NetLogo Tutorial Series: Introduction and Core Concepts

Nicholas Bennett
nickbenn@g-r-c.com

October 2015

Copyright and license

Acknowledgments

Development of this curricular material was funded in part by:

- Santa Fe Institute Summer Internship/Mentorship (SIM) and Summer Complexity and Modeling Program (CaMP) for high school students;
- New Mexico Supercomputing Challenge;
- Project GUTS;
- New Mexico Computer Science For All (CS4All).

Participants in the above programs have also provided invaluable feedback on earlier versions of this material.

Introduction

What is NetLogo?

NetLogo is an open-source *agent-based modeling and programming* (ABM) tool, developed by the Northwestern University Center for Connected Learning (CCL) and Computer-Based Modeling [1]. It builds on the original Logo programming language [2], incorporating and extending concepts and constructs introduced in StarLogo and MacStarLogo [3], both developed by the Media Lab and the Scheller Teacher Education Program at MIT.

NetLogo is referred to as an agent-based tool due to the fact that the programming language and user interface are primarily intended for modeling and simulating systems of multiple interacting agents (see "Types of Agents", p. 12). Usually, these agents need not have extensive or complicated behaviors; useful models can often be developed with agents following very simple rules.

Developed in Java and Scala, and running on the Java Virtual Machine (JVM), NetLogo is very portable: models written in NetLogo for Windows (for example) can be modified and executed using the Macintosh OS X and Linux editions of NetLogo. Models can also be executed as Java applets in web pages (though this usage is now officially deprecated), or translated "on the fly" to JavaScript for browser-based execution. Extensions to NetLogo can be written in Java or Scala, and NetLogo itself can be instantiated and controlled by a program written in Java, Scala, or virtually any other language running in the JVM.

With each major release, NetLogo has been enhanced significantly. The agent-oriented features of the language have been rationalized, to be more consistent and coherent than in earlier versions. List and set operations have been expanded, and the performance of those operations improved. A link agent type has been added, supporting not only modeling of networks (social, communication, etc.) but also of physical and logical structures and assemblies.

NetLogo terminology

Like virtually all modern programming languages, NetLogo shares numerous concepts and facilities with other languages. However, the terminology used in NetLogo sometimes differs significantly from that of other languages. Some of these differences originated in earlier dialects of Logo, while others are the result of deliberate decisions by the NetLogo designers. In any event, a familiarity with a few of these distinctive terms – and their correspondence to terms used in other languages – can be very useful.

Following are three such terms that are critical for getting started; these and other terms are explored in more detail, later in this document.

- **Agent**

  From a programming point of view, agents are essentially *objects*: entities containing data, behaviors, and separate execution contexts. For graphical display purposes, a NetLogo agent (and in particular, a turtle, or mobile agent) is roughly equivalent to the concept of a *sprite* in many other languages: an entity that can be moved and displayed independently of other graphical elements.

  For more information on the different kinds of agents supported by NetLogo, see "Types of Agents" (p. 12).

- **Command**

  In NetLogo, a command is conceptually equivalent to what we usually call a *statement*: a specification of some action to be performed, to change the state of the system. In NetLogo, this system state includes not only global variables, a graphical display space, and the file system, but also the individual states of all agents.

  Commands may be invocations of *command primitives* (commands defined by NetLogo itself) or *command procedures* (defined in the program code of a model).

- **Reporter**

  In most programming languages, we call this an *expression*. Whereas a command is used to change the state of the system, the purpose of a *reporter* is to compute and return (report) some value. This may be a value of a primitive data type (e.g. a number, Boolean, or string of characters), an agent, or a data structure containing (potentially) multiple data elements.

