NETLOGO 4.0 – QUICK GUIDE

Luis R. Izquierdo (http://luis.izquierdo.name)

Agents
The NetLogo world is made up of agents. Agents are beings that can follow instructions. There are four 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.
- **Links**: Links are agents that connect two turtles. Links can be directed (from one turtle to another turtle) or undirected (one turtle with another turtle).
- **The observer**: The observer doesn't have a location - you can imagine it as looking out over the world of turtles, links 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 NetLogo Dictionary has a complete list of built-in instructions (i.e. a Primitives Dictionary).

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

In this program, setup is a procedure (since it is implemented by us), whereas clear-all and create-turtles are both primitives (they are built into NetLogo).

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", "most-popular-girl"). These names are preceded by the keyword to-report. The keyword end marks the end of the instructions in the procedure.

```lisp
to-report average-wealth
  report mean [wealth] of turtles ; average wealth in the population of turtles
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 verbs are preceded by the keyword to (instead of to-report). The keyword end marks the end of the instructions in the procedure.

```lisp
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.

```lisp
to-report absolute-value [ number ]
  ifelse number >= 0
    return number   ; number is already non-negative
  [ report number ] ; return number (a non-negative value)
  [ report 0 - number ] ; Otherwise, return the opposite, which is necessarily positive.
end
```
Variables

Variables are places to store values (such as numbers). A variable can be a global variable, a turtle variable, a patch variable, a link 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 ["number-of-trees"]
```

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

- **Built-in variables:** For example, all turtles and all links have a color variable, and all patches have a pcolor variable. If you set this variable, the corresponding turtle, link or patch changes color. Other built-in turtle variables are xcor, ycor, and heading. Other built-in patch variables include pxcor and pycor. Other built-in link variables are end1, end2, and thickness. You can find the complete list in the NetLogo Dictionary.

- **User-defined turtle, patch, and link variables:** You can also define new turtle, patch or link variables using the turtles-own, patches-own, and links-own keywords respectively, like this:

```
turtles-own [energy] ; each turtle has its own energy
patches-own [roughness] ; each patch has its own roughness
links-own [strength] ; each link has its own strength
```

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] ; turtle1 and turtle2 are inputs
  let temp [(color) of turtle1] ; store the color of turtle1 in temp
  ask turtle1 [set color [:color] of turtle2]
  set turtle1's color to turtle2's color
  ask turtle2 [set color [:color]] ; ; new set turtle2's color to turtle1 (original) color
```

Setting and reading the value of variables

Global variables can be read and set at any time by any agent. Every agent has direct access to her own variables, both for reading and setting. Sometimes you will want an agent to read or set a different agent's variable; to do that, you can use ask (which is explained in detail a bit later):

```
ask turtle 5 [show color] ; print turtle 5's color
ask turtle 5 [set color blue] ; turtle 5 becomes blue
```

You can also use of to make one agent read another agent's variable. of is written in between the variable name and the relevant agent (i.e. reporter of agent). Example:

```
show [color] of turtle 5 ; same as the first line in the code above
```

Finally, a turtle can read and set patch variables of the patch it is standing on directly, e.g.

```
ask turtles [set pcolor red]
```
The code above causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you cannot have a turtle variable and a patch variable with the same name – e.g. that is why we have color for turtles and pcolor for patches).

**Ask**

NetLogo uses the ask command to specify commands that are to be run by turtles, patches or links. 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:

```plaintext
to setup
  clear-all ; clear the world.
  create-turtles 100 ; create 100 new turtles with random heading
  ask turtles [ ; turn them red.
    set color red
    forward 50 ; make them move 50 steps forward.
  ]
  ask patches [ ; patches with pcolor green set to 0.
    [ if (pcolor > 0) set pcolor green ]
  ]
end
```

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

```plaintext
to setup
  clear-all ; clear the world.
  create-turtles 2 ; make 2 turtles named 0 and 1
  ask turtle 0 [ ; tell the first one to go forward 10 steps.
    set fd 10
  ]
  ask turtle 1 [ ; ask the second turtle to become green.
    set color green
  ]
  ask patch 2 -2 [ ; ask the patch at (2,-2) to become blue.
    set pcolor blue
  ]
  ask turtle 0 [ ; ask the first turtle (with who number 0)
    ; to become blue.
    set color blue
  ]
  ask link 0 2 [ ; ask the link between turtle 0 and 2...
    ; to become green.
    set color green
  ]
  ask turtle 0 [ ; ask the first turtle (with who number 0).
    ; to become red.
    set color red
  ]
  ask patch-at 1 0 [ ; to ask the patch 0, 0, 0.
    set pcolor red]
end
```

**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, e.g.:

```plaintext
set my-list [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.

