NetLogo Programming
An Introduction

Alan G. Isaac

American University

Contents

Background

This introduction is written for Netlogo 5.
Before reading these notes, please read NetLogo Basics
You may also find the following to be useful preliminaries:

- Gabriel Wurzer's Introduction: `http://publik.tuwien.ac.at/files/PubDat_200771.pdf`
- The OpenABM introduction `http://www.openabm.org/book/introduction-agent-based-modeling`

Goals

After mastering this material, you will be able to:

- understand the basic structure of a NetLogo program
- find NetLogo “Code examples” to simplify your own programming tasks
- create command procedures and reporter procedures
- use NetLogo’s branching and looping constructs
- modify an existing NetLogo model
- build a new NetLogo model

Steps toward Making Your Own Models

1. Experiment with existing models via GUI
   - Models Library downloads with installation (File > Models Library)
     `http://ccl.northwestern.edu/netlogo/models/`
   - other publicly available models on the web
     `http://www.openabm.org/`
2. Modify existing models
3. Roll your own

Modifying Models

Find a model that does something close to what you want:

- make sure your intended use complies with the copyright
- save it under a new name
- add appropriate attribution to the header
- modify the model to suit your needs
- update the model documentation to match your changes
- make sure all borrowed code is clearly and appropriately attributed

Structured and Commented Code

- use help names for variables and procedures
- create a procedure for any repeated code blocks (DRY)
- turn your procedures into pure functions when possible
- structure and comment your code for readability

This will be:
– helpful to others who read your code – helpful to you in both writing and understanding your code

Simple Modifications: World Settings

You might want to change the World settings:

- size
- location of 0, 0
- torus or rectangle
- patch size (this and size determine size of world screen)

New Model: Key Decisions

- system to be modeled
- what do the agents represent?
- what are the rules of action and interaction?
- how will you approximate these rules in the modeling environment?
- if your model repeatedly runs a schedule, how much time is represented by one iteration (e.g., by 1 tick)?

Basic Model

A basic model will have:

- `setup` procedure
- `go` procedure

For example:

`setup`

- initialize global variables
- initialize any agents (e.g., the properties of patches and turtles)
- initialize output files

`go`

- run one interation of the model
- update output files

Basic GUI Elements: Graphs

A simple model may add:

graphs

- graphs can dynamically monitor a single value (`plot`), pairs of values (`plotxy`), or collections of values (`histogram`)
- easy to add in NetLogo GUI

Basic GUI Elements: Sliders, Choosers, and Switches

A simple model may also add:
sliders, choosers, and switches:

- sliders, choosers, and switches provide users with GUI control over model parameters (e.g., the number of agents in the model)

`startup` **procedure:**

- If you name a procedue `startup`, it will be run when your model first loads in the GUI. This is the right place to set default values for your sliders.

Note

For experiements using BehaviorSpace, `startup` run only once: `http://ccl.northwestern.edu/netlogo/docs/versions.html`)

Using Sliders and Choosers

Global variables can be declared in the `Interface` (instead of in the `Code` tab).
NetLogo allows using a "right click" on the `Interface` window to produce a context menu that can add GUI elements.
Interface globals (set in sliders, switches, and choosers) are a convenience feature with downsides as well as upsides.

Sliders and Choosers: Some Upsides

- Allow easy experimentation with a model.
- Designed to interact well with BehaviorSpace (as we will see later).

Sliders and Choosers: Some Downsides

- `Interface` variables are not declared in the `Code` tab, so while you are coding they are less visible. (However, if you code with a text editor, they are visible in the .nlogo file, as plain text.)
- An `Interface` variable does not include a default value, so if a user resets the slider and then saves your model, it saves a different value than you intended. (So it is a good idea to use NetLogo's special `startup` procedure to set default values, perhaps by calling a `reset-defaults` procedure.)

Basic Structure

The `Code` tab can contain

- comments
- declarations
- procedures (commands and reporters)

Comments

- a semicolon (`;`) begins a comment for remainder of line
- be sure to state the author and date in a comment at the beginning of each program

Declarations Section

The declarations section precedes the procedures section.

The two most common declarations are global variables and agent attributes.

Declaration of Global Variables

- `globals [...]` declares a list of global variables (may be thought of as `observer-owns`)

Note: other global variables declared in sliders. We will call those “interface globals”.

Global Variables

- each global variable must be declared
- a global variable can be declared in the declarations section **or** in the GUI (e.g., in a slider or chooser)
- every agent can access or `set` a global variable
- values are assigned (or reassigned) with the `set` command

Declaration of Instance Attributes

- `patches-own [...]`: list of patch attributes
- `turtles-own [...]`: list of turtle attributes
- `links-own [...]`: list of link attributes
- can also declare breed attributes

Procedures Section

The procedures section *only* contains procedures (user-written commands and reporters). It comes *after* the declarations section.

command procedure

- `to my-procedure ... end`
- body contains NetLogo commands

reporter procedure

- `to-report my-reporter ... end`
- returns a value
- must use the `report` command

Review: Procedures

Enter the following in the `Code` window:

```
globals [ nHeads ]

to setup
  clear-all
end

to go
  set nHeads (nHeads + fairCoinFlip)
end

to-report fairCoinFlip
  report ifelse-value (random-float 1 < 0.5) [1] [0]
end
```

Review: Procedures ...