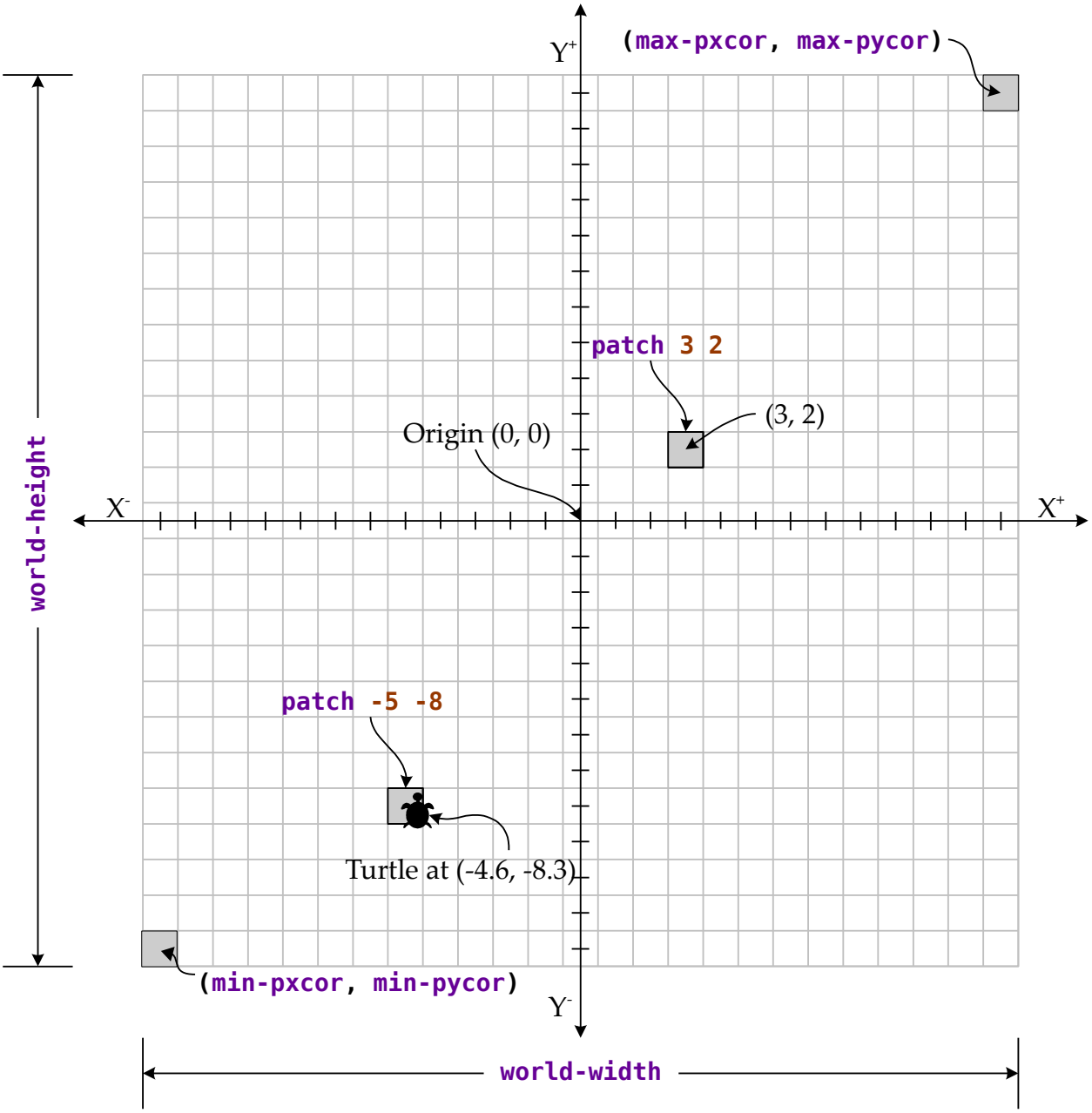
  Like commands, reporters can include invocation of *reporter primitives* (built into NetLogo itself) as well as *reporter procedures* (defined in a model's code). Note that what we usually call *operators* (e.g. the arithmetic symbols for addition, subtraction, etc.) are also reporter primitives in NetLogo.

  For more information on commands and reporters, see "Commands, Reporters, Definitions, and Declarations" (p. 11).