```plaintext
set my-random-list list: (random 10) : (random 20)
```

To make shorter or longer 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.:

```plaintext
show (list random 10)
show (list random 10 random 20 random 30)
```
The primitive lists you construct a list from an agentset (i.e. a set of agents). It reports a list containing each agent's value for the given reporter (syntax: \texttt{[reporter] of agentset}).

\begin{verbatim}
set fitness-list ([fitness] of turtles);  
// list containing the fitness of each turtle (in random order)  
show [pxcor; pycor] of patches;
\end{verbatim}

See also: \texttt{n-values}, and \texttt{sentence}.

\textbf{Changing list items}

Technically, lists cannot be modified, but you can construct new lists based on old lists. If you want the new list to replace the old list, use \texttt{set}. For example:

\begin{verbatim}
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 10 5 Bob [3 0 -2]]
\end{verbatim}

See also: \texttt{lput}, \texttt{fput}, \texttt{but-last}, and \texttt{but-first}. (Note that \texttt{fput} is much faster than \texttt{lput})

\textbf{Iterating over lists}

To apply a function on each item in a list, you can use \texttt{foreach} or \texttt{map}. \texttt{foreach} is used to run a block of commands on each item in a list. It takes as inputs the list and a block of commands, e.g.:

\begin{verbatim}
foreach [2 4 6] [xrt ?]  
  show "created " ? + ! turtles  
=> created 3 turtles
\end{verbatim}

\texttt{map} is similar to \texttt{foreach}, but it is a reporter (it returns a list). It takes as inputs the input list and a reporter, and returns an output list containing the results of applying the reporter to each item in the input list. As when using \texttt{foreach}, the keyword \texttt{?} refers to the current item in the list.

\begin{verbatim}
show map [round 3] [1 2 2 2 2 7]  ; prints [1 2 3]
\end{verbatim}

Both \texttt{foreach} and \texttt{map} can take \textit{multiple} lists as input; in that case, the block of commands (in \texttt{foreach}), or the input reporter (in \texttt{map}), is run for each group of items from each list, i.e. it is run once for the first items, once for the second items, and so on. In the block of commands (if you are using \texttt{foreach}), or in the input reporter (if you are using \texttt{map}), write \texttt{?1} through \texttt{?n} to refer to the current item of each list. \texttt{?1} refers to an item from the first list, \texttt{?2} an item from the second list...

\begin{verbatim}
show (map [?1 + ?2] [1 2 3] [100 200 300])  ; prints [101 202 303]
\end{verbatim}

See also: \texttt{repeat} and \texttt{while}.

\textbf{Agentsets}

An agentset is a set of agents; all agents in an agentset must be of the same type (i.e. turtles, patches, or links). An agentset is not in any particular order. In fact, it's always in a random order\(^\dagger\). What's powerful about the agentset concept is that you can construct agentsets that contain only \textit{some} turtles, \textit{some} patches, or \textit{some} links. For example, all the red turtles, or the patches with \texttt{pxcor} evenly divisible by five, or all the links departing from a certain agent. These agentsets can then be used by \texttt{ask} or by various reporters that take agentsets as inputs.

Simple built-in agentsets are given by \texttt{turtles}-\texttt{here} (which contains only the turtles on my patch) and by \texttt{turtles}-\texttt{at} (only the turtles on some other particular patch). The \texttt{primitive with} \textit{is very useful to build agentsets}. Here are some examples of how to make agentsets:

\begin{verbatim}
turtles with [color = red]  ; all red turtles
other-turtles-here with [color = red]  ; all other red turtles on my patch
patches with [pxcor > 0]  ; patches on right side of view
turtles in-radius 3  ; all turtles three or fewer patches away
patches at-points [[1 0] [0 2] [-1 0] [0 -1]]  ; the four patches to the east, north, west, and south
neighbors4  ; shorthand for those four patches
[my-links] of turtle 0  ; my links to this turtle
\end{verbatim}

\(^\dagger\) If you want agents to do something in a fixed order, you can make a list of the agents instead.
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 `all?` to see if every agent in an agentset satisfies a condition.
- Use `count` to find out exactly how many agents are in the set.

Here are some more complex things you can do:

```turtle
ask one-of turtles | set color green |
```

Use `ask` to make a random agent from an agentset.