Next, return to the Command Center and enter the following lines:

```
setup
show nHeads
go
show nHeads
```

Note: `clear-all` calls `clear-globals`, which sets all global variables to their default value. The default value of variables declared with `globals` is `0`.

Placeholders

When you are writing code, you may wish to refer to a procedure you have not written yet. If you do not define a procedure with this name, the NetLogo syntax checker will complain.
The solution is to define an empty procedure or a procedure that warns you that it needs to be written.
Such placeholders and warnings are sometimes called “scaffolding”. The idea is that assist you in construction of your model, but you intend to remove them from the final product.

Code Tab: Basic Structure

In the Code tab, outside of code blocks, we should only find:

- comments
- NetLogo keywords (see below)
- `<instances>-own` (see below)

Comment: many of the NetLogo documentation examples are slightly misleading on this score. E.g., `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#breed`

Keywords

http://ccl.northwestern.edu/netlogo/docs/dictionary.html#Keywords

- __includes http://ccl.northwestern.edu/netlogo/docs/dictionary.html#includes
- extensions http://ccl.northwestern.edu/netlogo/docs/extensions.html
- globals
- breed, directed-link-breed, undirected-link-breed
- patches-own, turtles-own, <breeds>-own, links-own, <link-breeds>-own
- to, to-report, end

Code Examples

The NetLogo Models Library includes a collection of code examples. Be sure to look at these for hints whenever you get stuck.

Basic Program Structure

Even the simplest NetLogo programs traditionally include the following structure:

globals declaration of global variables using the `globals` keyword

setup a procedure named `setup` that initializes the global varaibles and does other setup operations

go a procedure that runs one iteration of the model; this holds the "schedule" for your program

Example: Minimal Program Structure

```
globals [ gvar01 gvar02 ]

to setup
  clear-all
end

to go
  do-stuff
end

to do-stuff
  ...
end
```

Program Structure: Setup

As soon as you add any complexity to your model, you will want to break the model set up into parts:

- the global variables,
- the patches
- the turtles

So your model set up procedure will often look like:

```
to setup
  ca
  setupGlobals
  setupPatches
  setupTurtles
  reset-ticks
end
```

Note: NetLogo already has a `setup-plots` command, which in turn is called by `reset-ticks`. If you want to set up your plots in the `Code` tab, use the name `setupPlots` or `init-plots` instead.

Application: Minimal Program Structure

```
globals [ nHeads ]

to setup
  clear-all  ;sets nHeads to 0
end

to go
  repeat 50 [
    set nHeads (nHeads + fairCoinFlip01)
  ]
end

to-report fairCoinFlip01
  ;; fill in procedure body
end
```

Parameters for Commands and Reporters

Procedures can specifiy formal parmeters in brackets after the name. Here is a silly reporter that illustrates the use of parameters.

```
to-report is-equal? [#x #y]
  report (#x = #y)
end
```

Note that parameter names are strictly local to the procedure, which means you cannot refer to them outside the procedure body.

Hash Convention for Formal Parameters

In these notes, we will adopt the helpful convention that parameter names begin with a hash mark (#). This is just a convention; it is not required by NetLogo.

Write Once Use Anywhere

After you copy `is-equal?` into your `Code` tab, you can use it elsewhere in your code. You can even use it in the Command Center. E.g., go to the Command Center and type in:

```
show is-equal? 2 3
```

The observer will show you the value `false`. Note how the reporter "consumes" two arguments (the 2 and the 3), because we defined it to do so. Note that you do *not* put the arguments in brackets, even though you must use brackets in the definition.
See: `http://ccl.northwestern.edu/netlogo/2.0/docs/programming.html#procedures2`

Parameters for Procedures (another example)

Suppose we want to simulate a coin flip with a specified probability.

```
to-report coinFlip01 [#p]
  report ifelse-value (random-float 1 < #p) [1] [0]
end
```

Once you copy that to your `Code` tab, you can use it like this:

```
show coinFlip01 0.3
```

Note

By convention, the names of formal parameters begin with a hashmark (#).

Procedures Can Call Procedures

You can define new procedures in terms of existing procedures.

```
to-report fairCoinFlip01
  report coinFlip 0.5
end
```

Style Guide

Currently there is no official NetLogo style guide. The NetLogo Models Library is stylistically fairly consistent, so it can serve as a guide by example.
A course-related guide is available at
`http://ccl.northwestern.edu/courses/mam2005/styleguide.htm`
Contrary to that style guide, I recommend:

- do not use spaces in your file names
- use camel-case instead of hyphenated long names, beginning with a lower-case letter

While hyphenated names are a convention in Lisp derived languages, they are not possible in many other languages.

Note

Remember that NetLogo is case insensitive, so case conventions are purely for reader convenience.

Common Styles: Naming

Here are some style guidelines that reflect some fairly common practices.

- do not use underscores in names
- name boolean variables with a question mark: `attempted-task?`
- name command procedures with nouns and reporter procedures with verbs
- start parameter names with a hash and local variable names with an underscore:

```
to-report sq [#x]
  let _xsq (#x * #x)
  report _xsq
end
```

- breed names should be plural

Common Styles ...

