# NetLogo Tutorial Series:
# Set Theory Concepts and Applications

Nicholas Bennett
nickbenn@g-r-c.com

October 2015

## *Copyright and license*



## *Typographic conventions*

- ***Section headings: Palatino Linotype Bold Italic 14pt on white background***
- **Topic headings: Palatino Linotype Bold 12pt on white background**
- Body text: Palatino Linotype 12pt on white background
- Mathematical expressions: Times New Roman 12pt on white background
- NetLogo-specific content: Palatino Linotype 12pt on gray background
- **NetLogo expressions and commands: DejaVu Sans Mono Bold 11pt on gray background**
  - `Command primitives`
  - `Reporter primitives`
  - `Keywords`
  - `Literal values and predefined symbolic constants`
  - `Code comments`
  - `Breeds, procedures, variables`
  - ***`Placeholders`***

## *Introduction*

### Why use sets in NetLogo?

Compared to some other programming languages, the variety of data types supported by NetLogo is relatively small – especially when it comes to *structured data* (compositions of multiple data items – each of which is of a simple or structured type). Besides the agents themselves, which we can view as structured data with behaviors, the only data structures natively understood by NetLogo are *agentsets* and *lists*. But while there are many NetLogo models that don't use lists in any visible fashion, nearly all of them use agentsets – and many otherwise complicated tasks become much simpler if we leverage NetLogo's set-oriented facilities. Because of this, an understanding of agentsets is essential in mastering NetLogo.

## *Basic concepts*

### Definition

A *set* is an *unordered* collection of zero or more distinct *elements* (numbers, events, symbols, objects, etc. – even sets).

A set may be *finite* (i.e. containing a finite number of elements), or *infinite*.

The most direct way we can specify a finite set is by listing its elements in curly braces. For example, we can specify the set of natural numbers from 1 to 10 (inclusive) as

$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

We can replace some elements with an ellipses, if the meaning is clear:

$\{1, 2, 3, \ldots, 10\}$.

These are two examples of defining a set *by roster* – i.e. enumerating its elements.[1] Another common technique is the use of *set-builder notation*, in which we specify a set symbolically as a subset of some other (and generally well understood) set, or as a *mapping* (transformation) of the contents of a different set.[2] (See “Predicates”, p. 13, and “Set mapping”, p. 19.)

1 Even though sets are unordered, when using explicit enumeration to define a set, we typically do so in natural order – e.g. if the set contains numbers, they're usually listed in ascending order.

In NetLogo, a set can only contain agents (thus the term *agentset*) and can only contain agents of a single type at any given time. So we can have sets of turtles, sets of patches, and sets of links – but we can't define a set that simultaneously contains two or more types of agents. (By the way, the observer can't be a member of a NetLogo agentset.)

There are three standard agentsets that are predefined and managed automatically for us: **turtles**, **patches**, and **links**. For example, whenever we create a turtle, it's automatically added to **turtles**. Not surprisingly, when the turtle dies (by executing the **die** command), it's removed automatically from **turtles**; similarly, dead link agents are automatically removed from **links**. (We can't directly create or destroy individual patch agents in our code, so the contents of the **patches** set are constant, except when the world is resized.)

In addition to the three predefined agentsets, we can create others. When we use the **breed**, **undirected-link-breed**, or **directed-link-breed** keyword to declare a new turtle or link breed, we're actually declaring a new agentset, and declaring the type of agent that it will contain.[2] For example,

```
breed [ants ant]
```

declares the agentset **ants**, which will automatically include all turtles of the **ants** breed. Whenever we use **create-ants**, **sprout-ants**, or **hatch-ants** to create one or more turtles of the **ants** breed, or when we explicitly the set the breed of a turtle using **set breed ants**, NetLogo automatically updates the contents of the **ants** agentset to include the given turtle. Also, just as it does for the predefined agentsets, NetLogo automatically removes dead turtles and links from the breed-based agentsets.

In addition to the predefined agentsets, and the automatic agentsets defined when we declare breeds, we can declare and store our own agentsets in global, local, or agent variables. However, the only automatic updates that NetLogo performs on these sets is the removal of dead agents.

For example, in one of our procedures, we could have a local variable referring to the patches that are neighbors of the current patch, and that have turtles on them:

```
let occupied-neighbors (neighbors with [any? turtles-here])
```

2 For programmers with experience in languages that have class-based objects (e.g. C++, Java, Python, C#), a common source of confusion is the tendency to think of NetLogo breeds as classes. Given that breeds provide neither inheritance nor polymorphism nor encapsulation, there's little value in viewing them as classes. It's much more useful – and accurate – to see them as sets.

