# Structure of a model/program

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

# Comments

| | |
|---|---|
| **Block comments** | `/* A block comment starts with the an opening symbol.`<br>`  The comment runs until the closing symbol below.`<br>`*/` |
| **Inline comments** | `// This is an inline comment.`<br>`// 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`<br>`import "path_to_model/model2.gaml"` |

# Primitive types

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

# Other types

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

**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*

```
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))  )
```

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

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

*# 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**)*

```
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";
        }
}
```

# 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

**Procedure** ProcedureName (pd1, pd2)
  actions

```
action myAction {
   write "Action without param";
}

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.*

*# if the procedure has been defined in another species, the current agent has to ask an agent of this species to call the procedure.*

```
do myAction();
do myActionWithParam(3, "other string");

do myActionWithParam(3);  // the second parameter has its default value

ask an_agent {
   do proc(3);
}
```

## Declaration of a function

**Function** FunctionName **: type**
  actions
  return value

**Function** FunctionName (pd1, pd2) **: type**
  actions
  return value

```
int myFunction {
   return 1+1;
}


int myFunctionWithParam(int i, int j <- 0){
   return i + j;
}
```

## Call of a function

Variable ← FunctionName ()

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.*

```
// 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);


// 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**

*# Replacement of an element from list or matrix.*
*# In map, we can replace the value associated to a key.*

```
// 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;

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 maps have to be done on keys, values or pairs list*

```
// loop over values of a list
loop i over: list_int {   }

// 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.*

```
create mySpecies1 number: 10;
```

**Creation of** N agents of a species
  Initialization of the agents

```
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

**experiment** expName **type: gui**
  Set of parameters

  **Outputs definition**
    **display**
      **species, grid, agents**

    **display**
      **chart**
        **data**

*#As many displays as needed can be created (charts or agent display). Each represents a point of view on the simulation.*

**experiment** expName **type: batch**
  Set of parameters
  Exploration method

  **Outputs definition**
    **display**
      **chart**
        **data**

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

```
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;
         }
      }
   }
}
```

## Scheduler

*# Agents of a species are executed at each step, by default in their creation order.*
**Default schedule**

**Random schedule**

**No schedule**
*# The agents are not scheduled (i.e. not executed). It could be useful when defining passive agents.*

**Schedule_manager**
*# The schedule of each species is centralised and delegated to a manager agent. (All the species need to be unscheduled).*

```
// 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) {
{  }
```

## 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

```
// 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 manipulate tabular datafiles (e.g. DEM asc file), without creating agents.*

```
global  {
   field field_display <-  field(grid_file("includes/
Lesponne.tif"));
   field var_field <- field(field_display -
mean(field_display));
}

experiment Field_view type:gui{
   output {
      layout #split;
      display "field through mesh" type:opengl {
        mesh field_display grayscale:true scale: 0.05 refresh:
false triangulation: true smooth: true;
      }
      display "rgb field through mesh" type:opengl {
        mesh field_display color:field_display.bands scale: 0.0
refresh: false;
      }
      display "rnd field with palette mesh"  type:opengl {
        mesh field_display.bands[2] color:scale([#red::100,
#yellow::115, #green::101, #darkgreen::105]) scale:0.1
refresh:false;
      }
      display "var field" type:opengl  {
         mesh var_field color:(brewer_colors("RdBu")) scale:0.0;
      }
   }
}
```