- indent code blocks by 2 spaces per level, including procedure and reporter bodies;
- do not use tab characters (except possibly in output)
- declare variables (`globals`, `patches-own`, etc.) one per line, with an explanatory comment for each variable
- identify procedure context with a comment:

```
to move ;; turtle procedure
  right random-float 360
  forward 1
end
```

Common Styles ...

- avoid using `who` numbers
- put branching conditions in parentheses
- open code-block brackets at the end of a line; close them on their own line, except between the if and else clauses, e.g.,

```
ifelse (this?) [
  do-A
][
  do-B
]
```

Looping: repeat

The `repeat` primitive allows you to repeat a command block as many times as you wish. E.g., enter the following at the Command Center.

```
let %ct 0 repeat 50 [show %ct set %ct (%ct + 1)]
```

In the Command Center, the observer shows you the whole numbers up 0-49. As another example, at the Command Center enter:

```
clear-all
repeat 50 [set nHeads (nHeads + fairCoinFlip) ]
show nHeads
```

Note: `clear-all` sets all global variables to their default value of `0`.

Exiting a Loop: stop

At the Command Center enter:

```
let %ct 0 repeat 50 [show %ct set %ct (%ct + 1) stop]
```

The `stop` command exits the loop, so in the Command Center, the observer only shows `0`.

Exiting a Procedure: stop

We can use `stop` to exit a procedure, but `stop` only exits the procedure that executes it. To illustrate, add the following to the `Code` tab:

```
to test
  show 0  stop-me  show 2
end

to stop-me
  stop  show 1
end
```

Go to the Command Center and enter `test`. You will see `0` and `2` printed.

Looping: loop

Run a list of commands repeatedly (potentially forever):

```
loop [ commands ]
```

This is obviously a dangerous construct, but if one of the commands eventually calls `stop`, you will exit the loop.

```
loop [if (ticks > 100) [stop] tick]
```

Use of `loop` is not quite like use of a forever button. In NetLogo, we usually use a forever button in order to repeat something forever. We can click again on a forever button to exit the loop. If the button calls a procedure that executes the stop command, that will also exit the forever-button loop. However, procedures do not pass on the stop command to `loop`: to break out of `loop`, `stop` must be called by a command directly in the loop body.

Stopping Forever Buttons

NetLogo models often have a `go` command that is called by a forever button. If you want to stop on a condition, rather than by again clicking the button, use `stop` conditionally at the **top** of your procedure:

```
to go
  if condition? [ stop ]
  ...
end
```

This prevents the user from forcing additional step in the model by repeatedly pressing the button.

Code Analysis

- scaffolding (`print` statements)
- inline tests (`if` tests with `error` statements)
- test procedures (e.g., `test-setup` and `test-go`)
- procedure timing (e.g., `reset-timer myproc print timer`)
- profiling (see the profiler“ extension) `http://ccl.northwestern.edu/netlogo/docs/profiler.html`

NetLogo Source (.nls) Files

In support of DRY programming, NetLogo allows a model to load a `.nls` file declaring variables, breeds, and (most importantly) procedure definitions.
`http://ccl.northwestern.edu/netlogo/5.0/docs/interface.html#includes`
Unfortunately, as of NetLogo 5.1, the process for creating a new `.nls` file in the Code tab is rather awkward. For a description, see
`http://netlogo-users.18673.x6.nabble.com/Using-includes-td4869749.html`

Types of Plots

NetLogo builds in the following plot types:

plot you provide the `y` value; the `x` values are automatically incremented.

plotxy you provide the `x` and `y` values

histogram you provide a collection of values as a list

Exercise: Simple plot

In the `Interface` tab, add a plot with the pen update command `plot nHeads`.
In the `Code` tab, create a coin-flipping program that has the following `go` procedure:

```
to go
  set nHeads 0
  repeat 50 [set nHeads (nHeads + flipCoin)]
  update-plots
end
```

Clearly this is not the complete program: you need to declare `nHeads` as a global variable, define a `flipCoin` reporter procedure, and define an appropriate `setup` procedure.
In the Command Center, run your `setup` procedure, and then run your `go` procedure 100 times.

Basic Concepts: Plots

Review plots in NetLogo Basics.
For the moment, we will only change the `pen update commands`.

pen update commands commands to be executed when the plot updates

`setup-plots` NetLogo primitive to initialize all plots. Often comes at the end of our `setup` procedure. (However, it is more common to use `reset-ticks`, which calls `setup-plots`.)
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#reset-ticks`

`update-plots` NetLogo primitive to update all plots. Often comes at the enf of our `go` procedure. (However, it is more common to use `tick`, which calls `update-plots`.)
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#tick`

In the NetLogo Models Library, under Code Examples, see `Plotting Example`.