The NetLogo world

Coordinate system

In building NetLogo models, it's important to understand the coordinate system used by NetLogo. This diagram, and the explanations that follow, illustrate some important points to remember:



1. Like the Cartesian coordinate system traditionally used in algebra, analytic geometry, and calculus, the NetLogo world has X and Y axes. The center of the coordinate system is the origin (which is often – but not always – located in the physical center of the NetLogo world), where *X* and *Y* have values of zero (0).

2. Overlaid on the coordinate system is a grid of *patches* (1 X 1 squares), each of which is a stationary agent. Each patch has a color, and an optional label; a NetLogo program can also define additional variables for a patch.

3. The center of a patch is a point in the coordinate system with integral *X* and *Y* values; these coordinates are used to refer to the patch. For example, **`patch 3 2`** in the diagram is a square with its center at (3, 2); this square is the region where $2.5 \leq X < 3.5$ and $1.5 \leq Y < 2.5$. (We can also refer to patches with floating point coordinates; they'll be rounded to integers as necessary.)

4. A patch's coordinates are always integer values, but that's not necessarily the case for a turtle. In the diagram, there's a turtle located at (-4.6, -8.3), which is on the patch centered at (-5, -8). Though a turtle can appear as if it's on two or more patches at once, the turtle's center point is what matters: this center point is treated as the actual location of the turtle, and the patch containing that center point is considered to be the patch on which the turtle is standing.

5. The user can change the width or height of the NetLogo world at any time; because of this, NetLogo program code should not have world dimensions hard-coded as literal values, unless absolutely necessary. Fortunately, NetLogo programs can always use **`world-width`** and **`world-height`** to get the current dimensions of the world.

6. The patches on the extreme right-hand side of the NetLogo world have an X value of **`max-pxcor`**; those on the top of the world have a Y value of **`max-pycor`**. Similarly, **`min-pxcor`** and **`min-pycor`** are the X and Y coordinates (respectively) of the patches on the extreme left-hand side and bottom (respectively) of the NetLogo world. These variables are related to the overall size of the world, as follows:

```
world-width = (max-pxcor - min-pxcor) + 1
world-height = (max-pycor - min-pycor) + 1
```

Angles and directions

All angles in NetLogo are specified in degrees, and directions are based on compass headings, with 0° being "up" (i.e. north), 90° being towards the right (i.e. east), etc.

When instructing a turtle to face a particular direction, we can do so by setting the heading of the turtle to the desired compass heading, or by telling the turtle to turn right or left by the number of degrees required to orient the turtle as desired. We can also instruct a turtle to face another agent by specifying the second agent in a `face` command, rather than computing the compass direction or turn angle required.

Topology

Notice that we can specify that the NetLogo world should wrap horizontally, vertically, both horizontally and vertically, or not at all. When wrapping is turned on horizontally (for example), a turtle moving off the right edge of the world will reappear on the left edge, and vice versa. If horizontal wrapping is not enabled, a turtle will be unable to move off the right or left edge.

1. What is the logical "shape" of the NetLogo world, if wrapping is turned on horizontally, but not vertically?
2. What is the shape of the NetLogo world, if wrapping is turned on vertically, but not horizontally?
3. What is the shape of the NetLogo world, if wrapping is turned on both vertically and horizontally?

Programming in general: teaching the computer

Although computers (more precisely, the processors inside computers) are capable of manipulating data very efficiently, and though modern processors include floating-point processing units that can perform impressive arithmetic, trigonometric, and logarithmic calculations, they're also simple-minded: they're generally incapable of performing most tasks the average user would consider meaningful – *until they're taught to do these meaningful tasks*. We teach computers to do this through programming: encoding an algorithm (a procedure for completing a task or solving a problem) into a form that the computer can understand, for which it will take specified inputs, and from which it can present a meaningful result as output.

Fortunately for us, virtually every modern, commercially-available computer comes with millions of lines of these algorithmic instructions already written, and preloaded on hard drives, programmable memory chips, etc. These instructions make up the operating system (which lets us read and write data from and to the keyboard, display, and files), drivers (which tell the computer how to connect to and make use of hardware devices – e.g. video display adapters, disk drives, printers, external memory devices), and applications (special files which can be executed on demand by the user, for more specific functionality). We can augment this further by installing or writing new programs for the computer to execute; when we do this, we're literally teaching the computer to perform new tasks.

