NETLOGO 3.1.4. – QUICK GUIDE

Luis R. Izquierdo

Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. There are three types of agents:

- **Turtles**. Turtles are agents that move around in the world.
- **Patches**: The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move.
- The **observer**: The observer doesn't have a location - you can imagine it as looking out over the world of turtles and patches.

Instructions

Instructions tell agents what to do. There are three characteristics that are useful to remember about instructions:

a) Whether the instruction is implemented by the user (**procedures**), or whether it is built into NetLogo (**primitives**). Once you define a procedure, you can use it elsewhere in your program. The Primitives Dictionary has a complete list of built-in instructions.

```
to setup
  clear-all                    ;; clear the world
  create-turtles 10            ;; make 10 new turtles
end
```

In this program, **setup** is a procedure, whereas **clear-all** and **create-turtles** are both primitives.

b) Whether the instruction produces an output (**reporters**) or not (**commands**).

- A reporter computes a result and **reports** it. Most reporters are nouns or noun phrases (e.g. “average-wealth”, “best-known-farmer”). These names are preceded by the keyword **to-report**. The keyword **end** marks the end of the instructions in the procedure.

```
to-report average-wealth
  report mean values-from turtles [wealth]
end
```

- A command is an action for an agent to carry out. Most commands begin with verbs (e.g. "create", "die", "jump", "inspect", "clear"). These names are preceded by the keyword **to** (instead of **to-report**). The keyword **end** marks the end of the instructions in the procedure.

```
to go
  ask turtles
    [ forward 1            ;; all turtles move forward one step
      right random 360 ]   ;; ...and turn a random amount
end
```

c) Whether the instruction takes an input (or several inputs) or not. Inputs are values that the instruction uses in carrying out its actions.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report 0 - number ]
end
```

Variables

Variables are places to store values (such as numbers). A variable can be a **global** variable, a **turtle** variable, a **patch** variable, or a **local** variable (local to a procedure). To change the value of a variable you can use the **set** command (If you don't set the variable to any value, it starts out storing a value of zero).

a) **Global variables**: If a variable is a global variable, there is only one value for the variable, and every agent can access it. You can make a global variable by adding a switch or a slider to your model, or by using the **globals** keyword at the beginning of your code, like this:

```
globals [ clock ]
```

b) **Turtle and patch variables**: Each turtle has its own value for every turtle variable, and each patch has its own value for every patch variable. Turtle (and patch) variables can be built-in or defined by the user.

- Built-in turtle and patch variables: For example, all turtles have a **color** variable, and all patches have a **pcolor** variable. If you set the variable, the turtle or patch changes colour. Other built-in turtle variables are **xcor**, **ycor**, and **heading**. Other built-in patch variables include **pxcor** and **pycor**. You can find the complete list in the primitives dictionary.
- User-defined turtle and patch variables: You can also define new turtle and patch variables using the **turtles-own** and **patches-own** keywords, like this:

```
turtles-own [ energy speed ]
patches-own [ friction ]
```

c) **Local variables**: A local variable is defined and used only in the context of a particular procedure or part of a procedure. To create a local variable, use the **let** command. You can use this command anywhere. If you use it at the top of a procedure, the variable will exist throughout the procedure. If you use it inside a set of square brackets, for example inside an **ask**, then it will exist only inside those brackets.

```
to swap-colors [turtle1 turtle2]
   let temp color-of turtle1
   set (color-of turtle1) (color-of turtle2)
   set (color-of turtle2) temp
end
```

Setting and reading the value of variables

Global variables can be read and set at any time by any agent. As well, a turtle can read and set patch variables of the patch it is standing on. For example, this code:

```
ask turtles [ set pcolor red ]
```

causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you can't have a turtle variable and a patch variable with the same name.)

In other situations where you want an agent to read or set a different agent's variable, you put **`-of`** after the variable name and then specify which agent you mean. Examples:

```
set color-of turtle 5 red
                       ;; turtle with ID number 5 turns red

set pcolor-of patch 2 3 green
                       ;; patch with pxcor = 2 and pycor = 3 turns green

ask turtles [ set pcolor-of patch-at 1 0 blue ]
                       ;; every turtle turns the patch to its east blue

ask patches with [any? turtles-here]
  [ set color-of one-of turtles-here yellow ]
                       ;; on every patch, a random turtle turns yellow
```

Ask

NetLogo uses the **`ask`** command to specify commands that are to be run by turtles or patches. Usually, the observer uses **`ask`** to ask all turtles or all patches to run commands. Here's an example of the use of ask syntax in a NetLogo procedure:

```
to setup
        clear-all                      ;; clear the world
        create-turtles 100             ;; create 100 new turtles
        ask turtles
            [   set color red          ;; turn them red
                right random-float 360 ;; give them random headings
                forward 50 ]           ;; spread them around
        ask patches
            [   if (pxcor > 0)         ;; patches on the right side
                [ set pcolor green ] ] ;; of the view turn green
end
```

You can also use **`ask`** to have an individual turtle or patch run commands. The reporters **`turtle`**, **`patch`**, and **`patch-at`** are useful for this technique. For example:

```
to setup
  clear-all
  create-turtles 3             ;; make 3 turtles
  ask turtle 0                 ;; tell the first one...
    [ fd 1 ]                   ;; ...to go forward

  ask turtle 1                 ;; tell the second one...
    [ set color green ]        ;; ...to become green
  ask patch 2 -2               ;; ask the patch at (2,-2)
    [ set pcolor blue ]        ;; ...to become blue
  ask turtle 0                 ;; ask the first turtle
    [ ask patch-at 1 0         ;; ...to ask patch to the east
      [ set pcolor red ] ]     ;; ...to become red
end
```

Agentsets

An agentset is a set of agents that can contain either turtles or patches, but not both at once. An agentset is not in any particular order. In fact, it's always in a random order[1]. What's powerful about the agentset concept is that you can construct agentsets that contain only *some* turtles or *some* patches. For example, all the red turtles, or the patches with **pxcor** evenly divisible by five. These agentsets can then be used by **ask** or by various reporters that take agentsets as inputs.

One way is to use **turtles-here** or **turtles-at**, to make an agentset containing only the turtles on my patch, or only the turtles on some other particular patch. Here are some more examples of how to make agentsets:

```
turtles with [color = red]             ;; all red turtles

turtles-here with [color = red]        ;; all red turtles on my patch

patches with [pxcor > 0]               ;; patches on right side of view

turtles in-radius 3    ;; all turtles less than 3 patches away

patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
               ;; the four patches to the east, north, west, and south

neighbors4             ;; shorthand for those four patches
```

Once you have created an agentset, here are some simple things you can do:

- Use **ask** to make the agents in the agentset do something.
- Use **any?** to see if the agentset is empty.
- Use **count** to find out exactly how many agents are in the set.

Here are some more complex things you can do:

```
set color-of one-of turtles green
       ;; one-of reports a random agent from an agentset

ask max-one-of turtles [sum assets] [ die ]
       ;; max-one-of agentset [reporter] reports an agent in the
       ;; agentset that has the highest value for the given reporter

show max values-from turtles [sum assets]
       ;; values-from agentset [reporter] reports a list that contains
       ;; the value of the reporter for each agent in the agentset.
```

Synchronization

In NetLogo, by default, commands are executed asynchronously; each turtle or patch does its list of commands as fast as it can. To be clear, consider the following code:

```
ask turtles
  [ forward random 10
    do-stuff ]
```

[1] If you want agents to do something in a fixed order, you can make a list of the agents instead.

Since the turtles will take varying amounts of time to move, they'll begin **do-stuff** at different times. If you want all turtles to wait after moving until all the other turtles are done moving, before executing **do-stuff**, then you can write it this way:

```
ask turtles [ fd random 10 ]
ask turtles [ do-stuff ]
```

Then the turtles all begin **do-stuff** at the same time. Finally, if you want agents to execute a set of commands in a fixed order, then you have to convert the agentset into a list. There are two primitives that help you do this, **sort** and **sort-by**.