After this code is executed, the local variable **occupied-neighbors** is an agentset, due to the fact that we've assigned an agentset value to it. (The **with** reporter is discussed more completely in "Predicates", p. 13.)

The example above creates an agentset as a subset of an existing set. We can also create an agentset by assembling it from pieces – individual agents, lists of agents, other agentsets, etc. The **turtle-set**, **patch-set**, and **link-set** reporter primitives can be used for this purpose. For example, if we wish to declare a local variable **neighborhood** and assign to it the set of patches consisting of the current patch and the surrounding eight patches, we can use the following command:

```
let neighborhood (patch-set patch-here neighbors)
```

(For a turtle, **patch-here** reports the patch where the turtle is currently standing; **neighbors** reports the set of patches that are directly or diagonally adjacent to the current patch; **patch-set** treats each of its inputs as a set of patches, and reports the union of those sets. For more information, see "Set union", p. 9.)

### Set membership

If set ***S*** contains *A*, then *A* is an *element* (member) of ***S***, written $A \in \boldsymbol{S}$. Conversely, if ***S*** does not contain *A*, then *A* is not an element of ***S***, i.e. $A \notin \boldsymbol{S}$.

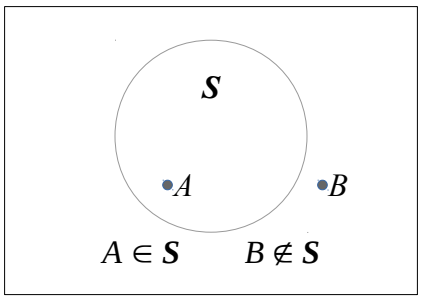


*Figure 1: Set membership*

In NetLogo, we use the **member?** reporter primitive to determine if some ***agent*** is a member of a specified ***agentset***.

```
member? agent agentset
```

reports **true** if ***agent*** is a member of ***agentset***, and **false** otherwise.

Membership testing of an agent is often done within the context of that agent – that is, by referring to the agent as **self**. An agent might perform some action if it is a member of a given set. For example,

```
if (member? self active-predators) [
  hunt
]
```

In the fragment above, the **hunt** procedure (presumably defined elsewhere in the code) would only be performed by the current agent if **member? self active-predators** reports **true** – that is, if the current agent is a member of the **active-predators** set.

Though a turtle or link can only be a member of a single breed, it can be a member of multiple non-breed agentsets simultaneously – if the logic of our model allows it.

### Empty set

A set is empty if it contains no elements. The *empty set* is denoted by $\emptyset$.

In NetLogo, the **any?** reporter primitive tells us whether a set is empty or not:

```
any? agentset
```

reports **true** if ***agentset*** is not empty, and **false** if it is empty.

One common use of **any?** is in model stopping conditions. Many models stop running a **go** procedure when there are no **turtles** left, or none left of a given breed. That is, they stop when some specified agentset is empty. This is typically done with code like this:

```
to go
  if not any? turtles [
    stop
  ]
  ...
end
```

Sometimes it's useful to construct empty agentsets in NetLogo – e.g. when initializing a variable that will later be used to contain a set of agents. The **no-turtles**, **no-patches**, and **no-links** reporter primitives report empty sets suitable for containing turtles, patches, and links, respectively.

### Set equality

Sets ***S*** and ***T*** are equal if every element of ***S*** is also an element of ***T***, and vice versa. (This is trivially true if neither ***S*** nor ***T*** has any elements. Thus, all empty sets are equal, mathematically speaking.)

NetLogo supports the use of the equals sign reporter primitive to compare sets for equality.

```
agentset-s = agentset-t
```

reports **true** if ***agentset-s*** and ***agentset-t*** contain all the same agents – that is, every agent in ***agentset-s*** is also in ***agentset-t***, and vice versa. If there is at least one agent not contained in common between the two agentsets, then the equality reporter will report **false**.

There's a twist to the equality of empty sets in NetLogo: an empty turtle set, empty patch set, and empty link set are not considered equal to each other. Essentially, when comparing two agentsets, NetLogo is comparing not only the sets' members, but also the types of agents the sets are capable of containing (for the standard sets or breed sets), or the types of agents they currently contain or most recently contained (for agentsets stored in global, agent, or local variables).

### Subset relation

If every element of ***S*** is also an element of ***T***, *whether or not the reverse is true*, then ***S*** is a *subset* of ***T***, written as $S \subseteq T$; equivalently, ***T*** is a *superset* of ***S***, written $T \supseteq S$ This implies that equal sets are subsets of each other, and that the empty set is a subset of all sets.