Some computer programs are instruction translators: they allow programmers to write new programs, without them having to understand much of the internal workings of the computer; these translators then convert the programmers' instructions into a form that the computer can execute. NetLogo is one such translator: it allows us to write programs in a specialized language to describe the behaviors of agents; it then converts these programs (NetLogo models) into a form the computer can execute,[1] without us having to know anything about how that conversion takes place. Nonetheless, we can still think of the NetLogo models we write as being sets of instructions that we teach to the computer; perhaps more usefully, we can think of our task, when building NetLogo models, as being that of teaching NetLogo itself.

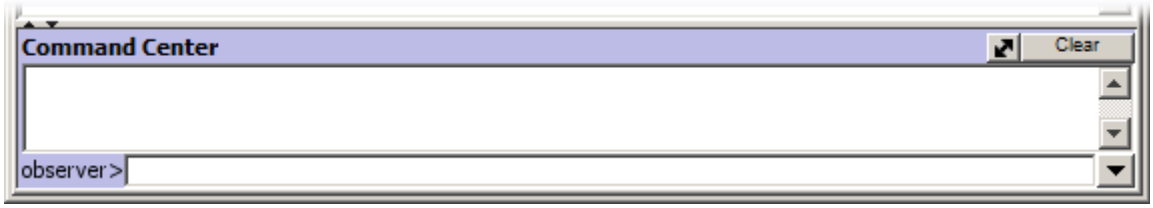
1 This is actually a slight over-simplification. NetLogo is built on top of Java, so it converts models into instructions which the Java Virtual Machine (JVM) can understand. As the model runs, the JVM translates those into instructions the computer hardware can execute directly.

Programming in NetLogo

Overview

We give instructions to NetLogo in three main ways:

1. We can type instructions in the **Command Center**:



   Any instructions we type in the input line of the **Command Center** are executed as soon as we press the *Enter* key – but they don't become part of what we're teaching NetLogo to do (i.e. the program[2] we're writing). In other words, we can use the **Command Center** to instruct NetLogo to perform actions it already knows how to do, but we can't use it to teach NetLogo new capabilities.

2. Some instructions can be included in buttons and other controls in the user interfaces we create. This functionality is most often used to connect the buttons we create to the new capabilities that we've taught NetLogo in our program.

3. Finally, and most importantly, when we write instructions in the **Code** window, we're creating a NetLogo program, consisting of one or more *procedures*. What we write in the **Code** window isn't executed immediately, but adds to what NetLogo knows how to do. We can invoke this new functionality through buttons and monitors in the user interface, by typing commands in the **Command Center**, or by referring to one or more of the new procedures in other code we write in the **Procedures** window.

Procedures

When you teach another person a procedure for completing some task, you might begin by saying: "To do X, first do A, then do B," and so on. Teaching NetLogo to perform some task is very similar: we use the keyword `to`, followed by the name of the task, and

2 "Model" and "program" are often used interchangeably when talking about NetLogo. Here, we'll distinguish between them by using "program" to refer to the contents of the **Code** window, and "model" to refer to the combined contents of the **Interface**, **Info**, and **Code** windows – i.e. the procedures, the user interface, and any embedded user information and documentation. Of course, there's also a more general – and very important – meaning of "model": a representation – abstracted or idealized to some extent – of a real world entity or system.

then the set of instructions that make up the procedure; finally, we indicate that there are no more instructions for this task by ending the procedure with the keyword **end**.

For example, here we teach NetLogo a procedure a turtle can follow, to draw a square:

```
to draw-square
  pen-down
  repeat 4 [
    forward 10
    right 90
  ]
  pen-up
end
```

Note that there are hyphenated words in the code of this example. Though this isn't allowed in most programming languages, it's valid and common in Logo dialects, and there are a number of *primitives* (built-in commands and reporters) with hyphenated names. However, while procedure and variable names can include hyphens – as well as many other punctuation symbols – they can't include spaces.

