one [1,2,3,4,5] {
                // block of statements
        }

        default {
                // block of statements
        }
}

switch "FOO" {
// match to a regular expression. Note the break statement, making the switch interrupted if the match_regex "[A-Z]" is fulfilled.
        match_regex "[A-Z]" {
                write "MAJ";
                break;
        }
        default {
                write "NOT MAJ";
        }
}
```

</td></tr>
</table>

Loops

Repeat n times actions	`loop times: 10 {` `write "loop times";` `}`
For index **from** 0 **to** n **Do** actions *# the index does not need to be declared before this loop*	`loop i from: 1 to: 10 step: 1 {` `write "loop for " + i;` `}`
While Condition **Repeat** actions	`int j <- 1;` `loop while: (j <= 10) {` `write "loop while " + j;` `j <- j + 1;` `}`
For each element **of** a container **Do** actions *# the variable containing each element does not need to be declared before this loop*	`list<int> list_int <- [1,2,3,4,5,6,7,8,9,10];` `loop i over: list_int {` `write "loop over " + i;` `}`
For each agent of a species or a set of agents **Do** actions executed **in the context of the agent** *# in the ask, **self** keyword refers to the current agent (i.e. each agent of the species parameter of the ask) and **myself** refers to the agent calling the ask statement.*	`ask mySpecies2 {` `// statements` `}` `ask list_agent {` `// statements` `}`

Declaration of a procedure / an action

# Procedures and functions are very similar in their definition. The only difference is that a function has the returned type (instead of the keyword action) and it returns a value.	
Procedure ProcedureName actions	`action myAction {` `write "Action without param";` `}`
Procedure ProcedureName (pd1, pd2) actions	`action myActionWithParam( int int_param,` `string my_string <- "default value") {` `write my_string + int_param;` `}`

Call of a procedure / an action

Call ProcedureName **Call** ProcedureName (pa1, pa2, pa3) *# if a parameter has a default value, it can be omitted when calling the action. It will thus have the default value.*	`do myAction();` `do myActionWithParam(3, "other string");` `do myActionWithParam(3); // the second parameter has its default value`
# if the procedure has been defined in another species, the current agent has to ask an agent of this species to call the procedure.	`ask an_agent {` `do proc(3);` `}`

Declaration of a function

Function FunctionName **: type** actions return value	int myFunction { return 1+1; }
Function FunctionName (pd1, pd2) **: type** actions return value	int myFunctionWithParam(int i, int j <- 0){ return i + j; }

Call of a function

Variable ← FunctionName ()	// the current agent calls the function int i <- myFunction(); int j <- self.myFunction(); // The current agent calls a function with parameters int l <- myFunctionWithParam(1); int m <- myFunctionWithParam(1,5);
Variable ← FunctionName (pa1, pa2) *# if a parameter has a default value, it can be omitted when calling the action. It will thus have the default value.* *# if the function has been defined in another species, the current agent has to ask an agent of this species to call the function.*	// another agent calls a function with parameters int n <- an_agent.myFunctionWithParam(1,5);

List, map and matrix

Declaration and explicit initialization of list, map and matrix variables.	list<int> list_int <- [1,2,3,4,5]; map<int,string> map_int <- map([1::"one",2::"two"]); matrix<int> m <- matrix([[1,2],[3,4]]);
Incremental creation of lists and maps	// Add 7 at the end of the list add 7 to: list_int; // Add the pair 6::"six" to the map add "six" at: 6 to: map_int;
# Replacement of an element from list or matrix. *# In map, we can replace the value associated to a key.*	put 8 at: 5 in: list_int; put 7 at: {0,0} in: m;
Access to elements *# List access using the index, map access using the key, matrix access using coordinates in the matrix.* *# the first element of a list has an index of 0.*	// Access of an list element out of bounds will throw an error, Access to the value associated to a non-existing key will return nil list_int[1] map_int[2] m[{1,1}]
Loop over elements of a list, map, matrix	// loop over values of a list loop i over: list_int { }
# Loop over maps have to be done on keys, values or pairs list	// loop over values of the map (similar with keys and pairs) loop v over: map_int.values { }

Definition of a species

Species SpeciesName Definition of the set of attributes **init** statements **behavior** behaviorName statements **aspect** aspectName statements to draw the agents *# built-in attributes: name, shape, location…*	`species mySpecies1 {` `  int s1_int;` `  float energy <- 10.0;` `  init {` `    // statements dedicated to the initialization of agents` `  }` `  reflex reflex_name {` `      // set of statements` `  }` `  aspect square {` `    draw square(10);` `    draw circle(5) color: #red ;` `  }` `}`
Use of an architecture *# by default, species use the reflex architecture* *# Agents can still use reflex behaviors, even with another architecture.*	`species mySpeciesArchi control: fsm {` `}`
Use of skills *# by default, no skill is associated with a species.* *# A skill provides additional attributes and actions.*	`species mySpecies3 skills: [moving, communicating] {` `}`
Inheritance *# No multiple inheritance is allowed.*	`// mySpecies2 gets all attributes and behaviors from mySpecies1` `species mySpecies2 parent: mySpecies1 {   }`

Creation of agents

Creation of N agents of a species *# Agent creation is often done in the global init.* **Creation of** N agents of a species Initialization of the agents	`create mySpecies1 number: 10;` `create mySpecies1 number: 20 {` `  an_int <- 0;` `}`
Creation from (shapefile or csv_file) data *# Objects of the file have an id attribute.*	`create mySpecies1 from: a_shp_file` `      with: [an_int::int(read('id'))];`

Definition of an experiment

<table>
<tr>
<td>
<u>experiment</u> expName <u>type: gui</u>
Set of parameters

<u>Outputs definition</u>
<u>display</u>
species, grid, agents

<u>display</u>
<u>chart</u>
data

#As many displays as needed can be created (charts or agent display). Each represents a point of view on the simulation.
<u>experiment</u> expName <u>type: batch</u>
Set of parameters
Exploration method

<u>Outputs definition</u>
<u>display</u>
<u>chart</u>
data

#In the batch experiment, charts can be used to plot the evolution over the simulations of a global indicator.
</td>
<td>

