# 2. Practice - An Introduction to NetLogo

Tamás Takács, PhD student, Department of Artificial Intelligence

90 min read

January 22, 2025

Collective Intelligence

---

NetLogo is a **programmable modeling environment** for simulating natural and **social phenomena**, based on **Logo** by Seymour Papert. It is designed to model **complex system** development over time.

1. **Complex System** a system made up of many interacting components or agents, where the interactions give rise to emergent behaviors that cannot be easily predicted from the behavior of individual components (congested road network).
2. **Social Phenomena**: behaviors, patterns, or events that arise within societies due to the interactions and relationships among individuals or groups (voting patterns, Romania 2024).
3. **Logo**: high-level, interpreted, dynamically-typed programming language designed for educational purposes. Comes with functional paradigms, symbolic processing and turtle graphics.
4. **Programmable Modeling Environment**: a software tool that allows users to create, simulate, and analyze computational models of complex systems. It provides a framework for defining agents, their behaviors, and interactions within an environment (NetLogo).

## Why NetLogo?

Are there any other programmable modeling environments out there? **Yes, there are**.

Many competing modeling environments use programming languages inspired by or similar to Logo. Most of these platforms integrate another high-level programming language, such as Java, to enhance the user experience. However, much of the Java code in these systems has become outdated.

### Close Competitors:

| Feature | GAMA | NetLogo | Repast |
|---|---|---|---|
| **License** | Open-source (GPL v3.0) | Open-source (GPL) | Open-source (BSD) |
| **Programming Language** | GAML (Gama Modeling Language) for simulations; Java for extensions | NetLogo language | Java (RepastS, RepastJ); Python (Repast4Py); .NET languages (Repast.NET) |
| **Operating Systems** | Cross-platform: Windows, Linux, macOS | Cross-platform: Windows, Linux, macOS | Cross-platform: Windows, Linux, macOS |
| **Primary Domain** | Spatially explicit agent-based simulations | Social and natural sciences; educational purposes | Social sciences; complex adaptive systems |
| **User Support** | Tutorials, manual, FAQ, forums, documentation, selected publications, examples | Documentation, FAQ, selected references, tutorials, third-party extensions, mailing lists | Documentation, mailing list, defect list, reference papers, external tools, tutorials, FAQ, examples |
| **GIS Capabilities** | Advanced GIS support, allowing integration and manipulation of spatial data | Basic GIS support; can handle simple spatial data | Extensive GIS capabilities; supports integration with various spatial data formats |
| **3D Capabilities** | Supports 3D simulations and visualizations | Basic 3D capabilities; primarily 2D | Supports 3D simulations and visualizations |
| **Learning Curve** | Moderate; requires understanding of GAML and modeling concepts | Beginner-friendly; designed for ease of use and learning | Steeper; requires proficiency in Java or other supported languages |
| **Performance** | Suitable for large-scale simulations; performance depends on model complexity | Best suited for small to medium-scale models; performance may degrade with very large models | Designed for high-performance simulations; suitable for large-scale and complex models |
| **Debugging** | Reports syntactic and semantic errors and gives semantic warnings that indicate “flaws in the logic of the model” | Limited amount of debugging functionalities | Many syntax errors are only identified at runtime |
| **Simulation Speed** | Slowest (tested on Game of Life) | Fastest (tested on Game of Life) | Fast (tested on Game of Life) |

| Feature | GAMA | NetLogo | Repast |
|---|---|---|---|
| **Multi-Threading** | Yes | No | Yes with Repast HPC |
| **Latest Version** | 2.11 (as of January 20, 2025) | 6.4.0 (as of January 20, 2025) | 1.9.3 (as of January 20, 2025) |

*Reference: Raab, R., Lenger, K., Stickler, D., Granigg, W., & Lichtenegger, K. (2022). An Initial Comparison of Selected Agent-Based Simulation Tools in the Context of Industrial Health and Safety Management. Proceedings of the 2022 8th International Conference on Computer Technology Applications, 106–112. Presented at the Vienna, Austria.* doi:10.1145/3543712.3543745

---

### Code Comparison:

```
to go
  ask patches
   [ set live-neighbors count neighbors with [ living? = true ] ]
  ask patches
   [ if living? = true  and live-neighbors < 2 [ cell-death ]
     if living? = true  and live-neighbors > 3 [ cell-death ]
    if living? = false and live-neighbors = 3 [ cell-birth ] ]
  set living-percentage (count patches with [ living? = true ])
   / (count patches)
  tick
end
```

**Figure 1: Main Procedure in NetLogo.**

```
def go(){
    ask(patches()){
     living_neighbors = count(neighbors(1,1).with{living == true})
    }
     ask(patches()){
      if(living == true  & living_neighbors <  2) {cell_death()}
       if(living == true  & living_neighbors >  3) {cell_death()}
       if(living == false & living_neighbors == 3) {cell_birth()}
      }
     living_percentage = (count(patches().with
         { living == true })/count(patches()))
    tick()
}
```

**Figure 2: Main Procedure in Repast Simphony ReLogo.**

```
reflex go {
    ask life_cell {
     live_neighbors <- self neighbors_at 1 count each.living;
    }
    ask life_cell {
    if living = true  and live_neighbors < 2 {do cell_death;}
    if living = true  and live_neighbors > 3 {do cell_death;}
    if living = false and live_neighbors = 3 {do cell_birth;}
    }
   living_percentage <- life_cell count each.living / length(life_cell);
}
```

**Figure 3: Main Procedure in GAMA.**

---

## New Competitors:

New competitor modeling environments aim to leverage multi-threading and more efficient, faster programming languages as their backbone to outperform traditional Java-based platforms. The market remains highly competitive, with ongoing efforts to develop the most comprehensive and versatile programmable modeling software.

**Table 1.** A comparison of four ABM frameworks covering objective categories focusing on ease of use, available functionality and performance. Colours represent implementation quality. Red: poor/none, Yellow: basic, Green: good, Blue: clear class leader. Further details corresponding to the superscript numbers are given in the main text.

| | Agents.jl 4.2 | Mesa 0.8 | NetLogo 6.2 | Mason 20.0 |
|---|---|---|---|---|
| | Objective property comparisons. | | | |
| **Core** | Core design decisions and aspects that cannot be changed or implemented by users | | | |
| **Continuous Space** | Yes | Yes | Yes | Yes |
| **Graph Space** | Yes, and mutable | Only undirectional | Link Agents (not a Space) | Networks (not a Space) |
| **Grid Space** | Yes | Yes (+Hexagonal) | Yes | Yes (+Hexagonal, Triangular) |
| **OpenStreetMap Space** | Yes | No | No | No |
| **Dimensionality** | Any[1] | 2D | 2D & 3D (separate applications) | 2D & 3D (complicated install for 3D) |
| **License permissiveness** | MIT | Apache v2.0 | GPL v2 | Academic Free License |
| **Mixed-agent models** | Yes | Yes | Yes | Yes |
| **Simulation termination** | After 'n' steps or user-provided boolean condition of model state | Explicitly written user loop | Manually by pressing a button on the interface, stop command in code | When Schedule is empty, or user provided custom finish function |
| **Parameter types** | Anything | Anything | Float64, Lists Hashtables and Assoc. Arrays in the Table extension | Anything |
| **Modeling and Analysis in the same language** | Yes, Julia v1.5+ | Yes, Python v3+ | No | Yes, Java but designed to work within the console or GUI of the applet |
| **Maximum memory capacity** | Hardware limits | Hardware limits | 1 GB Manually expanded by increasing JVM heap | 1 GB Manually expanded by increasing JVM heap |
| **Distributed computing[2]** | Yes | No. BatchRunnerMP is only multithreaded | No. BehaviorSpace is only multithreaded | Yes |
| **Interop with external libraries** | Yes, also couples to anything in Python / R / C / C++ seamlessly. | Yes, modular design. | Partial, via the Extensions API. JVM languages (Scala, Clojure) | Partial. Extensions in the 'contrib' directory. No simple user API |
| **Language ecosystem integration** | By Design. Examples: black box optimization, differential equations | Any of Python's analytical tools can be used | Complex. Must create plugins or use Control API | Warned against (e.g. Random), provides custom types in place of Java primitives |
| **Browser-based online ABM execution** | No | No | Yes (NetLogo Web) | No |
| **Data collection** | Any chosen parameter / property or function mapped over them. Aggregating and filtered aggregate functions | Any chosen parameter / property. Aggregating functions. No conditional options | boolean, number, string and lists of these types. | Inspectors track & chart any parameter / property. Entire model saved to disk via checkpointing, no custom export |