Temporary Plot Pens

If you want to add background features (like a 45 degree line) to a plot, you can use a temporary plot pen during setup.

```
;;plot 45 degree line from (0,0) to (1,1)
create-temporary-plot-pen "equal"
plotxy 0 0 plotxy 1 1
```

export-plot

The `export-plot` command writes a comma-separated values file. There data written includes the x and y values of all points plotted by all the plot pens in the plot.
Specify the plotname as a string: it is the same as whatever you entered as the name in the plot dialogue (which is used as the title of your plot).
The data is written to an external file. You specify the filename as a string. Use *forward* slashes.
See: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#export-plot`

Plot Commands

most-used plot commands: histogram plot plotxy set-current-plot set-current-plot-pen set-plot-pen-mode

often-used plot commands: set-histogram-num-bars set-plot-pen-color set-plot-x-range set-plot-y-range

autoplot (automatic axes range adjustemnts): autoplot? auto-plot-off auto-plot-on

clear-plot related commands: clear-all-plots clear-plot plot-pen-reset

other plot commands: `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#plottinggroup`

Simplest Histogram

A histogram plots the frequency of occurence of items in a list. Add a new plot in the `Interface` tab and replace the default pen update commands with `histogram [1 2 2 3 3 3]`. Click `OK` then then in the Command Center enter `ca update-plots`.
Note that by default `histogram` produces a line plot. For the corresponding bar chart, you need to change the pen-mode. At the Command Center, you can enter `set-plot-pen-mode 1`. But you can set the pen mode in NetLogo's plot dialogue.

Dynamic Histogram

If we have a histogram of turtle colors, we would like our histogram to be redrawn when our turtles change colors.
In the NetLogo Models Library, under Code Examples, see `Histogram Example`.
Note that the `tick` command calls `update-plots`. Note that the x-axis is not autoscaled; you must scale in appropriately be histogramming your data.

Simple Histogram

Suppose we have turtles classified by color: red, green, or blue. After using the GUI to create a plot titled "Class Histogram", we can:

```
to update-class-histogram
  set-current-plot "Class Histogram"
  histogram map
    [position ? [red green blue]]
    ([color] of turtles)
end
```

Custom Histogram

If we would like to color-code our bars, we cannot use `histogram`. Instead we plot a bar for each value.

```
to update-class-histogram
  set-current-plot "Class Histogram"
  plot-pen-reset
  set plot-pen-mode 1      ;; bar mode
  set-plot-pen-color red
  plot count turtles with [color = red]
  set-plot-pen-color green
  plot count turtles with [color = green]
  set-plot-pen-color blue
  plot count turtles with [color = blue]
end
```

Parameter Sweep

parameter sweep systematic variation of scenarios

standard method for exploring the parameter space

BehaviorSpace a NetLogo tool for easily implementing a parameter sweep

Using BehaviorSpace

- Tools > Behavior Space > New
- give your experiment a name
- specify multiple values of parameters for your experiment
  - list the values explicitly, or use a range `[start increment end]`
  - `["myparam" 0 1 2 3]`, or `["myparam" [0 1 3]]`
  - you can vary the world size! (not a parameter in an ordinary sense)
- fill in the experiment information
- click “ok” when you are done
- save your NetLogo model (this will save your experiment settings)
- run the experiment by pressing the `Run` button
  - you will be prompted for how to save your data before the experiment runs; usually you should “table” output.
  - be sure to include the .csv extension if you enter a filename; NetLogo does not add it for you.
  - it runs much more quickly if you uncheck “update graphics” and “update plots and monitors”.

Behavior Space ...

Repetitions the number of replicates for each scenario (i.e., for each parameter combination)

reporters One reporter per line, for each value you want to record.

Setup commands: usually just your `setup` procedure

Go commands: the command(s) to run **one step** of your model; usually just `go` or `step`, but sometimes e.g. `repeat 10 [go]`. (Your recording reporters are called after each step.)

Stop condition: a reporter that reports `true` when the run should stop (or just set the number of steps as a `Time Limit`)

BehaviorSpace Cautions

- BehaviorSpace sets the values of globals variables *before* executing its `setup` and `go` commands. So if your `setup` commands set the value of these variables, the BS values will no longer apply.
  Example: if you use `clear-all` in your `setup`, you will reset to `0` all the non-interface globals that BehaviorSpace has set. (Note: `clear-all` does not reset your sliders or choosers, so BehaviorSpace works well with these.)
- If you set the value of `random-seed` in BehaviorSpace, it is reset to that value before each run (not once per experiment). Workaround: set the value during `setup` as a function of `behaviorspace-run-number`. As a simplest example: `random-seed behaviorspace-run-number`.

BehaviorSpace Output Formats

spreadsheet format written at the end of the experiment (data held in memory)
tries to be more "human readable"

table format written as the data is generated (data not held in memory)
use this for experiments generating large datasets

Review of Language Basics

- reassignment: set a b
- use parentheses to control order of operations
- use brackets [ ] for code blocks
- white space ignored after initial space
- procedures (commands and reporters; see above)

Basic Data Types

numbers *all* numbers are floating point (as in Javascript)

lists ordered, immutable collection of objects; concatenate with `sentence`

strings immutable sequence of characters; create with double quotes; concatenate with `word`

booleans true or false; reported by comparisons

Agentsets

- turtlesets
- patchsets
- linksets

Extension Data Types

- tables
- arrays

Language Surprises

- use `(- numbername)`, not `-numbername`
- case-insensitive
- necessary white space: `set a (3 * b)`