If ***S*** a subset of ***T***, and at least one element of ***T*** is *not* an element of ***S***, then ***S*** is a *proper subset* of ***T***, written $S \subset T$, and ***T*** is a *proper superset* of ***S***, or $T \supset S$.

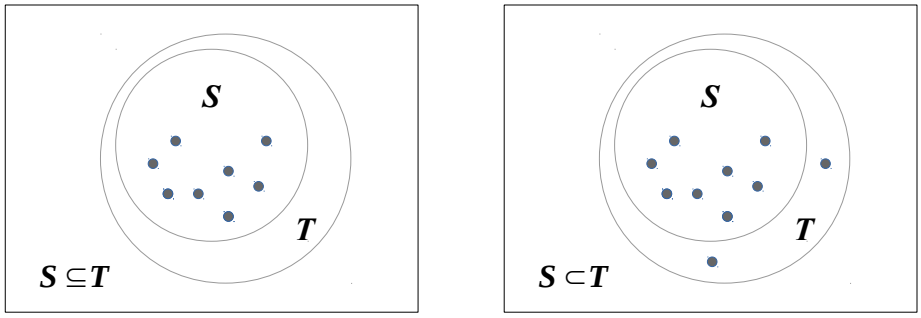


*Figure 2: Subsets and proper subsets*

The NetLogo **all?** reporter primitive – which provides the general capability to test all members of an agentset for some condition (expressed as a Boolean reporter), reporting **true** only if the condition is true for all of them – gives us a straightforward way to test whether ***agentset-S*** is a subset of ***agentset-T***:

```
all? agentset-S [member? self agentset-T]
```

reports **true** if all members of ***agentset-S*** are also members of ***agentset-T***, and **false** otherwise.

A concrete example of this would be testing to see whether all **turtles** on **orange** patches are members of the **students** breed – i.e. testing whether the set of **turtles** currently on **orange** patches is a subset of the **students** set. The following reporter expression returns **true** if this is the case, and **false** otherwise.

```
all? (turtles with [pcolor = orange]) [member? self students]
```

As is often the case, we can express this in multiple ways. For example,

```
all? (turtles with [pcolor = orange]) [breed = students]
```

This second form is arguably simpler when ***agentset-T*** is a breed – i.e. when we can take advantage of the fact that the **breed** variable of each turtle refers to the breed-based agentset to which it belongs.

Curiously, even though **no-turtles**, **no-patches**, and **no-links** are not considered equal to each other in NetLogo, they are considered subsets of each other, according to the above general reporter expression for testing subsets. For example,

```
all? no-turtles [member? self no-patches]
```

reports **true**.

### Cardinality of a finite set

The cardinality of a *finite* set $S$, written $|S|$, is equal to the number of elements in $S$.

The NetLogo **count** reporter primitive reports the number of members of an agentset:

```
count agentset
```

### Cardinality of an infinite set

Our intuition about the size of a set might not serve us well when dealing with infinite sets. For example, it can be shown that the cardinalities of some infinite sets are different from those of other infinite sets.

Infinite agentsets aren't possible in NetLogo – so we won't be dealing with them here.

### Set union

The union of two or more sets, written with the union operator $\cup$, is the set containing every element that is a member of at least one of the source sets. For example, if

$A \in \boldsymbol{S}$

and

$B \in \boldsymbol{T}$,

then

$A \in \boldsymbol{S} \cup \boldsymbol{T}$

and

$B \in \boldsymbol{S} \cup \boldsymbol{T}$.

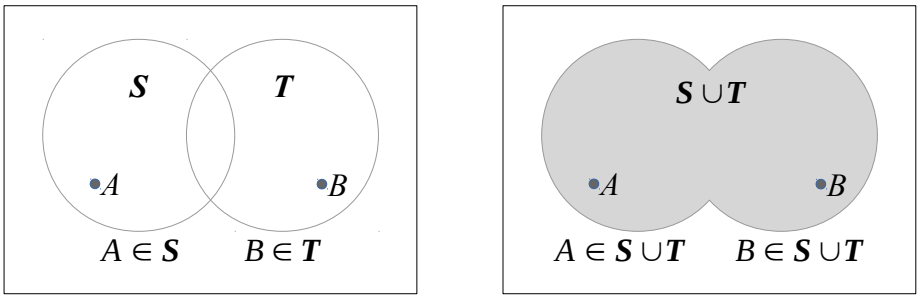


*Figure 3: Set union*

$\boldsymbol{S}_1 \cup \boldsymbol{S}_2 = \boldsymbol{S}_2 \cup \boldsymbol{S}_1$.