```
experiment expeName type: gui {
  parameter "A variable" var: an_int <- 2
       min: 0 max: 1000 step: 1 category: "Parameters";
  output {
    display display_name {
            species mySpecies2 aspect: square;
            species mySpecies1;
    }
    display other_display_name {
            chart "chart_name" type: series {
              data "time series" value: a_float;
            }
    }
  }
}
// repeat defines the number of replications for the same parameter values
// keep_seed means whether the same random generator seed is used at the first replication for each parameter values
experiment expeNameBatch type: batch repeat: 2
       keep_seed: true until: (booleanExpression) {
  parameter "A variable" var: an_int <- 2 min: 0 max: 1000 step: 1 ;
  method exhaustive maximize: an_indicator ;

  permanent {
    display other_display_name {
      chart "chart_name" type: series {
        data "time series" value: a_float;
      }
    }
  }
}
```

</td>
</tr>
</table>

Scheduler

<table>
<tr>
<td>
Agents of a species are executed at each step, by default in their creation order.
<u>Default schedule</u>

<u>Random schedule</u>

<u>No schedule</u>
The agents are not scheduled (i.e. not executed). It could be useful when defining passive agents.

<u>Schedule manager</u>
The schedule of each species is centralised and delegated to a manager agent. (All the species need to be unscheduled).
</td>
<td>