```turtle
ask (max-one-of turtles [wealth]) | donate |
```

Use `max-one-of agentset [reporter]` to report the agent in the agentset with the highest value for the given reporter.

```turtle
show mean [wealth] of turtles |
```

Use `mean` to make a list of values one for each agent in the agentset. Then use one of NetLogo's list primitives to do something with the list (i.e., calculate the mean).

```turtle
show (turtle-set turtle 0 turtle 2 turtle 3) |
```

Use `turtle-set`, `patch-set`, and `link-set` with reporters to make new agent sets by gathering together agents from a variety of sources.

```turtle
show turtles = patches |
```

Check whether two agent sets are equal using `=` or `\neq`.

```turtle
show member? turtle 0 turtles |
```

Use `member?` to see if an agent is a member of an agentset.

```turtle
if all? turtles [color = red] |
```

Use `all?` to see if every agent in the agentset satisfies a certain condition.

**Synchronization**

When you `ask` a set of agents to run more than one command, each agent must finish before the next agent starts. One agent runs all of the commands, then the next agent runs all of them, and so on. As mentioned before, the order in which agents are chosen to run the commands is random. To be clear, consider the following code:

```turtle
ask turtles |
  | forward random 10 |
  | wait 0.5 |
  | set color blue |
```

The first (randomly chosen) turtle will move forward some steps, she will then wait half a second, and she will finally set her color to blue. Then, and only then, another turtle will start doing the same, and so on until all turtles have run the commands inside ask without being interrupted by any other turtle. The order in which turtles are selected to run the commands is random. If you want all turtles to move, and then all wait, and then all become blue, you can write it this way:

```turtle
ask turtles |
  | forward random 10 |
  | wait 0.5 |
  | set color blue |
```

Finally, you can make agents execute a set of commands in a certain order by converting the agentset into a list. There are two primitives that help you do this, `sort` and `sort-by`.

```turtle
set my-list-of-agents sort-by ? [size] of ? turtles |
```

This sets `my-list-of-agents` to a list of turtles sorted by the number of turtles.

```turtle
foreach my-list-of-agents |
  ask ? | |
    forward random 10 |
    wait 0.5 |
    set color blue |
```

Each agent undertakes the list of commands without being interrupted. The next agent does not start until the previous one has finished.

See also: `ask-concurrent` and `without-interruption`. 5/6
Tick counter

In many NetLogo models, time passes in discrete steps, called "ticks". NetLogo includes a built-in tick counter so you can keep track of how many ticks have passed. The current value of the tick counter is shown above the view (the black window where you can see patches and turtles).

In code, to retrieve the current value of the tick counter, use the ticks reporter. The tick command advances the tick counter by 1. The clear-all command resets the tick counter to 0. If you want to reset the counter to 0 without clearing everything, use the reset-ticks command.

Skeleton of many NetLogo Models

```netlogo
globals [ ... ]  ; global variables (also defined with sliders)
turtles-own [ ... ]  ; user-defined turtle variables (also breeds-own)
patches-own [ ... ]  ; user-defined patch variables
patches-own [ ... ]  ; user-defined link variables (also breeds-own)

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 ];
    tick
    update-graphs  ; this may include update-view, update-plots ;
end

to update-plots
    set-current-plot "myPlot"
    ; set-current-plot-pen "myPen"
    plot statistics
    set-current-plot "myXYPlot"
    plotxy ticks (count turtles) ;
    set-current-plot "myHistogram"
    histogram [age] of turtles
end

to-report statistics
    report theResultOfSomeFormula
end

Common Primitives

Turtle-related: die, forward (fd), move-to, my-links, myself, nobody, of, other, other-end, patch-here, right (rt), self, setxy, turtle, turtle-set, turtles, turtles-at, turtles-here, turtles-on, turtles-own.

Patch-related: clear-patches (cp), distance, import-pcols, myself, neighbors, neighbors4, nobody, of, other, patch, patch-at, patch-set, patches, patches-own.

Link-related: both-ends, clear-links, create-link-from, create-link-to, create-link-with, in-link-neighbor?, in-link-neighbors, in-link-from, is-directed-link?, link, link-length, link-neighbors, link-set, link-with, my-in-links, my-links, other-end, out-link-neighbor?, out-link-neighbors, out-link-to, tie.

Agentset primitives: all?, any?, ask, ask-concurrent, count, in-radius, is-agentset?, max-one-of, min-one-of, n-of, neighbors, of, one-of, other, sort, sort-by, with, with-max, with-min.

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

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

6/6