They are also associative:

$\left(\boldsymbol{S}_1 \cup \boldsymbol{S}_2\right) \cup \boldsymbol{S}_3 = \boldsymbol{S}_1 \cup \left(\boldsymbol{S}_2 \cup \boldsymbol{S}_3\right)$.

NetLogo provides three reporter primitives – **turtle-set**, **patch-set**, and **link-set** – that construct and report set unions.

```
(turtle-set turtleset-S turtleset-T)
```

reports the union of ***turtleset-S*** and ***turtleset-T***. Strictly speaking, the parentheses aren't required, as long as we're providing exactly two inputs to the **turtle-set** reporter; however, we can also provide zero, one, three, or more inputs to **turtle-set**, and parentheses *are* required in all of those cases – opening before the **turtle-set** reporter, and closing after the last input. To increase clarity and reduce the potential for errors in the code, we recommend using parentheses whenever you use the **turtle-set**, **patch-set**, or **link-set** reporters in your code.

In some models, we might have two or more breeds of turtles, all of whose members should execute the same commands. We could do this with multiple **ask** commands, or we could use a single **ask** command, addressing the union of the breeds. For example, if we have breeds with the plural names **dogs** and **cats**, and we have an **eat** command procedure applicable to both (but not to other breeds we've defined in the same model), we could use this code to ask all members of both breads to execute the command:

```
ask (turtle-set dogs cats) [
  eat
]
```

The **turtle-set**, **patch-set**, and **link-set** primitives are more flexible than we've seen so far. They can be used to construct agentsets from zero or more agents, agentsets, lists of agents, lists of agentsets, and virtually any combination of these. (**turtle-set**, **patch-set**, and **link-set** with no inputs are equivalent to **no-turtles**, **no-patches**, and **no-links** – i.e. they create empty sets of turtles, patches, and links, respectively.)

### Set intersection

The intersection of two or more sets, written with the intersection operator $\cap$, is the set containing every element that is a member of *all* of the specified sets. For example, if

$A \in \boldsymbol{S}$

and

$A \in \boldsymbol{T}$,

then

$A \in \boldsymbol{S} \cap \boldsymbol{T}$;

however, if

$B \in \boldsymbol{S}$,

but

$B \notin \boldsymbol{T}$,

then

$B \notin \boldsymbol{S} \cap \boldsymbol{T}$.

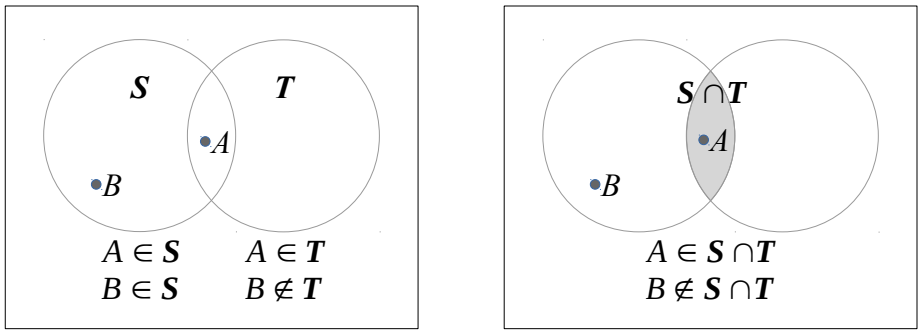


*Figure 4: Set intersection*

$\boldsymbol{S}_1 \cap \boldsymbol{S}_2 = \boldsymbol{S}_2 \cap \boldsymbol{S}_1$

Set intersection is also associative:

$\left(\boldsymbol{S}_1 \cap \boldsymbol{S}_2\right) \cap \boldsymbol{S}_3 = \boldsymbol{S}_1 \cap \left(\boldsymbol{S}_2 \cap \boldsymbol{S}_3\right)$.

Note that by definition, the intersection of two or more sets is a subset of all of those sets. That is,

$$\begin{aligned} \boldsymbol{S}_1 \cap \boldsymbol{S}_2 \cap \ldots &\subseteq \boldsymbol{S}_1 \\ &\subseteq \boldsymbol{S}_2 \\ &\ldots \end{aligned}$$

Keeping this in mind, it is sometimes easier to express (for example) the intersection of two sets as the subset of the first that also satisfies the condition of membership in the second.