```
// Equivalent to species schedul_def { }
species schedul_def schedules: schedul_def
{ }

species schedul_rnd schedules: shuffle(schedul_rnd)
{ }

species no_schedul schedules: []
{ }


species spec1 schedules: []
{ }

species spec2 schedules: []
{ }

// The schedul_manager agent will first schedule agents of spec2 species and then the ones from spec1 (in a random order)
species schedul_manager schedules: spec2 + shuffle(spec1) {
{ }
```

</td>
</tr>
</table>

Grid and field

*# **grid** allows the modeler to define a specific kind of species: agents representing the cells of the grid cannot move, have a default square shape, and additional attributes, such as **color** (used for the default display of the grid), **grid_x**, **grid_y** (coordinates of the cell in the grid), **neighbors**, **grid_value.*** **grid** SpeciesName [additional attributes] Definition of the set of attributes **init** statements **behavior** behaviorName statements **aspect** aspectName statements to draw the agents *# **grid** can thus be initialised from a tabular datafile (e.g. asc, tiff). The value in the datafile will thus be stored in the built-in attribute **grid_value.***	`// Definition of a grid with 10x10 cells, and where the number of neighbors is specified (can be 4, 6 or 8 neighbors). When it I s 6, cells have a hexagon shape, with a given orientation` `grid cell height: 10 width: 10` `neighbors: 6 horizontal_orientation: true {` `}` `//Grid agents can be initialized using the tabular file (e.g. a DEM file as an asc file): the width and height of the grid are directly read from the file. The values of the asc file are stored in the grid_value attribute of the cells.` `grid cell file: file('../includes/hab10.asc') {` `init {` `color <- grid_value = 0.0 ? #black :` `(grid_value = 1.0 ? #green :` `#yellow);` `}` `}` `// Various facets have been introduced to optimize the use of grids (in memory and execution time): e.g.:` `grid cell file: dem_file neighbors: 8` `frequency: 0` `use_regular_agents: false use_individual_shapes: false` `use_neighbors_cache: false` `schedules: [] parallel: parallel {` `}`
*# **field** datatype has been introduced to store and to manipulate tabular datafiles (e.g. DEM asc file), without creating agents.* *# **field** has a built-in attribute **bands** (to read several dimensions data)* *# **field** can be displayed using the specific **mesh** statement* **experiment** expName **type: gui** **Outputs definition** **display** "foo" type: opengl **mesh** *a_field_var* [additional facets]	`// Load the data in a field` `field field_display <- field(grid_file("includes/Lesponne.tif"));` `// data in field can be updated` `field var_field <- field(field_display - mean(field_display));` `// Fields can be displayed using the mesh statement` `experiment Field_view type:gui{` `output {` `display "field through mesh" type:opengl {` `mesh field_display grayscale:true scale: 0.05` `triangulation: true smooth: true` `refresh: false;` `}` `display "rgb field through mesh" type:opengl {` `mesh field_display color: field_display.bands` `scale: 0.0 refresh: false;` `}` `}` `}`

Multi-level species

*# The **Multi-level architecture** in GAMA is based on the idea that some agents can aggregate some agents, to provide a higher level of agents in the model. To this purpose the higher-level agent can **capture** lower-level agents (aggregation) and **release** them (desegregation)* *# Technically, agents of a species **spec1** can be aggregated in agents of the **low_level_spec** species (that **inherits** from spec1) defined inside the **high_level_spec** species* *# The environment of low_level_spec agents is the shape of the high_level_spec agent that captured them.* *# The release should thus specify in which environment the agents are released (in general in the global world).*	`// Species pedestrian which will be captured by the corridor agent.` `species pedestrian {` `    point target_location;` `    rgb color;` `}` `//Agents of the species corridor will be the high-level agents.` `species corridor {` `    //Subspecies for the multi-level architectures : captured_pedestrian agents are the low-level agents` `    species captured_pedestrian parent: pedestrian` `        schedules: [] {` `      float release_time;` `    }` `// Reflex to capture pedestrians if the condition is true` `    reflex aggregate when: capture_pedestrians {` `      capture (pedestrian where (a_condition))` `      as: captured_pedestrian {` `        release_time <-rnd(10.0);` `      }` `    }` `//Reflex to release pedestrians which have already passed enough time in the corridor` `    reflex disaggregate {` `      list tobe_released_pedestrians <- captured_pedestrian where (time >= each.release_time);` `      if !(empty(tobe_released_pedestrians)) {` `        release tobe_released_pedestrians` `      as: pedestrian in: world {` `          location <- any_location_in(world);` `        }` `      }` `    }` `}`