Now that we've written the **draw-square** procedure, we can invoke it by name in the **Command Center**, in a button, or in another procedure.

Most programming languages support two different – but not mutually exclusive – kinds of procedures (also called *functions*, *methods*, *subroutines*, etc.): those that modify the state of the system, and those that compute and return a result. The procedure above is an example of the former: it modifies the heading and position of an agent, but doesn't return a result. In NetLogo, this type of procedure is called a *command procedure*. We can also write *reporter procedures*, which return results. For example, the following reporter procedure computes and returns the square of a specified number:

```
to-report square [input-value]
  report (input-value * input-value)
end
```

The syntax for a reporter procedure differs from that of a command procedure in two key aspects:

1. The definition of a reporter procedure begins with **to-report**, instead of **to**.
2. The *command primitive* (built-in command) **report** is used to exit and return a value from a reporter procedure.

As seen above, input parameters can be included in the definition of a procedure (command or reporter) by enclosing them in square brackets after the procedure name.

Commands, Reporters, Definitions, and Declarations

Previously, we saw that we can create command and reporter procedures in our code. In fact, we can view any NetLogo program as consisting of commands, reporters, definitions, and declarations.

Commands are instructions that invoke either a command procedure or a *command primitive* (a command defined by NetLogo, rather than by a command procedure we've written), specifying any input parameters required by the primitive or procedure. For example, `forward 5` is a command that instructs a turtle to move 5 steps in the direction it is facing; here, the command primitive is `forward`, and the input required is a numeric value indicating the distance to move. Of course, while 5 is a simple literal numeric value, we could have written `forward (2 + 3)` instead, and the result would be the same. Similarly, if the variable `step-length` exists in our program, and the current value of `step-length` is 5, then `forward step-length` will also result in the given turtle moving 5 steps ahead.

What do `5`, `(2 + 3)`, and `step-length` all have in common? They're all expressions that NetLogo can evaluate – that is, from which NetLogo can compute a value. Generically, and in most programming languages, expressions like these are called just that: *expressions*. In NetLogo, we call them *reporters*. So a reporter is either a simple literal value, a variable reference, an invocation of a reporter procedure or *reporter primitives* (a reporter defined by NetLogo itself) with the required inputs – which are themselves reporters – or a combination of these, using arithmetic or logical operators (which are also reporter primitives) to compute the result. (With any of the above possibilities, parentheses may be used for specifying the order of evaluation explicitly – or even just for visual clarity.)

The reporter procedure example in the last section includes the line

```
report (input-value * input-value)
```

Here, the entire line is a command, invoking the command primitive `report`, and supplying the input that `report` expects. That input is the reporter

```
(input-value * input-value)
```

This reporter consists of parentheses (used here to make it clear that however complicated the reporter, we're computing and reporting a single value) surrounding the reporter

```
input-value * input-value
```

The arithmetic operator **`*`** is a reporter primitive of a special type: an *infix* reporter (one for which inputs are required before and after). So this reporter consists of the reporter primitive **`*`**, with **`input-value`** (a variable reference – and thus a reporter) specified for both of its required inputs.

We can go a long way with command and reporter primitives – but we can't actually write NetLogo programs until we start defining new procedures. As we saw previously, we do that with the **`to`** or **`to-report`** keyword, followed by the name of the procedure, optionally followed by a bracketed list of input parameters. Then, after the commands that make up the body of the procedure, we use the **`end`** keyword to indicate to NetLogo that the definition of the procedure is complete.

Finally, most non-trivial NetLogo programs also require *declarations*. These are special statements written at the start of our code (before any procedures) that declare essential information about the program to NetLogo itself: the global variables that will be assigned and referred to in the code; the *breeds* (different kinds) of turtle and link agents it will use; the variables that will be the attributes of our agents (beyond those predefined by NetLogo); the NetLogo extensions and additional source files that our program requires. We won't explore these further in this overview; for now, simply note that these declarations employ one or more of the **`globals`**, **`breed`**, **`undirected-link-breed`**, **`directed-link-breed`**, **`patches-own`**, **`turtles-own`**, **`links-own`**, ***`breeds`*`-own`**, **`extensions`**, and **`__includes`** keywords.