Constructing and reporting set intersection is one of the many operations we can perform using the **with** reporter primitive. Given two agentsets, ***agentset-S*** and ***agentset-T***,

```
agentset-S with [member? self agentset-T]
```

reports the intersection of ***agentset-S*** and ***agentset-T***.

Since set intersection is commutative, we could also write

```
agentset-T with [member? self agentset-S]
```

and get the same result.[3]

For example, to get the intersection of the set of all **turtles** of the **ants** breed, and the set of **turtles** whose **color** is **green**, we could write

```
(turtles with [color = green]) with [member? self ants]
```

However, taking advantage of the fact that the intersection of two sets is a subset of each of those sets, we could express this far more simply as

```
ants with [color = green]
```

3 Since all turtles are members of the **turtles** agentset, we can also write

```
turtles with [member? self turtleset-S and member? self turtleset-T]
```

If we need the intersection of more than two sets, this form is sometimes easier to read.

## *Subset definition and creation*

### Predicates

As seen above, some subsets are defined by intersections of multiple sets. Others can be defined that way, but are more simply defined in other terms. More generally, we might define a logical condition, called a *predicate*, which must be satisfied by an element of a set in order for that element to be included in a given subset. In this approach, a subset is thus defined by the *domain* (the set from which the candidate elements are taken) and the predicate (which is used to select or reject members of the domain for membership in the subset).

In set-builder notation, we can express this as

$$\{A \in \boldsymbol{S} \mid \boldsymbol{P}(A)\}.$$

We read this as "the set of all elements $A$ of $\boldsymbol{S}$, such that $\boldsymbol{P}(A)$ is true". Equivalently, it's the set of all elements of $\boldsymbol{S}$ that satisfy the condition $\boldsymbol{P}$. Of course, if $\boldsymbol{P}$ is the condition of membership in another set, $\boldsymbol{T}$, then the desired subset is the intersection of $\boldsymbol{S}$ and $\boldsymbol{T}$.

In NetLogo, the **`with`** reporter primitive is directly equivalent to "such that" (the vertical bar in set-builder notation). For example, if we write

```
turtles with [color = green]
```

we are specifying the subset of **`turtles`**, such that the reporter expression **`color = green`** evaluates to **`true`** – i.e. the set of all green turtles. More generally,

```
agentset with [boolean-reporter]
```

creates and reports the subset of ***`agentset`*** for which ***`boolean-reporter`*** is true.

Given its general applicability to creating subsets, **`with`** is one of the most important reporter primitives in NetLogo. However, as we'll see next, some subset creation tasks are more simply expressed – and sometimes more efficiently computed – via special-purpose reporter primitives.

### Specialized subset creation

Using NetLogo's **with** reporter primitive, we can create virtually any agent subset we might need. However, there are a number of subset creation operations that are used frequently enough, or for which the required **with**-based expression is sufficiently cumbersome (or computationally inefficient), that specialized subset reporter primitives have been included in the language. The list that follows is not exhaustive, but it includes some of the most commonly used of these specialized subset reporters. For additional information, please refer to the NetLogo Dictionary.[3]

| Reporter | Reports … |
|---|---|
| **other** ***agentset*** | … the subset of ***agentset*** excluding the current agent. |
| **neighbors** | … the subset of **patches** that are directly or diagonally adjacent to the current patch. |
| **neighbors4** | … the subset of **patches** that are directly – but not diagonally – adjacent to the current patch. |
| **turtles-here**, ***breeds*****-here** | … the subset of **turtles** or ***breeds*** that are located on the current patch. |
| ***agentset*** **in-radius** ***distance*** | … the subset of ***agentset*** with a distance less than or equal to ***distance*** from the current agent. |
| ***agentset*** **in-cone** ***distance angle*** | … the subset of ***agentset*** with a distance less than or equal to ***distance*** from the current turtle, and located within a cone of ***angle*** degrees, with its vertex on the current turtle, and centered around its heading. |
| **turtles-on** ***agentset***<br>***breeds*****-on** ***agentset*** | … the subset of **turtles** or ***breeds*** standing on any of the patches in ***agentsent***, or sharing a patch with any of the turtles in ***agentset***. |
| **turtles-on** ***agent***<br>***breeds*****-on** ***agent*** | … the subset of **turtles** or ***breeds*** standing on ***agent*** (if it's a patch), or sharing a patch with ***agent*** (if it's a turtle). |