*Reference: Datseris, G., Vahdati, A. R., & DuBois, T. C. (2022). Agents.jl: a performant and feature-full agent-based modeling software of minimal code complexity. SIMULATION, 100(10), 1019–1031.* doi:10.1177/00375497211068820

Other notable advantages of NetLogo:

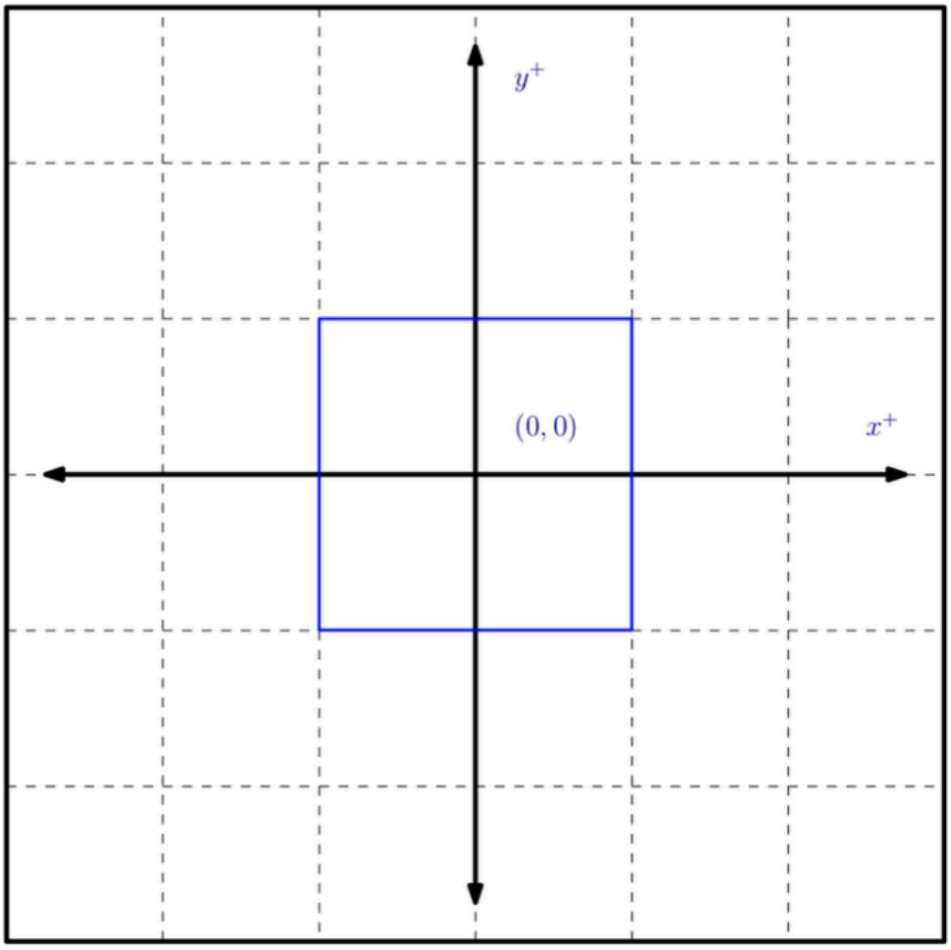
- Extensive documentation (literally contains everything a good documentation needs)
- Huge collections of pre-written simulations on Biology, Medicine, Physics, Chemistry and more
- Very easy to get into

---

# Building Blocks

## The Environment

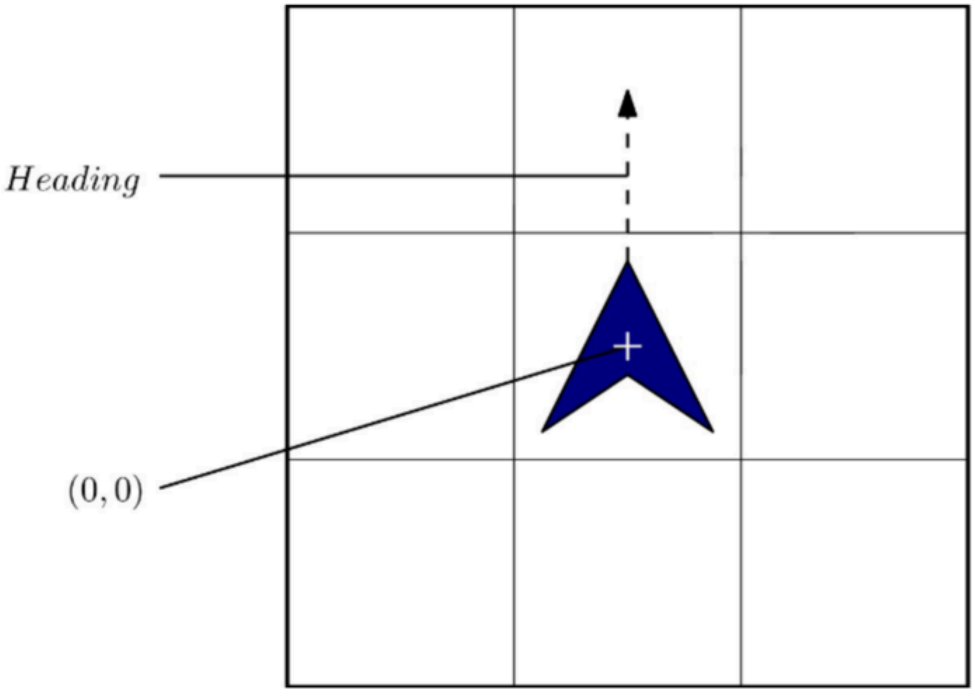
The whole world is a **discrete grid**. Each basic region is called a **patch**.