Recent Changes

- see the Transition Guide: `http://ccl.northwestern.edu/netlogo/docs/transition.html`
- `reset-ticks`: as of version 5, you must explicitly call `reset-ticks` to initialize the ticks counter; it is no longer called by `clear-all`
- NetLogo 5: can no longer concatenate strings with `+`; use `word`
- `random-one-of` was renamed `one-of`

Language Conventions

- Logical variables end in `?`
- procedure body indented
- two semicolons to start comment `;;`

Ticks

NetLogo includes a built-in tick counter:

```
print ticks   ;; display current value of ticks
tick          ;; increment ticks (by 1)
print ticks   ;; display current value of ticks
reset-ticks   ;; reset ticks to 0
print ticks   ;; display current value of ticks
```

Booleans and Comparisons: Numerical Issues

Be careful with numerical comparisons when you are not working with integers. Computers must work with approximations of fractions. The value of `(0.1 + 0.2)` is `0.30000000000000004`, so the value of `(0.1 + 0.2 = 0.3)` is `false`, and the value of `(0.1 + 0.2 > 0.3)` is `true`.

Control Flow: Branching

`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#controlgroup`

Booleans can be used for conditional code execution.

Suppose `condition` is a boolean variable (i.e., has a value of either `true` or `false`). We can use such a variable to determine the flow of control in a NetLogo program.

- if (condition) [*commands*]
- ifelse (condition) [*commands4true*] [*commands4false*]
- ifelse-value (condition) [*reporter4true*] [*reporter4false*]

Booleans and Conditional Branching

E.g., noting that `random-float 1` is between zero and one:

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

We might also like the observer to print “tails” for larger outcomes. We can use the `ifelse` construct to do this.:

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

Note that to create a string, we bracket a sequence of characters with double quotes.

Example: Conditional Setting of Global Variables

- start NetLogo
- In the `Code` window enter `globals [nHeads nTails]`
- Go to the Command Center and enter the following code:

```
ifelse (random-float 1 < 0.5)
  [set nHeads (nHeads + 1)]
  [set nTails (nTails + 1)]
show nHeads
show nTails
```

ifelse-value

NetLogo also provides the unusual `ifelse-value` primitive, which allows condition determination of a value.

```
ask turtles [
  set color ifelse-value (wealth < 0) [red] [blue]
]
```

http://ccl.northwestern.edu/netlogo/docs/dictionary.html#ifelse

Control Flow: Looping

- ask *agentset* [*commands*]
- foreach *list* [*commands*]
- repeat
- stop
- while
- loop (forever!)

Control Flow: Other

- ask-concurrent
- carefully (and `error-message`)
- every
- run runresult
- to to-report
- wait
- with-local-randomness
- without-interruption

Operators: Math, Logic and Comparison

math

- +, -, /, ^
- white space delimited (e.g., `3 + 2` not `3+2`)
- all are *binary*, but can write `(- x)` for `0 - x`

logical operators (operate on booleans)

- and, not, or, xor

comparison

- >, >=, <, <=, =, !=

Global Variables

- have global scope (i.e., are available anywhere in the program)
- must be declared before used
  - in the declarations section, or
  - by adding a button
- use `set a b` to change the value of variable `a`

Local Variables

Local variable can be created with `let` inside a procedure body. They are invisible outside their code block.[1]

- `let a b` declares a new local variable `a` and assigns it the value of `b`
- `set a b` *changes* the value of `a` to the value of `b`
- scope restricted to code block in which declared

A procedures formal parameters are also local to the procedure.

[1] However, see the discussion below in Tasks Are Closures.

NetLogo Lists: Basic Concepts

NetLogo lists can contain a variety of items in a fixed order.
Lists are:

- ordered
- immutable
- potentially heterogeneous (e.g., numbers and strings)

Examples:

```
let lst00 []                    ;; empty list
let lst01 [0 1]                 ;; list of numbers
let lst02 ["zero" "one"]        ;; list of strings
let lst03 [0 1 "zero" "one"] ;; list of numbers and strings
```

Zero-Based Indexing

Because lists are ordered, it makes sense to ask what item is at a particular location. We do this with the `item` primitive.

Indexing is zero-based: the first index is `0`.

`item` *index list* return item *index* of *list* (e.g., `item 0 [3 2 1]` is `3`)

Constructing Lists

The `list` primitive provides a general list constructor (with parentheses):

```
(list )                    ;;emtpy list
(list 0 "one" myvar)       ;;list of three items
```

A shorter bracket notation can be used with literals (but not variables):

```
[]           ;; emtpy list
[0 "one"]    ;; list of two items
```

List Length

The length of a list is the number of items in the list.

`length` *lst* reports the length of *lst*

`empty?` *lst* reports `true` if *lst* is empty

Adding Items to a List

`fput` *item list* prepends *item* to *list* (e.g., `fput 1 [2 3]`)

`fput` *item list* appends *item* to *list* (e.g., `lput 3 [1 2]`)

`sentence` *list1 list2* concatenates *list1* and *list2* (e.g., `(sentence [1 2] [3])`)

Note: in each case, a new list is returned: `[1 2 3]`.