| | |
|---|---|
| **n-of** ***number agentset*** | … a random subset of ***agentset***, containing ***number*** elements, selected without replacement and with equal likelihood. |
| **min-n-of** ***number agentset*** **[*reporter*]**<br>**max-n-of** ***number agentset*** **[*reporter*]** | … the subset of ***agentset*** containing the ***number*** elements that reported the minimum (or maximum) values of ***reporter***. |
| ***agentset*** **with-min [*reporter*]**<br>***agentset*** **with-max [*reporter*]** | … the subset of ***agentset*** containing all agents that report the minimum (or maximum) value of ***reporter***. |
| **my-links**<br>**my-*breeds*** | … the subset of undirected **links** or ***breeds*** that are connected to the current turtle. |
| **my-in-links**<br>**my-in-*breeds*** | … the subset of directed **links** or ***breeds*** that originate with other turtles and terminate at the current turtle. |
| **my-out-links**<br>**my-out-*breeds*** | … the subset of directed **links** or ***breeds*** that originate with other turtles and terminate at the current turtle. |
| **link-neighbors**<br>***breeds*-neighbors** | … the subset of **turtles** linked to the current turtle via undirected **links** or ***breeds***. |
| **in-link-neighbors**<br>**in-*breeds*-neighbors** | … the subset of **turtles** linked to the current turtle via directed **links** or ***breeds*** terminating at the current turtle. |
| **out-link-neighbors**<br>**out-*breeds*-neighbors** | … the subset of **turtles** linked to the current turtle via directed **links** or ***breeds*** originating from the current turtle. |

## *Extracting and using set elements*

### Iterating over a set

Many set-oriented mathematical definitions and algorithms include assertions about all members of a set, or conditions to be tested for all members of a set, or operations to be performed on all members of a set. These are all forms of *iteration over a set* – performing an operation or evaluating an expression on all members of a set. We often see the ∀ symbol (read "for all" or "for any") used to indicate this kind of iteration.

One example of the use of ∀ is this definition of a convex set of points:

$$\boldsymbol{S} \text{ is convex } \Leftrightarrow (\alpha\, \boldsymbol{p}_1 + (1-\alpha)\, \boldsymbol{p}_2) \in \boldsymbol{S},\ \forall\, \boldsymbol{p}_{1,}\, \boldsymbol{p}_2 \in \boldsymbol{S},\ \alpha \in [0,\ 1]$$

We might read this as

> A set of points $\boldsymbol{S}$ is convex, if and only if *for all* pairs of points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ in $\boldsymbol{S}$, all points on the line segment connecting those two points – i.e. the points given by $(\alpha\, \boldsymbol{p}_1 + (1-\alpha)\, \boldsymbol{p}_2),\ \alpha \in [0,\ 1]$ – are also in the set.

In NetLogo, iteration over an agentset to perform a set of commands for every agent in the set is the role of the **ask** command:

```
ask agentset [
  commands
]
```

With **ask**, we specify a set of agents, ***agentset***, and the ***commands*** to be executed by each agent in the set; NetLogo iterates over the agentset and executes the commands. We don't have to concern ourselves with how many agents there are in ***agentset***, how they're stored in memory, or in what order.

For example, we could write

```
ask turtles [
  forward 1
]
```

In this code, all members of the **turtles** set will execute the **forward 1** command.

The **with** reporter represents another, implicit form of iteration: a predicate is applied to all members of an agentset, and only the agents for whom the predicate is true (i.e. that report a value of **true** for the reporter expression making up the predicate) are included in the subset reported by **with**.

Both **ask** and **with** – as well as some of the reporters listed in "Specialized subset creation", p. 14, and the **of** reporter described in "Mapping sets to lists", p. 20 – operate across the entire specified agentset, but some useful set iteration primitives are *short-circuit* reporters. For example, we can use **all?** to test that some condition holds for all members of the specified agentset:

```
all? agentset [boolean-reporter]
```

reports **true** if *all* members of ***agentset*** report a value of **true** for ***boolean-reporter*** – that is, if the condition represented by ***boolean-reporter*** holds for all members of ***agentset***. On the other hand, if even one member of ***agentset*** reports a value of **false** for ***boolean-reporter***, then the value of the entire **all?** reporter expression is **false**. When that happens, there's no point in evaluating ***boolean-reporter*** for any remaining agents in ***agentset***, so NetLogo stops iterating immediately and reports **false**.

Keep in mind that for all of these set iteration operations, the order of iteration over the members of a set is not under our control, and is officially undefined. (Remember that sets are unordered.) In fact, NetLogo shuffles the members of an agentset into a random order before executing **ask** or evaluating **with**, **all?**, or **of**, and before returning results from the specialized subset creation reporters. This shuffling is useful for avoiding strange effects that are possible when some agents consistently act before others.