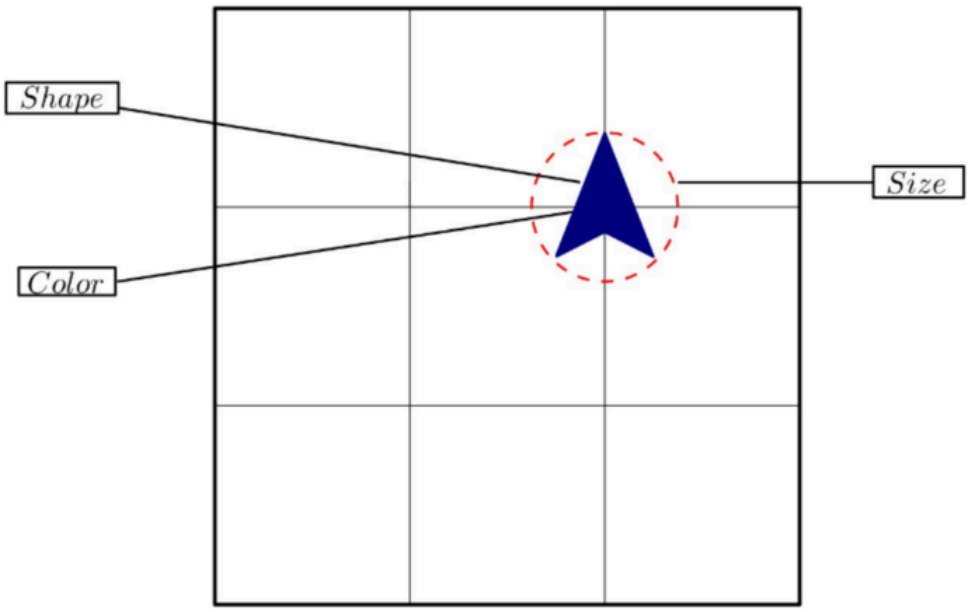

---

## The Agents

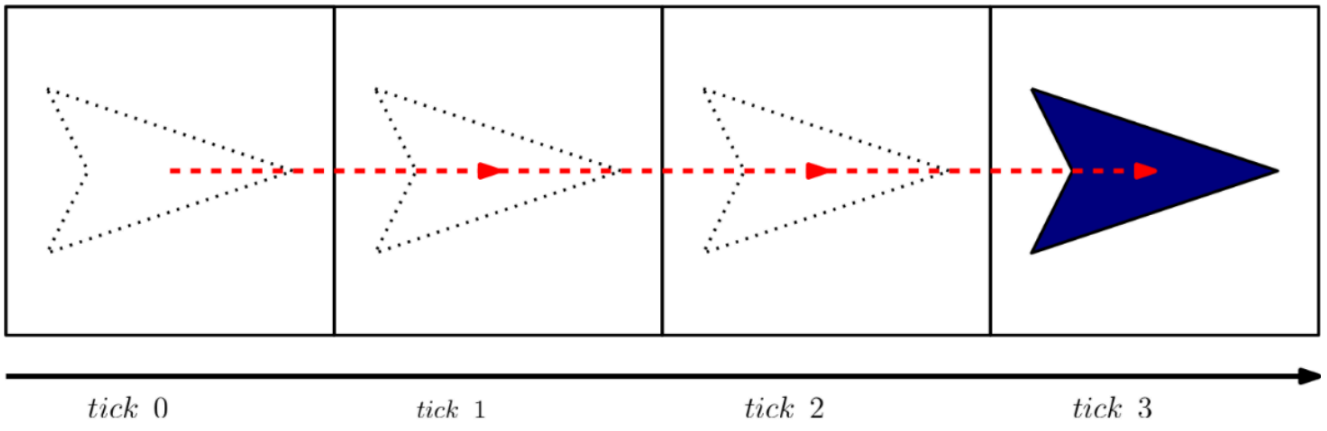
The environment is composed of **agents called turtles** that can independently move. Each turtle has a **position, coordinates, and a heading**, expressed in degrees. 0° is north.




Agents possess descriptive features as well such as their **size, color and shape.** (Mostly used for visualization purposes)




Just like space, time in simulations is also discrete, progressing in units called **ticks**. A tick represents a moment in simulation time during which agents perform their actions. By default, a scheduler ensures agents act in a random order each tick, though this behavior can be customized as needed.




Each agent is equipped with a set of properties:

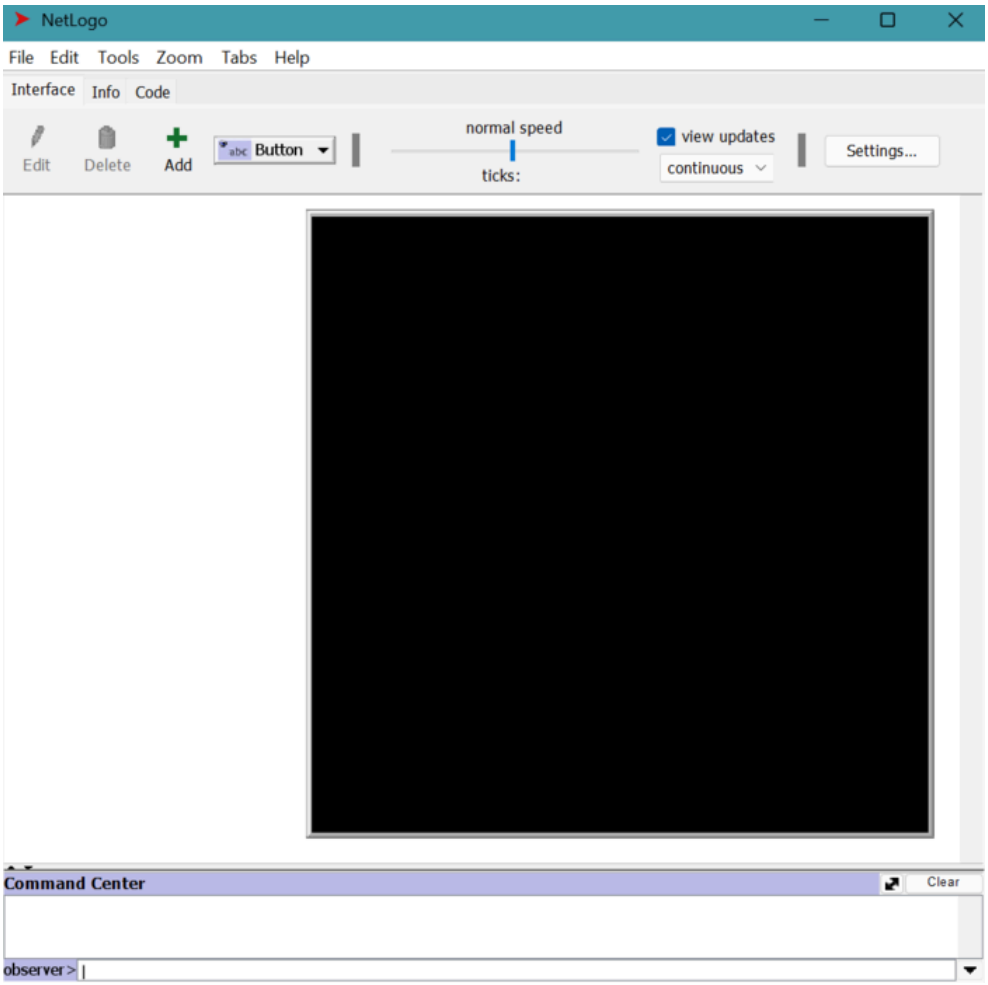
- **who**: A unique identifier assigned to each agent, distinguishing it from others in the simulation.
- **heading**: The direction the agent is facing, typically measured in degrees.
- **xcor and ycor**: The agent's coordinates on the grid, defining its position in the simulation environment.
- **shape, size, color**: Visual attributes of the agent, determining how it appears in the simulation.
- **hidden**: A boolean property indicating whether the agent is visible or hidden in the simulation.