```
set my-list-of-agents sort-by [size-of ?1 < size-of ?2] turtles
        ;; This sets my-list-of-agents to a list of turtles sorted in
        ;; ascending order by their turtle variable size.

foreach my-list-of-agents [
  ask ? [
    forward random 10
    do-stuff ]
  ]
]
```

If you use **foreach** like above, the agents in the list run the commands inside the **ask** sequentially, not concurrently. Each agent finishes the commands before the next agent begins them. See also: **without-interruption**.

Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of information in a list. Each value in the list can be any type of value: a number, a string, an agent, an agentset, or even another list.

Constant lists

You can make a list by simply putting the values you want in the list between brackets, like this: set mylist [2 4 6 8].

Building lists on the fly

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the **list** reporter. The **list** reporter accepts two other reporters, runs them, and reports the results as a list.

```
set random-list list (random 10) (random 20)
```

To make longer or shorter lists, you can use the list reporter with fewer or more than two inputs, but in order to do so, you must enclose the entire call in parentheses, e.g.:

```
(list random 10)
(list random 10 random 20 random 30)
```

See also: **n-values**, **values-from**, and **sentence**.

Changing list items

Technically, lists can't be modified, but you can construct new lists based on old lists. If you want the new list to replace the old list, use **set**. For example:

```
set mylist [2 7 5 Bob [3 0 -2]]      ;; mylist is now [2 7 5 Bob [3 0 -2]]
set mylist replace-item 2 mylist 10  ;; mylist is now [2 7 10 Bob [3 0 -2]]
```

See also: **lput**, **fput**, **but-last**, and **but-first**.

Iterating over lists

If you want to do some operation on each item in a list in turn, the **foreach** command and the **map** reporter may be helpful. **foreach** is used to run a command or commands on each item in a list. It takes an input list and a block of commands, like this:

```
foreach [2 4 6]
  [ crt ?
    show "created " + ? + " turtles" ]
=> created 2 turtles   => created 4 turtles   => created 6 turtles
```

map is similar to **foreach**, but it is a reporter. It takes an input list and another reporter.

```
show map [round ?] [1.2 2.2 2.7]              ;; prints [1 2 3]
```

map reports a list containing the results of applying the reporter to each item in the input list. Again, use ? to refer to the current item in the list. See also: **repeat** and **while**.

Skeleton of many NetLogo Models

```
globals [ … ]           ;; global variables (also defined with sliders, …)
turtles-own [ … ]       ;; user-defined turtle variables (also <breeds>-own)
patches-own [ … ]       ;; user-defined patch variables
…
to setup
        clear-all … setup-patches … setup-turtles … setup-graphs …
end
…
to go
        conduct-observer-procedure …
        ask turtles [conduct-turtle-procedure] …
        ask patches [conduct-patch-procedure] …
        update-graphs  ;; this may include update-view, update-plots…
end
…
to update-plots
        set-current-plot “myPlot” … set-current-plot-pen “myPen” …
        plot statistics …
end
…
to-report statistics
        … report theResultOfSomeFormula
end
```

Common Primitives

Turtle-related: die, forward (fd), myself, nobody, -of, other-turtles-here, patch-here, right (rt), self, setxy, turtle, turtles, turtles-at, turtles-from, turtles-own, value-from.

Patch-related: clear-patches (cp), distance, myself, neighbors, neighbors4, nobody, patch-at, patches, patches-from, patches-own, value-from.

Agentset primitives: any?, ask, count, histogram-from, is-agentset?, max-one-of, min-one-of, n-of, one-of, sort, sort-by, with, with-max, with-min, values-from.

Control flow and logic primitives: and, foreach, if, ifelse, ifelse-value, let, loop, map, not, or, repeat, report, stop, startup, wait, while, without-interruption, xor.

World primitives: clear-all (ca), clear-patches (cp), clear-turtles (ct), display, max-pxcor, min-pxcor, no-display, random-pxcor, world-width, world-height.