List Members

`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#listsgroup`

`member?` *value list* report `true` if *value* is in *list*

`first` *list* report first *list* item

`last` report last *list* item

`item` *index list* report item at (zero-based) *index*

`one-of` *list* report a random item of *list*

`position` *item list* report the index of *item* in *list*

Creating Lists with n-values

Newcomers to NetLogo often find it puzzling to use `n-values`. Read the documentation carefully. `http://ccl.northwestern.edu/netlogo/docs/dictionary.html#n-values`

Consider the command `n-values 5 [?]`. This reports `[0 1 2 3 4]`: a list of 5 successive values, starting at 0.

Note that `n-values` takes two arguments: an integer size (here `5`), and a "reporter task" (here `[?]`).

Question Mark

The question mark in the reporter task is a special NetLogo variable that cannot be set directly by the user. The values taken by this special variable are determined by the command. When we use the `n-values` command, the question mark will take on successive integer values, starting at 0. (The number of values is determined by the size argument.)

These successive values that NetLogo assigns to `?` determine the successive values of the reporter task, which constitute the reported list.

Example: to produce a list of the squares of 0 through 9 we can use `n-values 10 [? * ?]`.

You are not required to use the question mark if you do not need it.

Example: `n-values 5 [random 2]`

Iterating over a List with foreach

- run commands for each item of a list
- syntax: `foreach *list* [*commands*]`
- simple example:

```
let range n-values 10 [?]
foreach range [show ? * ?]
```

or equivalently:

```
foreach n-values 10 [?] [show ? * ?]
```

http://ccl.northwestern.edu/netlogo/docs/dictionary.html#foreach

Question Mark

Note: `?` is special name that is sequentially assigned to each item in the `foreach` sequence. It can also be written as `?1`.
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#ques`

Cumulative Sum using foreach

```
to-report partial-sums [#nums]
  let total 0
  let result []
  foreach #nums [
    set total total + ?
    set result lput total result
  ]
  report result
end
```

Simple Function Plot

To plot a function `f`, we need to decide on a domain over which to plot it, and how many points to plot. For example:

```
let domain n-values 101 [? / 100]
foreach domain [plotxy ? f ?]
```

Exercise: Enter the following in your `Code` tab and plot it.

```
to-report f [#x]
  report 3.75 * #x * (1 - #x)
end
```

foreach Example: Multiple Lists

The `foreach` command can be used with multiple lists of identical length. The first result is computed from the first elements of the arguments. The second result is computed from the second elements of the arguments. For example:

```
(foreach [1 2] [3 4] [5 6] [print ?1 + ?2 + ?3])
```

Note the required parentheses.
Note: `?1`, `?2`, and `?3` are special names that are sequentially assigned to each item in the first, second, and third `foreach` sequence.

Operating on Lists

http://ccl.northwestern.edu/netlogo/docs/dictionary.html#listsgroup

sublists

- sublist, remove-duplicates
- remove *item list*, remove-item *int list*
- but-first, but-last
- n-of *int list*

new lists

- replace-item *int list*
- fput, lput, sentence
- n-values *int* [*reporter*]

rearranged lists

- reverse, shuffle
- sort, sort-by

Immutability

- NetLogo lists are immutable: you construct new lists based on old lists.
- if you want an extant variable to refer to a new list, use `set`.

```
set mylist replace-item 0 mylist 99
; mylist's first element is now 99
set mylist lput 100 mylist
; appends the value 100 to mylst
set mylist fput -1 mylist
; mylist now has a new first element
```

Creating Lists from Agentsets with of

use `of` with an agentset:

```
[color] of turtles

[pcolor] of patches

[(list self pcolor)] of patches

[(list self color size)] of turtles
```

Note that lists can contain lists!

Lists to Agentsets

`patch-set` *lst*

creats a patch set from any patches in *lst* (or its sublists)

`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#patch-set`

`turtle-set` *lst*

creats a turtle set from any turtles in *lst* (or its sublists)

`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#turtle-set`

Introduction to Functional Programming

`map` *reporter-task list* Apply a reporter to a list and produce a new list.
The current item is represented by `?`.
E.g., `map [? * ?] [0 1 2]` reports `[0 1 4]`.

`filter` *boolean-reporter-task list* Report a list of criterion-satsifying members of the input list. (See discussion below.)

`reduce` *two-input-reporter-task list* Combine the items of a list into a single result by repeatedly applying a binary operation.
Suppose `op` is an infix binary operation then `reduce op [a b c]` will report `((a op b) op c)`
E.g., `reduce [?1 + ?2] [1 2 3]` reports `6` while `reduce [?1 - ?2] [1 2 3]` reports `-4`.

`sort-by` *reporter-task list* Sort a list based on pairwise comparisons.
E.g., `sort-by [?1 > ?2] [3 1 4 2]` reports `[4 3 2 1]`

Mapping

```
to-report string-to-digits [#s]
  let _idxs n-values (length #s) [?]
  report map [read-from-string item ? #s] _idxs
end
```

Filtering

Sometimes we want a sublist of elements that meet a certain criterion. We can use `filter` for this. E.g.,

```
filter [? < 3] [1 2 1 3]
```

reports `[1 2 1]`.