---

## The Observer

The **observer** in NetLogo acts as an overseer, responsible for managing and modifying the environment and agents without being an agent itself. It can execute commands to create, move, or modify turtles, patches, and links, as well as control the simulation by adjusting global settings, running procedures, and monitoring overall behavior.

## Starting Up NetLogo (6.4.0)




Opening NetLogo presents a minimalist interface with a blank project, including an empty grid window by default. The easiest way to begin interacting with this environment is through the **Command Center**.

By default, you are acting as the **observer**, which grants full control over the entire environment with a global perspective. In observer mode, commands are executed at the global level and can directly manipulate the environment, agents, and simulation settings.

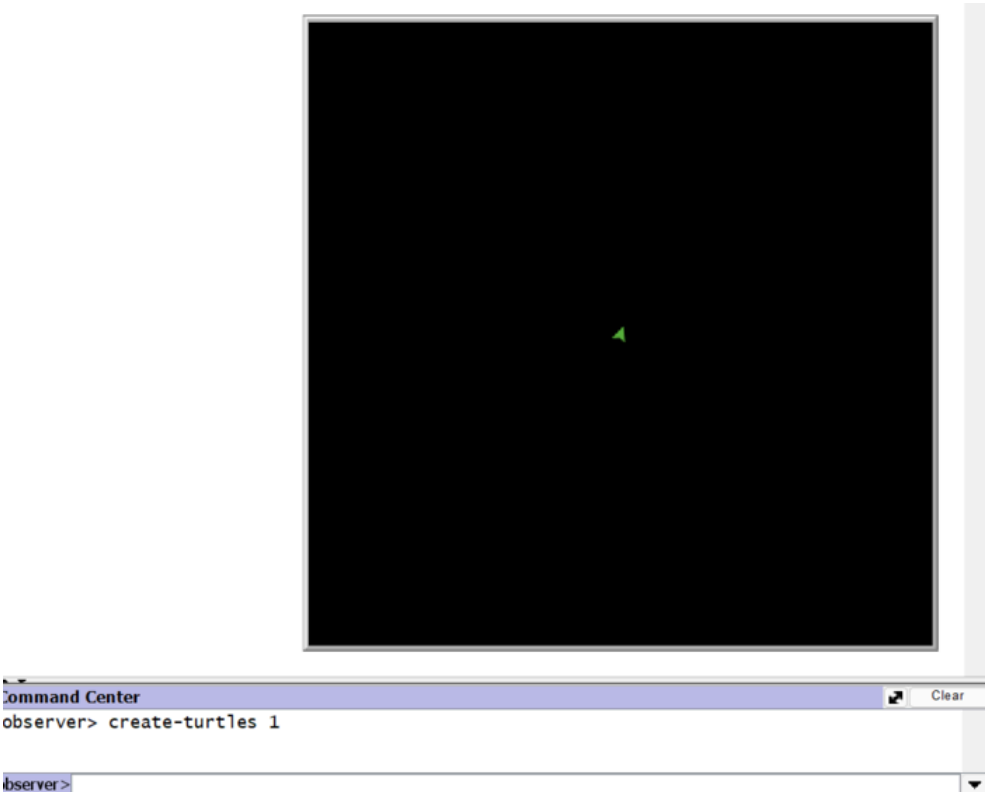
---

## Appetizer Commands

```
create-turtles 1
```

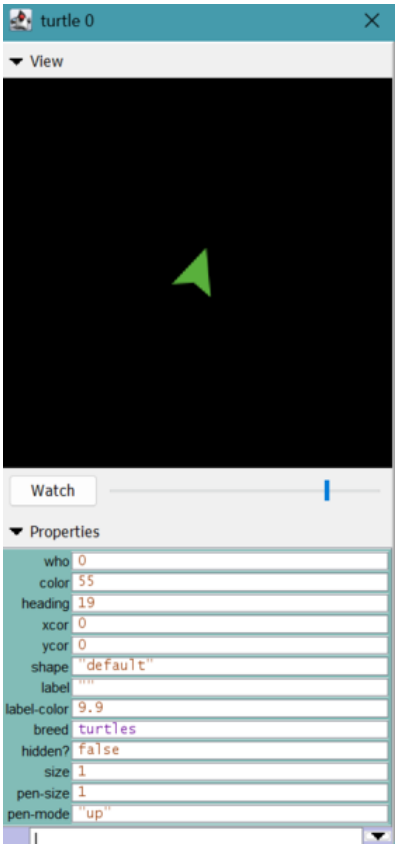
This code may appear simple, but it performs multiple actions behind the scenes:

- **Action**: It creates **1 new turtle** in the simulation.
- **Default Properties**: The newly created turtle is assigned default values for its properties, such as:
  - A unique identifier ( `who` number, in this case it will be 0).
  - Randomly chosen initial `heading` (direction).
  - A random `xcor` and `ycor` (position) within the world's boundaries.
  - Default visual properties like `shape`, `color`, `size`, and `hidden` status (not hidden by default).
- **Scope**: This command is executed from the **observer** context, meaning the observer initiates the creation of the turtle(s) in the environment.




```
inpsect turtle 0
```

The command `inspect turtle` is used to inspect all properties of a turtle.




In this window, you can observe additional properties of the turtle that were not mentioned previously:

- **label**: A text string displayed next to the turtle. By default, it is empty, but it can be customized to display numbers, words, or other information.
- **label-color**: The color of the text in the turtle's label, displayed as a numerical color code.
- **breed**: The classification of the turtle, used to group turtles into subcategories for specific behaviors or roles. By default, turtles belong to the `turtles` breed.

- **pen-size**: The width of the pen used by the turtle when drawing on the grid, measured in pixels.
- **pen-mode**: Determines the drawing behavior of the turtle's pen. Possible values include `up` (not drawing), `down` (drawing as it moves), or `erase` (erasing lines as it moves).

There is also an **input field** at the bottom of the Inspect window, which allows you to directly modify properties or execute commands for the selected agent or object (such as a turtle, patch, or link) in the simulation.

For example, to change the label of a turtle, you can use the following command:

```
set label "Shrek"
```

The `set` command modifies the properties of the agent selected in the Inspect window. However, if you want to do this from the **observer level**, you need to use the `ask` command to specify which agent you are targeting:

```
ask turtle 0 [set label "Donkey"]
```

This observer-level command is slightly modified from the original, with the inclusion of the `ask` keyword, along with the breed (`turtle`) and the unique identifier (`who`) of the agent in question. The `ask` command in NetLogo allows the observer to direct specific agents (or groups of agents) to perform actions.

> **NOTE:**
>
> 1. The `set` command works within the context of the selected agent and can modify its own properties directly.
> 2. Using the `ask` command, an agent can modify the properties of another agent. When an agent uses `ask`, it essentially "steps into" the context of the target agent(s).

> **Exercise:**
>
> 1. Use the `create-turtles` command in the **Command Center** to add a new turtle.
> 2. Right-click on the newly created turtle (Turtle 1) and select "Inspect" to open its Inspect window.
> 3. In the input field of Turtle 1's Inspect window, try changing Turtle 0's label to "Shrek" again.