One important command primitive is actually a combination of a declaration and a command: The **`let`** command is used inside a procedure or *command block* (a sequence of commands enclosed by square brackets) to declare a local variable and assign to it an initial value. That value – and the variable itself – is preserved only within the procedure or command block where it is declared.

Types of Agents

There are four types of agents in NetLogo; each is capable of following different kinds of instructions, and each serves a different purpose in a NetLogo model:

1. **Observer** – There's always exactly one of this kind of agent. This agent is not displayed on the NetLogo world, but it is the only agent that can perform certain global operations in a model (e.g. **`clear-all`**, **`tick`**).

2. **Patches** – These are stationary agents; there's exactly one patch agent per square in the grid of the NetLogo world. A patch can't be displayed as any shape other than a square, but each patch can have its own color, as well as a label.

3. **Turtles** – These are mobile agents, capable of moving about the NetLogo world independently of other agents; any instructions that tell an agent to move can only be executed by turtle agents. The shape, color, size, and label of a turtle can be manipulated by the code of a NetLogo model.

4. **Links** – These are agents which connect one turtle to another. There are no instructions to move links directly; a link moves when one or both of the turtles at the endpoints move. (A link can also be configured as a *tie*, where motion of one endpoint turtle will automatically result in movement of the other endpoint turtle.) Links can be directed or undirected: with undirected links, we don't recognize the link as coming from one turtle to another, but simply that it is between the two; a directed link, on the other hand, is always *from* one turtle *to* another.

Links and turtles are the only agents that can be created or destroyed by the instructions contained within the model itself. Also, links and turtles are the only agents that can be organized into breeds.

Turtles can interact with other turtles by reading the attributes of those turtles, or by asking those turtles to execute instructions; they can also interact with patches in the same ways. Patches can interact with turtles, and with other patches. Links generally interact with their endpoint turtles, but they can also be made to interact with other links, turtles, and patches. The observer can ask turtles, patches, and links to perform specified operations. On the other hand, turtles, patches, and links can't explicitly ask the observer to perform any actions. However, models have global variables (some predefined by NetLogo, and others we can define in our sliders and program code); patches, links, and turtles can modify the values of some of these global variables – and the observer's actions may be affected by such changes.

Types of Data

NetLogo is a *weakly typed* (sometimes called *loosely typed*) programming language. When a variable is declared, no data type specification is included in the declaration; the variable can be used to store any of the supported types of data. In fact, over the lifetime of a given variable, it may contain data of different types at different moments (though that's usually not a good idea). Similarly, the definition of a reporter procedure doesn't specify the type of data returned by the procedure; it's possible (though generally not advisable) for a reporter procedure to return values of different types under different conditions.

NetLogo natively supports six types of data:

- **Numeric**

  The IEEE 754 standard double-precision floating-point format is used internally for all NetLogo numeric values [4]. This data type uses 64 bits to represent exact integers in the range $\left[-2^{53}, 2^{53}\right]$ and floating-point values (not exactly, for the most part) in the range $\left[-1.797693\times10^{308}, 1.797693\times10^{308}\right]$.

- **Boolean**

  This is a type containing only the values **`true`** and **`false`**. Note that these are not the string values "true" and "false"; neither are they interchangeable with the integer values 1 and 0, respectively.

- **String** (text)

  A *string* is a sequence of characters, which may include letters, digits, punctuation marks and other symbols, as well as white space. Note that NetLogo's string-handling primitives are quite minimal: for general-purpose text processing, NetLogo is rarely the language of choice. In particular, NetLogo has no built-in capability to convert a string of digits and other characters used for representing numbers into the corresponding numeric value.

  Another way we might think about NetLogo strings is as lists of characters (see the List datatype, p. 14): many of the list-oriented primitives also operate on strings. For this reason and others, it can be useful to think of strings not as a separate type of data structure, but as a specialized list.