Filtering Agentsets

If you want to filter an agentset `s` based on an attribute `w`, you would have to convert it to a list (`[self] of myagentset`) before your could apply `filter`. E.g.,

```
filter [[w] of ? < 3] [self] of patches
```

However, the better way is usually to use `with`:

```
patches with [w < 3]
```

(Of course, `with` reports an agentset, not a list.) It follows that `with` can remove an agent `a` from an agentset:

```
set myset myset with [self != a]
```

Lists: Contrast with Agentsets

agentset

- an *unordered*, mutable, homogeneous collection of agents
- traverse with `ask`
  - `ask agentset [ list of commands ]`

list

- an *ordered*, immutable, possibly heterogeneous collection of objects
- traverse the list items sequentially with `foreach`
  E.g., `foreach [1.1 2.2 2.6] [ print round ? ]`

Advanced List Use

Some example of advance list use:

nested foreach

```
to-report moore-offsets [radius]
  let dxdy (list )  ;; empty list
  let offsets n-values
    (2 * radius + 1) [? - radius]
  foreach offsets [
    let xoff ?
    foreach offsets [
      let yoff ?
      set dxdy lput (list xoff yoff) dxdy
      ]
  ]
  report dxdy
end
```

Permutations via Nested foreach

NetLogo supports recursion, which we use here to produce all the permutations of a list.

```
to-report permutations [#lst] ;Return all permutations of `
  let n length #lst
  if (n = 0) [report #lst]
  if (n = 1) [report (list #lst)]
  if (n = 2) [report (list #lst reverse #lst)]
  let result []
  let idxs n-values n [?]
  ;use each item as a first item, permuting remaining items
  foreach idxs [
    let xi item ? #lst
    foreach (permutations remove-item ? #lst) [
      set result lput (fput xi ?) result
    ]
  ]
  report result
end
```

Simple Reduction of a List

`reduce` *reporter list* repeatedly apply a binary operation to a list from left to right, using the binary operation supplied by *reporter*.

Use `?1` and `?2` in your reporter to refer to the two objects being combined. For example, we can sum the items in `[1 2 3]` as follows:

```
reduce [?1 + ?2] [1 2 3]
reduce + [1 2 3]  ;; short form
```

How reduce Works

Consider the following:

```
reduce [?1 + ?2] [1 2 4]
```

Remember, reduce works through the list from left to right. Here `?1` refers to the first argument, and `?2` refers to the second argument.

- Step 1: set `?1` to the first item (e.g., `1`) and set `?2` to the second item (e.g., `2`)
- Step 2: add `?1` and `?2`; if there are any more list items go to Step 3, otherwise report the result of the addition.
- Step 3: set `?1` to the result of addition (e.g., `3`), and then set `?2` to the next item in the list (e.g., `4`). Go to Step 2.

So the following would produce the same result:

```
to-report sum-list [lst]
  let arg1 first lst
  foreach butfirst lst [
    let arg2 ?
    set arg1 (arg1 + arg2)
```

Factorial via reduce

We can use `reduce` with `n-values` to produce the factorial of any positive integer.

```
to-report factorial [#n]
  report reduce * n-values #n [? + 1]
end
```

Binary to Integer via reduce

Suppose we have a list of zeros and ones representing a binary number.

```
to-report binary-to-integer [bits]
  report reduce [?1 * 2 + ?2] bits
end
```

All and Any

NetLogo does not provide *all* and *any* for lists. But they are easily implemented for list of booleans with *reduce*. For example:

```
to-report all-true [#lst]
  report reduce and #lst
end

to-report any-true [#lst]
  report reduce or #lst
end
```

flatten via reduce

Recall we can use `sentence` to concatenate lists. We can therefore use `reduce` with `sentence` to concatenate all the sublists in a list of lists.

```
to-report flatten [#lstlst]
report reduce sentence #lstlst
```

Example: `flatten [[0] [1 1] [2 2 2]]`

Reversing a List with reduce

To reverse a list using reduce, we use a useful trick: we modify our input argument by inserting an empty list at the front (with `fput`). This is the first value seen by `reduce`, so we can use it to successively accumulate items.

```
reduce [ fput ?2 ?1 ] (fput [] lst)
```

Note: since NetLogo has a `reverse` primitive, this exercise is simply to illustrate the capabilities of `reduce`.

Cumulative Sum using reduce

To form a list of cumulative sums, also called partial sums, we reuse the trick of inserting a list as the first item in our input. Construct new lists using reduce by first using `fput` to insert the initialized list.

```
to-report partial-sums [#lst]
  set #lst (fput [0] #lst)  ;;prepare for reduce
  report butfirst reduce [lput (?2 + last ?1) ?1] #lst
end
```

Challenge: carefully explain how this code works.