---

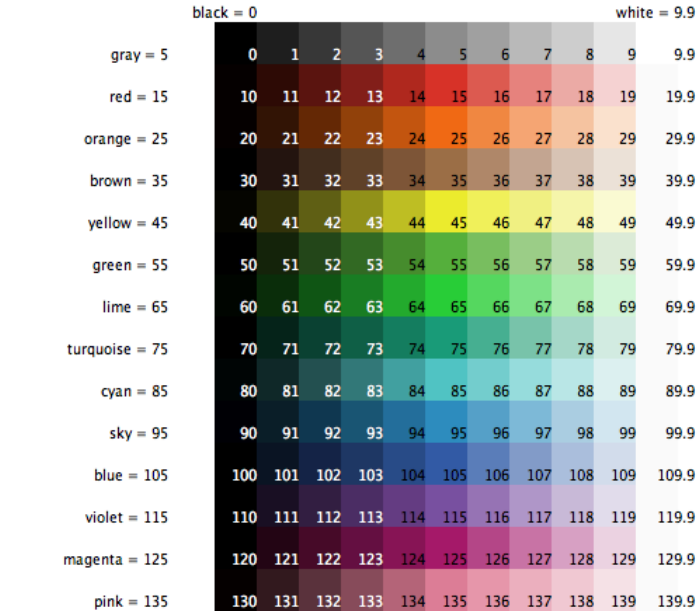
### The Color Property

You might have noticed that the `color` of **Turtle 0** is set to `55`, which might seem unusual. This is a greenish color in NetLogo's color scheme. Additionally, the `label-color` is set to `9.9`, which corresponds to white.

Let's try changing the `label-color` of **Turtle 0** to a different value, such as `19.9`:

```
set label-color 19.9
```

Surprisingly, nothing seems to happen. So, let's explore what's going on behind the scenes and understand how colors are coded in NetLogo.




*Reference:* https://ccl.northwestern.edu/netlogo/bind/article/shapes-and-colors-in-netlogo.html

NetLogo uses a **continuous color scale** based on numbers ranging from `0` to `140`. These numbers represent specific colors on the NetLogo color wheel:

- **Whole Numbers**: Each whole number corresponds to a base color (e.g., `0` is black, `15` is red, `65` is green, etc.).
- **Decimal Points**: Decimal values (e.g., `19.9`) create shades or variations of the base color. For example:
  - `9.9`: The lightest shade of a color, often close to white.
  - `19.9`: A lighter variation of red.

Let's try changing the `label-color` of **Turtle 0** to a value that is outside the bounds of NetLogo's color system:

```
set label-color 155
```

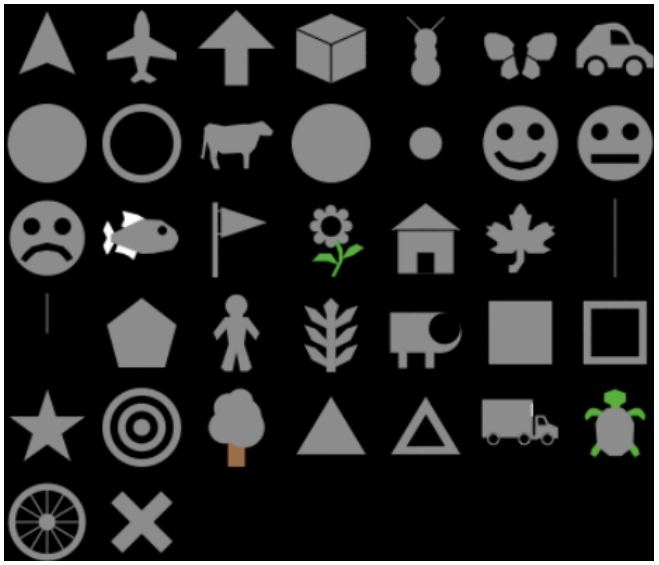
Interestingly, the `label-color` turns red. This happens because NetLogo handles out-of-bounds color values by applying the following formula: `color = set_color % 140` (graceful handling).

---

### Shapes

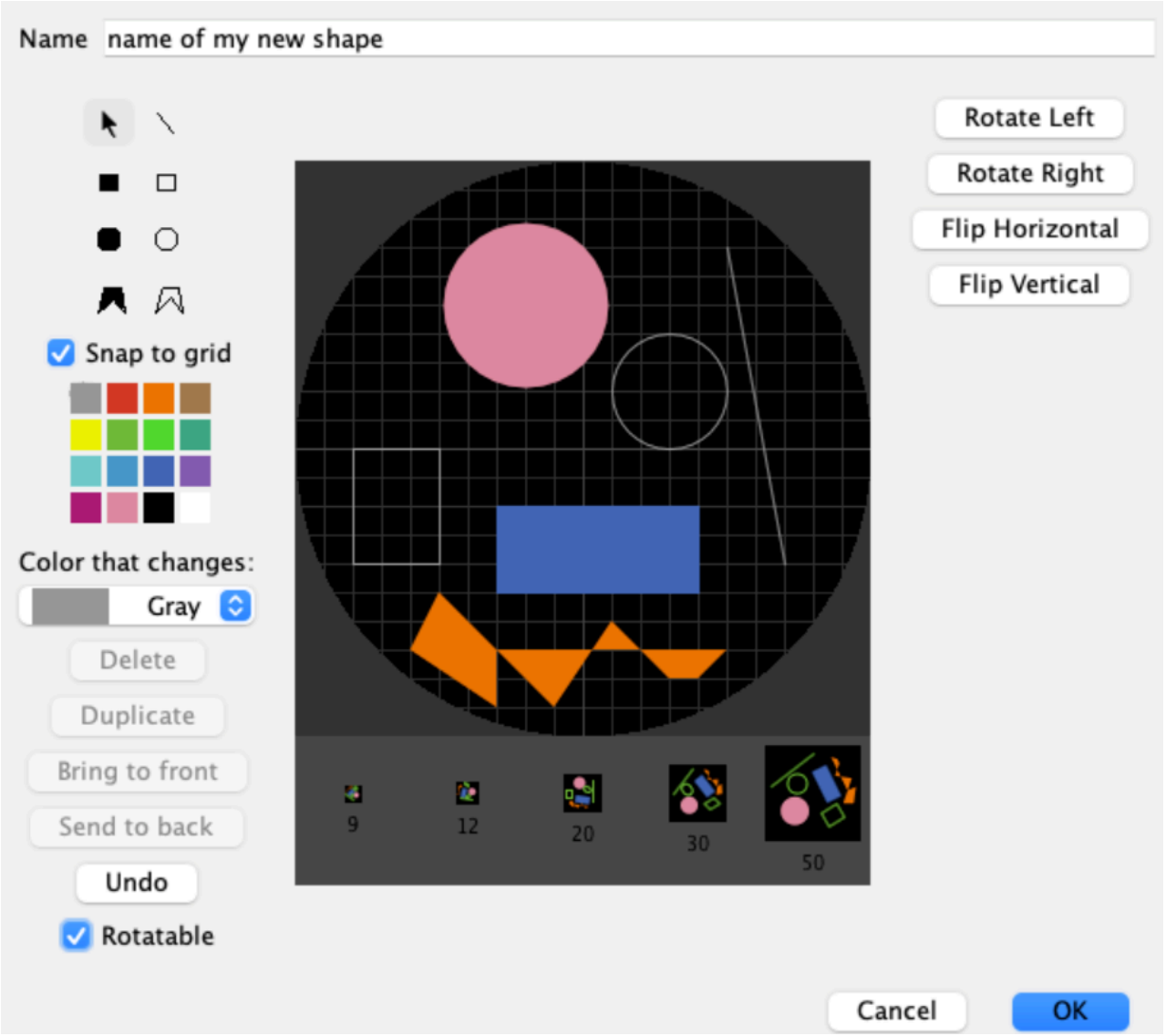
For some phenomena, modeling how agents **look** can be just as important as modeling their behavior. In other cases, creating visually appealing and creative visualizations can enhance our understanding and enjoyment of the modeling process.