(There are some models for which a fixed order of iteration is important, but these are relatively uncommon. We can deal with these cases by constructing a list of agents from the given agentset, and then iterating over the list. See "Mapping sets to lists" on p. 20 for more information on constructing agent lists from agentsets.)

**Finding an extreme-valued member of a set**

In some algorithms, we assume the existence of a *min* or *max* function, capable of iterating over the members of a set and returning the member with the minimum or maximum value of some attribute or function. How such iterations are performed (and indeed, how the set contents should be stored and accessed, in order to perform these searches as efficiently as possible) is generally left unspecified; these details are usually not relevant in a high-level description of many algorithms.

Just as the **min-n-of** and **max-n-of** reporters are used to report a subset of an agentset with extreme values (see "Specialized subset creation", p. 14), **min-one-of** and **max-one-of** report single agents with extreme values.

```
min-one-of agentset [reporter]
```

or

```
max-one-of agentset [reporter]
```

reports the member of ***agentset*** with the minimum or maximum value (respectively) of ***reporter***. If there are multiple agents with the minimum or maximum value, ties are broken randomly.

One typical use of **min-one-of** or **max-one-of** is in setting the direction of movement (**heading**) of a turtle, according to information taken from the patches in its vicinity. For example, assume we have a patch variable **altitude**, and we want the turtle to move in the direction of the patch in the local neighborhood that has the highest value of this variable. The **uphill** command gives us one approach to this; in some cases, however, it's too simplistic for the desired movement behavior. For example, we might want to constrain the turtle's step size. So as an alternative to **uphill**, we might use code like this:

```
let target max-one-of (patch-set patch-here neighbors) [altitude]
face target
ifelse (distance target < 1) [
  forward distance target
]
[
  forward 1
]
```

**Selecting a member of a set at random**

Sometimes, rather than selecting the element of a set with an extreme value, we simply want to select an element at random, with all elements of the set equally likely to be selected. Once again, the ability to do this selection is generally assumed by algorithms relying on this functionality; how the random selection is performed is generally left unspecified.

An example of the use of such a random element selection is found in *Prim's algorithm*

for finding the minimum spanning tree of a graph or network. This algorithm starts by selecting one of the set of nodes of the graph at random, using it as the initial node in the spanning tree.

In NetLogo, the **one-of** reporter selects and reports one member of an agentset (or one element of a list) at random, using the syntax

```
one-of agentset
```

This reporter is used in a wide variety of models. For example, in the **Sample Models/Biology/Evolution/Genetic Drift** section of the NetLogo Models Library, the GenDrift P local model includes the following code:[4]

```
ask patches [
  ;; each patch randomly picks a neighboring patch
  ;; to copy a color from
  set pcolor [pcolor] of one-of neighbors
]
```

Here, **one-of neighbors** reports one of the eight patches diagonally or directly adjacent to the current patch, selected at random with equal likelihood. **pcolor** (the patch color) of the selected patch is then assigned to the **pcolor** variable of the current patch.

## *Set mapping*

### Definition

A *set mapping* is a function that takes each member of a set and returns a value that (depending on the mapping function) is either an element of the same set or an element of a different set.

For example, we might have

$$x \in \mathbb{Z}$$
$$f(x) = |x|$$

where $\mathbb{Z}$ is the set of all integers. $f(x)$ is then a mapping – using the absolute value – of $\mathbb{Z}$ onto $\mathbb{N}$, the set of natural numbers (including 0). In set-builder notation, we might write this as

$$\{|x| : x \in \mathbb{Z}\}.$$

### Mapping sets to lists

It's easy to imagine mappings that would be useful in NetLogo. For example, we might want to get the set of all patches that have green turtles on them (possibly in order to **ask** those patches to do something); one approach to this would be to perform a mapping from the set of green turtles (i.e. the subset of turtles whose color is green) to the patches they're standing on. For another example, we might want to get the set of ($X$, $Y$) coordinate pairs of every red turtle, in order to compute their *centroid* (center of mass, or average locaton – the flocking models in the NetLogo Models Library perform such a computation); this is a mapping from a turtle subset to a set of numeric pairs.