- **Agent**

  Variables can store (and reporters can return) references to any given agent besides the observer. For example, in an ecosystem model with birth and death processes, it might be useful to have each turtle agent (representing some individual member of the ecosystem) remember its parents – i.e. maintain references to them in variables.

  If a variable is storing a reference to a turtle or link agent, and that agent dies, the variable will automatically be updated to contain the value **`nobody`** – a special reference that in fact refers to no agent.

- **List**

  The *list* is the fundamental *data structure* (a compound data type, possibly containing multiple components) in NetLogo – and in most other Logo dialects, for that matter. A list is an ordered sequence of zero or more elements. Elements can be appended to

(added to the end of) or prepended to (inserted at the beginning of) a list. Each time a given list is traversed (with no intervening modifications to the list contents), the order of the elements will be consistent from traversal to traversal.

Heterogeneity is another important property of lists in the NetLogo implementation: Each element can be of any of the available types (including lists).

Partly due to the early influence of Lisp on Logo, and partly due to later cross-pollination from Python and other languages, NetLogo includes a rich set of list-processing primitives. Even many NetLogo models that don't use most of these primitives, or that don't explicitly assign list values to variables, still employ lists implicitly – e.g. as inputs to reporters that compute aggregate statistics (**`count`**, **`max`**, **`mean`**, **`min`**, **`sum`**, etc.). Many such implicit instances of list usage are encountered when obtaining values from all agents in a set (below).

- **Agentset** (set of agents)

  The *agentset* is the second fundamental NetLogo data structure. As the name indicates, an agentset is a set of agents.

  Most programming languages and libraries that implement lists and sets differentiate between these two types in much the same way that we differentiate between their corresponding mathematical concepts:

  - While a list is ordered, a set is not. When adding a member to a set, we have no control over the position of that member within the set – in fact, "position" is meaningless for sets. If we traverse a set multiple times, the order of traversal may be different each time.
  - A list can contain the same value in multiple positions within the list. A set either contains a given value, or it doesn't – it cannot contain multiple instances of the same value simultaneously.

  In the NetLogo implementation, sets have some important limitations:

  - A set can contain only agents. NetLogo doesn't support sets of numeric values, sets of strings, sets of sets, etc. This is why NetLogo sets are called *agentsets*.
  - Agentsets are homogenous: Only one type of agent can be contained in an agentset at any given moment. So an agentset can't (for example) contain both patch and turtle agents simultaneously. (Also, just as simple agent variables can't refer to the observer agent, a reference to the observer can't be included in an agentset.)

Even with these constraints, NetLogo agentsets are very useful – and mastering agentsets is an essential part of becoming a skilled NetLogo practitioner. An agentset can be filtered by a variety of logical predicates; it can easily be traversed for the purpose of obtaining information from all the members (this produces a list of values); just as easily, it can be traversed for the purpose of instructing each member to perform one or more commands; it can be combined with another agentset, constructing a new set from the union, intersection, or difference of the two. Turtle and link breeds can even viewed as special agentsets, with NetLogo taking care of managing membership automatically.

In addition to the above data types, NetLogo extensions can define new simple and structured data types; references to instances of these data types can then be assigned to variables, or returned by reporter procedures. Notable examples of this are the data structures defined by the Table, Array, and GIS extensions, which are included in the standard NetLogo installation.

References

[1] Wilensky, U. *NetLogo*, 2015. [Online]. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. [Accessed: 24 Aug. 2015].

[2] Logo Programming, 2014. [Online]. Available: http://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html. The Logo Foundation, Cambridge, MA. [Accessed: 24 Aug. 2015].

[3] StarLogo TNG, 2015. [Online]. Available: http://education.mit.edu/portfolio_page/starlogo-tng/. MIT Scheller Teacher Education Program, Cambridge, MA. [Accessed: 24 Aug. 2015].

[4] "Double-precision floating-point format." 15 Aug. 2015. [Online]. Available: http://en.wikipedia.org/wiki/Double-precision_floating-point_format. Wikipedia. [Accessed: 24 Aug. 2015].