NetLogo uses **vectorized shapes** for turtles, which are built from basic geometric components. By default, turtles use the `default` shape, but NetLogo also provides a library of pre-defined shapes that can be assigned to turtles to represent different roles or states visually. These shapes can be customized to suit the needs of your model.



Other notable shapes include the following: `airplane`, `bug`, `butterfly`, `person`, `house`, `car`.

NetLogo has way more turtle shapes than the default ones for us to choose from. All we need to do is to click the **Import From Library** button, which will bring up a long list of shapes to choose from.




*Reference*: https://ccl.northwestern.edu/netlogo/bind/article/shapes-and-colors-in-netlogo.html

---

### The Breed Property

The **breed** property defines a classification of agents, specifying their roles within the system. NetLogo provides a fallback breed called `turtles`, which is the default class for all agents unless explicitly assigned to another breed. This ensures that agents always have a default classification, even if no additional breeds are defined.

You can define additional breeds to represent different roles or behaviors in the system. For example, in a simulation of hunters and prey, you could create two separate breeds:

- **Hunters**: Agents that have specific goals and actions, such as chasing prey.
- **Prey**: Agents with different behaviors, like avoiding hunters or foraging for resources.

---

### The Pen Mode Property

The **pen-mode** property enables agents to leave a visual trail, following their trajectory as they move around the environment. The property can take the following values:

- `up`: The pen is lifted, and no trail is drawn as the agent moves.
- `down`: The pen is lowered, drawing a trail along the agent's path.
- `erase`: The pen erases any previously drawn trails as the agent moves.

This feature allows for the creation of intricate visual patterns, showing emergent behaviors in multi-agent systems through simple movement rules. Let's set the `pen-mode` of **Turtle 0** to `down`.

```
set pen-mode "down"
```

---

### Moving the Agents

Basic movement in NetLogo involves the following commands:

- `forward`: Moves the agent in the direction specified by its current `heading` property.
- `right` and `left`: Adjust the `heading` value of the agent, changing its direction of movement.

The `left` command subtracts the specified angle from the current `heading`, while `right` adds the specified angle to it. It's important to note that in NetLogo, the `heading` value is measured in degrees, with **0 degrees** representing north. For example:

- If an agent's `heading` is **180** (facing south) and you execute `right 180`, the agent will turn to face north (back to a `heading` of 0).

> **Exercise:** Use **Turtle 0** to draw a perfect **equilateral triangle** (a triangle with three equal sides and 60-degree angles) on the simulation grid.
>
> You will use the **pen-mode** property and the basic movement commands ( `forward` and `right` ) to accomplish this.

---

### Patches

In NetLogo, there are four types of agents: **turtles**, **patches**, **links**, and the **observer**. Commands can be directed to any of these agents, including patches.

Patches are arranged in a grid with each patch having specific coordinates. The patch at coordinates `(0, 0)` is called the **origin**, and the coordinates of other patches are determined by their horizontal and vertical distances from this origin.

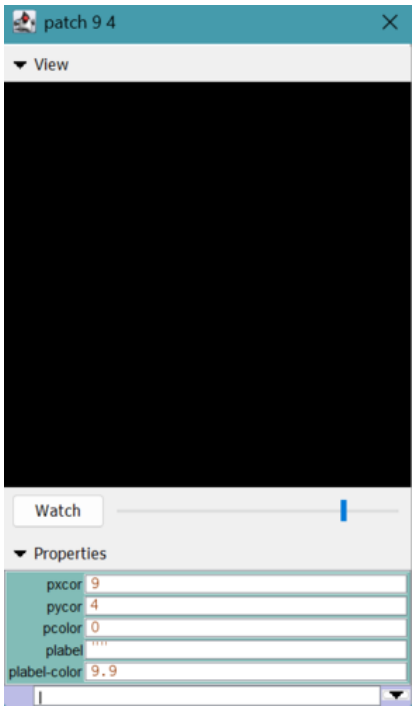
- `pxcor`: The horizontal coordinate (increases as you move to the right).
- `pycor`: The vertical coordinate (increases as you move upward).

These coordinates work similarly to the standard mathematical coordinate plane.

Commands in NetLogo can target a **specific turtle** or **specific patch** or the **entire set of turtles or patches**.

| | Turtles | Patches |
|---|---|---|
| **One** | `ask turtle 0 [ set color red ]` | `ask patch 2 3 [set pcolor red]` |
| **All** | `ask turtles [ set color red ]` | `ask patches [set pcolor red]` |

See https://ccl.northwestern.edu/netlogo/docs/dictionary.html for additional commands.




Patches also have a set of properties that can be manipulated, such as their **color** or **label**. For example, you can change the color of a patch to standard white by setting its color property to `9.9`. Here's how you can do it:

```
ask patch 9 4 [set pcolor 9.9]
```

So what would be needed to create, for example, a chessboard pattern on the grid? Is there anything beyond the Inspect Window and the Command Center to write more complex code, such as loops, creating breeds, handling complex data structures, and manipulating multiple elements at once? Of course, there is: **The Code Tab**.

---

## Programming in NetLogo

Instructions to agents can be classified according to three criteria:

- whether they are built into NetLogo (**primitive**) or user implemented (**procedure**)
- whether the instruction produces an output (**report**) or not (**command**)
- whether an instruction takes inputs or not=

---

### Commands

Commands are procedures that don't have any output, but only side effects on the environment.

```
to go
    clear-all
    create-turtles 10
    ask turtles [ forward 1 ]
end
```

This code defines a procedure called `go`, which performs the following actions:

1. `clear-all`: Resets the environment by clearing all agents, patches, and any previously drawn elements on the grid.
2. `create-turtles 10`: Creates 10 new turtles, each with default properties like random positions and headings.
3. `ask turtles [ forward 1 ]`: Asks all turtles to move forward by 1 step in the direction they are currently facing.

This command can then be called in the **Command Center** with the following line:

```
go
```

---

### Interface Elements

Is there another way to interact with the Code Pane from the Interface Tab? **Yes**, through **Interface Elements**, which allow users to modify and interact with the simulation without directly changing the code. The most notable elements include:

- **Button**: Executes a specific procedure or command when clicked.
- **Slider**: Adjusts numeric values dynamically to control variables.
- **Switch**: Toggles between `true` and `false` for boolean variables.
- **Chooser**: Allows selection from a predefined list of options.
- **Input**: Accepts user-provided text or numeric input.
- **Monitor**: Displays the current value of a variable in real time.
- **Plot**: Visualizes data over time or for specific conditions.
- **Output**: Prints text or data to a log-like area in the interface.
- **Note**: Displays descriptive text or instructions for the user.

> **Exercise:** Figure out which **Interface Element** would be appropriate to call this command once without interacting with the **Command Center**.
>
> Can you also determine how to call this function continuously within a loop?