There's a complication, however: In NetLogo, as noted previously, sets can only have agents as elements. So while it might be useful to map the **turtles** agentset (or a subset) to a set of ($X$, $Y$) coordinate pairs, the latter set isn't possible in NetLogo. Instead, NetLogo supports mapping of a set to a *list* – an ordered sequence of values.

```
[reporter] of agentset
```

reports a list of values, where each is the result of an individual member of ***agentset*** evaluating ***reporter***. (Note that ***reporter*** can be very simple – e.g. the turtle variable **color** – or a more complicated expression involving reporter primitives, agent variables, global variables, local variables, reporter procedures, etc.)

In general, the list returned by **[*reporter*] of *agentset*** is in a shuffled order. That is, while a list is an ordered structure, the order in which agentset members are mapped to list elements is random. For example, if we have 3 turtles, with **who** values **0**, **1**, and **2**, then

```
[who] of turtles
```

reports a list containing the values **0**, **1**, and **2** – but potentially in a different order every time **[who] of turtles** is evaluated. We can see this by running the following from the command center, with the **observer** agent selected:

```
clear-all
create-turtles 3
repeat 4 [show [who] of turtles]
```

This will produce output resembling (but probably not matching exactly, due to the shuffling of turtles)

```
observer: [1 0 2]
observer: [2 0 1]
observer: [1 2 0]
observer: [0 2 1]
```

In order to get a list of the ($X$, $Y$) coordinate pairs of all red turtles then (for example), we could use the reporter expression

```
[(list xcor ycor)] of turtles with [color = red]
```

This would return a list, in random order, of the coordinates of all red turtles, where each element in the final list would itself be a list with 2 items – namely, the $X$ and $Y$ coordinates of the corresponding turtle.

As noted in "Iterating over a set" (p. 16), one application of mapping an agentset to a list is to obtain a list of agents over which we can iterate in a fixed, consistent order. For this purpose, the form of the mapping reporter is usually

```
[self] of agentset
```

Once we have a list of agents, we can iterate over it using list-oriented reporter primitives such as **foreach**, **map**, **filter**, etc.

**Mapping agentsets to agentsets**

In the event that we do want to map one set of agents to another (as in our previous example, where we wanted the set of all patches with green turtles on them), we can use one of the set construction reporters – **turtle-set**, **patch-set**, or **link-set** – on a list of agents obtained from **[*reporter*] of *agentset***. For example,

```
(patch-set [patch-here] of turtles with [color = green])
```

reports a set of patches, constructed from the list of patches on which the green turtles are standing. Here, we see more evidence that the functionality of **turtle-set**, **patch-set**, and **link-set** goes far beyond reporting the unions of agentsets. In this case, it's being used to construct an agentset from a list of agents; as it does so, it discards any duplicates it finds in the list, since sets can contain only distinct values. (Note that because we're providing only one input – a list, in this case – to **patch-set**, and since **patch-set** expects 2 inputs by default, we *must* enclose it and its input in parentheses, to tell NetLogo that we're only providing a single input.)

Of course, as in most programming languages, there are multiple ways to perform many tasks in NetLogo. Sometimes, an agentset-to-agentset mapping can be expressed more simply by creating a subset of one of the agentsets. For example, the following reporter expression returns exactly the same set of patches as the previous expression.

```
patches with [any? turtles-here with [color = green]]
```

### *Summary*

Besides NetLogo, there are several other programming languages with native support for sets and other high-level data structures. In almost all of these, there are usually multiple ways to accomplish any given task; NetLogo is no exception in this regard. However, there are few programming languages in which sets and set-oriented operations are as central to the language as they are in NetLogo.

In general, the more that we can leverage the high-level facilities of any programming language, the more we are able to focus on *what* our programs should do, instead of focusing on *how* they should do it. This is especially true in NetLogo, where agents are automatically organized into sets, and where we have a very rich vocabulary of command and reporter primitives for dealing with these agentsets, and for creating our own.

## *References*


[1] U. Wilensky, *NetLogo*, 1999. [Online]. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed: 18 Aug. 2015].

[2] B. Simmons, "Set-Builder Notation", *Mathwords*, 28 Jul. 2014. [Online]. Available: http://www.mathwords.com/s/set_builder_notation.htm. [Accessed: 18 Aug. 2015].

[3] U. Wilensky, "NetLogo Dictionary", *NetLogo User Manual*, 3 Apr. 2015. [Online]. Available: http://ccl.northwestern.edu/netlogo/docs/dictionary.html. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed: 18 Aug. 2015].

[4] U. Wilensky, "NetLogo GenDrift P local mode", 1997. [Online]. Available: http://ccl.northwestern.edu/netlogo/models/GenDriftPlocal. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed: 18 Aug. 2015].