---

### Reporters

Reporters are procedures that compute a value and report it.

```
to-report double [ num ]
    report 2 * num
end
```

The above code defines a reporter named `double` that performs the following actions:

1. **Input Parameter**: It takes a single input, `num`, which is the value to be processed.
2. **Computation**: It multiplies the input ( `num` ) by 2.
3. **Reporting the Result**: The `report` keyword is used to return the computed value (i.e., `2 * num`).

This reporter can be called in the **Command Center** or within other procedures to compute the double of a given number. For example:

```
show double 5
```

In the code above, `num` acts as an **input parameter** to the command.

> **Exercise:** Figure out which **Interface Element** would be most appropriate to monitor the value of the `double` reporter.

---

### Styling

There isn't an official NetLogo style guide. Nonetheless the official documentation is fairly consistent and follows some good habits:

- use camel case beginning with a lower-case letter for procedure (e.g. `myProcedure`, Java style)
- do not use underscores in names
- name command procedure with nouns and reporters with verbs

---

### Variables

Variables in NetLogo can be divided into three main groups:

- **Local variables**, defined as part of a procedure: `let <name> <value>`

- **Agent variables**, defined as part of each agent: `<agent*>-own [ <name(s)> ]`
- **Global variables**, accessible by every agent and procedure: `globals [ <name(s)> ]`

**Exercise:** Create a global variable named `radius` and set its value to 5 in a command named `setup`. Subsequently, create a reporter called `calculate-area` that calculates the area of a circle.

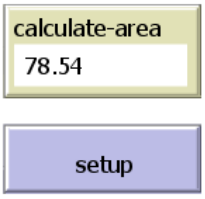
In this function, create a local variable named `area` that computes the area of the circle using the formula `area = π × radius²`, where `radius` is the global variable. The reporter should then return the computed area of the circle.

Use a **Button** interface element to run the `setup` command once, then utilize a **Monitor** interface element to show the result **with 3 decimal places**!

**Hints:**

- Use the `globals` keyword to define the global variable `radius`.
- Use the `set` command to assign the value `5` to `radius`.
- `pi` is already a predefined constant in NetLogo, so you don't need to define it manually.

**Solution:**
▶ **Click to show/hide solution**




**NOTE:**

- NetLogo variables are dynamically typed.
- Primitive types are **numbers, booleans, lists, strings**, along with the usual operations: + , -, * , /, ^ , > , >= , = , != , < , <= , and, or, not, xor.
- **All numbers are floating points**, be aware of approximations.
- When performing arithmetic operations **be aware of spaces**: the lack of parenthesis might bring ambiguity in parsing the operation and result in something different.

---

### Agentsets

When asking to update an agent variables a **subset of all the agents**, called `agentset`, can be used. An `agentset` contains one or more agents, all of the same type, and it's always randomly ordered.

```
ask one-of turtles [ <command> ]
```

The `one-of` primitive in NetLogo randomly selects one agent from a given set of agents, such as turtles, patches, or links. For example, `one-of turtles` randomly selects one turtle from the current set of turtles. Additionally, you can create subsets of agents using conditions (e.g., `turtles with [color = red]`) and then instruct these specific subsets with targeted commands.

```
let some-patches patches with [ pxcor < 3 ]
ask some-patches [ set pcolor red ]
```

---

### Conditionals

Conditionals in NetLogo allow agents to make decisions based on specific criteria using commands like `if`, `ifelse`, and `ifelse-value`. For example, `if pcolor = black [ set pcolor white ]` changes a patch's color to white only if its current color is black.

```
if (<condition>) [ <command(s)> ]

ifelse (<condition>)
    [ <command(s) if true]
    [ <command(s) if false]

ifelse-value (<condition)
    [ <reporter(s) if true]
    [ <reporter(s) if false]
```

```
if (random-float 1 < 0.5)
    [ show "heads" ]

ifelse (random-float 1 < 0.5)
    [ show "heads" ]
    [ show "tails" ]

ask turtles [
    set color ifelse-value (energy < 0)
    [ red ]
    [ green ]
]
```

Conditions are logical expressions (=, <, >, and, or, etc.) that evaluate to `true` or `false`.

**NOTE:**

- When using `if` or `ifelse` in an `ask` block, the condition is evaluated for each agent individually.

```
ask turtles [ if xcor > 0 [ set color red ] ]
```

---

### Loops

Loops in NetLogo allow repeated execution of commands, enabling dynamic and iterative behaviors. Common looping constructs include `repeat`, which runs a block of commands a fixed number of times, and `while`, which runs as long as a specified condition is true.

```
loop [ <command(s)> ]

repeat <num> [ <command(s)> ]

foreach <list> [ [<item>] -> <command(s)> ]
```

```
loop [ ifelse (counter > 100)
    [ stop ]
    [set counter counter + 1]
]

repeat 5 [
    ask one-of turtles [ set color red ]
]

foreach [1 2 3] [ [num] -> show num * 2 ]
```

**NOTE:**

- Loops inside an `ask` block are executed independently for each agent.

```
ask turtles [ repeat 5 [ forward 1 ] ]
```

---

### Lists

Lists in NetLogo are ordered collections of items, which can include numbers, strings, agents, or other lists. They are data structures that support operations like adding, removing, or accessing elements.

```
( list <element(s)> )

[ element(s) ]
```

```
( list 1 "two" true)

[ 1 "two" true ]
```

> **NOTE:**
>
> - In NetLogo lists are **immutable, ordered and potentially heterogeneous**.

**Some examples:**

```
let colors ["red" "blue" "green"]
show item 1 colors
```

The output will the firsts element or `item` in the list, which is `blue`.

```
let my-list [1 2 3]
set my-list replace-item 1 my-list 99
```

`my-list` will contain the values of [1, 99, 3] after using `replace-item`.

The `lput` primitive command adds an element to the end of a list, while `fput` adds an element to the beginning.

---

### Program Structure

The flexibility of NetLogo and its agent-centered way of building models quickly escalates to complex models that are difficult to work with.

Try to keep your structure as close as possible to:

- **global variable** declaration;
- **agent variable** declaration;
- **setup procedure**, in which global variables are initialized, agents are created and the environment is initialized;
- **go procedure**, which implements one cycle of the simulation.

---

### Higher-Order Procedure

Even though NetLogo is not a higher-order language we can simulate this behavior using **anonymous procedures/reporters**.

```
[ [ <var(s)> ] -> <body> ]
```

- `[ ]`: Encloses the entire anonymous procedure or reporter.
- `[ <var(s)> ]`: Specifies input variables (like function parameters) in a nested bracket. These variables can be used within the body.

- `->`: Indicates the start of the body of the procedure or reporter.
- `<body>`: The actual commands or expressions to execute. If it's a reporter, the result of this expression is returned.
  **Higher-order procedure:**

```
[ [ x y ] -> setxy y x ]
```

**Anonymous procedures assigned to variables (tasks):**

```
globals [ stack push ]

to setup
    set stack []  ; Initializes the stack as an empty list.
    set push [el -> set stack lput el stack] ; Defines the push task.
    run push 10
end
```

**Higher-order reporter:**

```
foreach [1 2 3] [ [x] -> show x * x ]
```

Unlike traditional procedures or reporters, anonymous ones are not stored in the **Code Tab** and cannot be reused unless redefined. While NetLogo doesn't directly support higher-order functions, anonymous procedures allow for similar behavior in many cases.

**NOTE:**

- **Map, filter and reduce** are basic constructors that allows efficient and elegant operations on lists.
- Map applies an anonymous-reporter to every element in a list.
- Filter applied a predicate (in the form of anonymous-reporter) to a list and returns only those items that entails the predicate.
- Reduce applies an anonymous-reporter from left to right, resulting in a single value.

```
map [ a -> a * a ] [ 1 2 3 ]
```

```
filter [ a -> a > 5 ] [ 1 9 2 ]
```

```
reduce [ [a b ] -> a + b ] [ 1 9 2 ]
```

---

### Breeds:

In NetLogo breeds are a way to "subclass" the turtle type.

```
breed [ <single name> <agentset name> ]
```

```
breed [ hunter hunters ]
breed [ prey preys]
```

After a breed has been created, the `ask` command can be used with the breed name (e.g., `ask hunters`) to execute actions for agents of that specific breed. All commands and properties applicable to turtles can also be used with the newly defined breed.

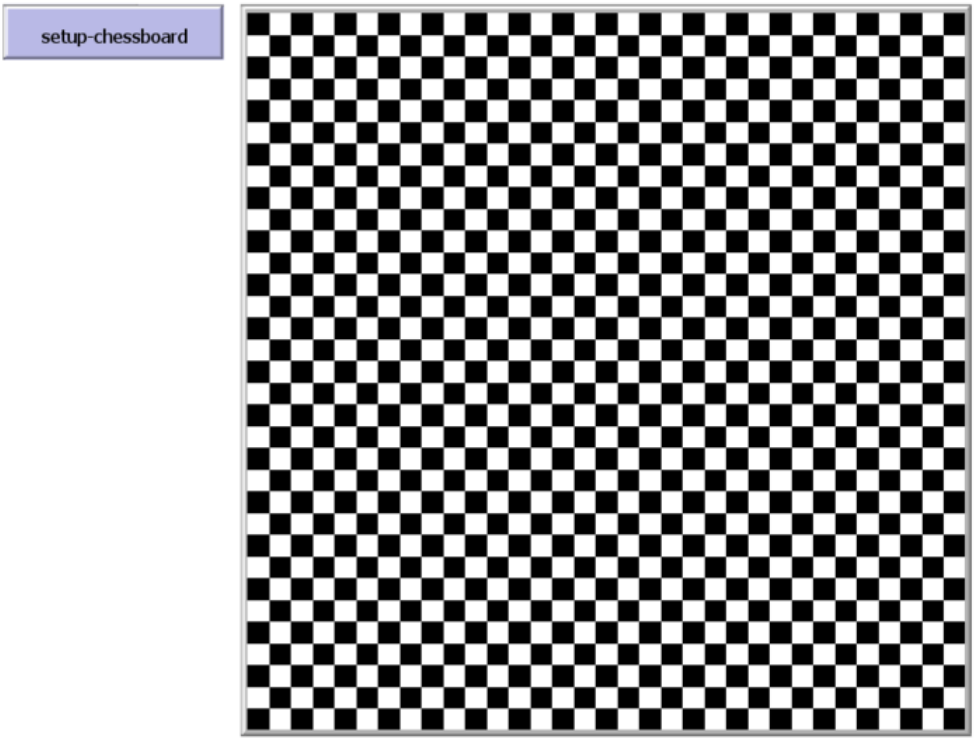
---

**Exercise:** Now that we've learned how to interact with the Code Pane, call functions from the Interface Tab, and use loops through interface elements, your task is to create a custom command called `setup-chessboard`. This command will clear the simulation environment and create a classic chessboard pattern on the grid using black and white patch colors.

**Optional:**

- Modify the grid size to fit different chessboard sizes (e.g., 8x8, 16x16).
- Use custom patch colors instead of black and white.
- Add another **Button** to clear the chessboard or overlay agents on specific patches.

**Solution:**
▶ **Click to show/hide solution**




---

**Exercise:** Design a **garden pattern** on the NetLogo grid using patches and turtles. The garden will consist of alternating flowerbeds (colored patches) and turtles (acting as flowers) placed in specific areas.

**Tasks:**

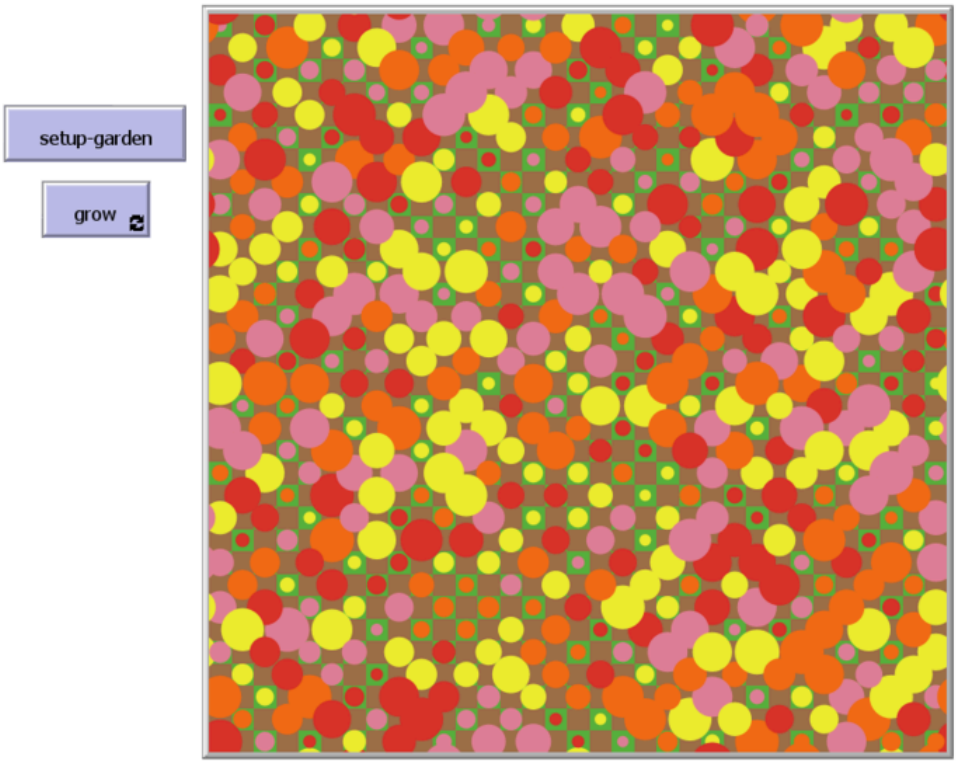
1. Write a procedure called `setup-garden` to:
- Clear the environment.
- Color the patches in a checkerboard pattern to represent flowerbeds.
- Place turtles (flowers) only on the green patches.
2. Customize the turtles:
- Set their **shape** to a flower (use `circle` if `flower` is unavailable in your version).
- Randomize their **size** and **color** to make the garden more realistic.
3. Use **ticks** to control the simulation. On each tick, make the turtles **grow** slightly (increase their size).

**Hints:**

- Use the `ask patches` command to create the checkerboard pattern.
- Use the `ask turtles` command to set their properties dynamically.
- Utilize loops, conditions (`if` statements), and lists where needed.

**Solution:**

▶ **Click to show/hide solution**




---

Tamás Takács

January 22, 2025

---