Computations for a Lorenz Curve

Use one of our `partial-sums` procedures (above).

```
to-report cumulative-shares [#wealths]
  ;; caclulate normalized sorted wealths
  let _n length #wealths
  let _ws sort #wealths
  let _cumsum-ws partial-sums _ws
  let _total-w last _cumsum-ws
  report map [? / _total-w] _cumsum-ws
end
```

Computing Gini Coefficient

Use the shares you computed for the Lorenz curve.

```
to-report shares2gini [#shares]
  let _n length #shares
  let _pop-shares n-values _n [(? + 1) / _n]
  let _gaps (map [?1 - ?2] _pop-shares #shares)
  report sum _gaps * 2 / _n
end
```

Plotting Lorenz Curve ...

```
plot 0
let _shares cumulative-shares [wealth] of turtles
let _n length _shares
set-plot-pen-interval 1 / _n
foreach _shares [plot ?]
```

Tasks vs. Procedures

Like procedures, tasks store code for later execution.
Unlike procedures, tasks are values, and they can be passed around like any other values.
We declare a task with the `task` primitive.
Analogously to procedures, we can have command tasks and reporter tasks.
We can ask which type of task we are dealing with with the `is-command-task?` and `is-reporter-task?` primitives.
Analogously to procedures, tasks can accept arguments. Taks arguments are represented as `?` or `?1`, `?2`, etc.

Command Tasks: Simple Example

A command task is used to run code without returning a value.
We use the `run` primitive to run a command task. Consider the following.

```
globals [x x++]

to setup
  set x 0
  set x++ task [set x (x + 1)]
end
```

Now we can `run x++` to add `1` to `x`.
NetLogo peculiarity: note that `x++` is just an ordinary NetLogo name.

Command Tasks

A command task is used to run code without returning a value.
We use the `run` primitive to run a command task. Consider the following.

```
globals [stack push]

to setup
  set stack []
  set push task [set stack lput ? stack]
end
```

Now we can `(run push 1)` to push a `1` on our stack. Note the required parentheses.

Surprising Need for Parentheses

IMPORTANT: If tasks take input arguments, they must be run with parentheses (as of NetLogo 5.1).
In the previous example `(run push 1)` works but `run push 1` fails. The parenthesis determine what is considered to be an input to the task. (Extra inputs are simply ignored.)

An Even More Surprising Need for Parentheses

```
let biggest task [ifelse-value (?1 >= ?2) [?1] [?2]]
show reduce [(runresult biggest ?1 ?2)] [1 2 3 4 3 2 1]
show reduce biggest [1 2 3 4 3 2 1]  ;shorthand for the sam
```

Reporter Tasks

A reporter task is used to run code and return a value.
We use the `runresult` primitive to run a reporter task. For example:

```
let square task [? * ?] print (runresult square 5)
```

Again, tasks with arguments must be run with parentheses (as of NetLogo 5.1). The following fails for lack of parentheses.

```
let square task [? * ?] print runresult square 5
```

Plotting Exercise: Simple Function Plot

Add a plot named `functionPlot` to your `Interface`.
Add the following command procedure to your `Code`.

```
to plotFunction [#fn #xl #xr #npts]
  let %x n-values #npts [#xl + ? * (#xr - #xl) / (#npts - 1
  foreach %x [plotxy ? (run-result #fn ?)]
end
```

Here `#fn` is a task that accepts one argument. The arguments `#xl` and `#xr` are the left and right boundaries of the plot domain. The argument `#npts` is the number of points to plot.
Exercise: On the interval `[0,1]`, plot the logistic map with an amplitude parameter of `3.5`.
Background: `http://en.wikipedia.org/wiki/Logistic_map`

Tasks Are Closures

Tasks reported by procedures close over variables local to the procedure. Consider the following reporter procedure, which reports a task.

```
to-report remainder-task [#divisor]
  report task [? mod #divisor]
end
```

Now we can do the following:

```
let mod3 remainder-task 3 show (runresult mod3 17)
let mod4 remainder-task 4 show (runresult mod4 17)
```

Tasks in the Models Library

- State Machine Example
- Termites 3D

Open a File

We use `file-open` to open a file for reading or appending.
Unlike many languages, NetLogo does not ask you to specify upon opening whether you will read from or write to the file. That is determined by the next file primitive you use (e.g., `file-read` or `file-write`).
Once you are done with an open file, you should close it with `file-close`.

```
file-open "temp.txt"
file-close  ;;close the last opened file
```

Note: use forward slashes, not backslashes, in your path names.
Note: NetLogo does not offer access to explicit file handles.

Open a File for Writing

If you open an existing file and write to it, you will *append* to that file.
If you want to *replace* the content of an existing file, you will have start with a `file-delete`.
In order to open a file for writing and close it afterwards, you need the following commands:

`file-delete` *string* delete the file designated by *string*

`file-open` *string* open a file for reading or appending (but not both)

`file-close:` close an open file

Using carefully with file-delete

Deleting a file that does not exist is a runtime error!
If you want to replace the content of a file, say `temp.txt`, you should begin by deleting the existing file. But suppose you do not know ahead of time whether the file exists. Then use `carefully`.
`http://ccl.northwestern.edu/netlogo/docs/dictionary.html#carefully`

```
carefully [file-delete "temp.txt"] []
file-open "temp.txt"
file-print "write this line to temp.txt"
file-close
```

Caution: you should ask before file-delete

As long as you are **sure** it is safe to delete any existing file by that name, you can use `carefully` with `file-delete`.
You can **never** be sure it is ok to delete a file that someone else might have created. So models that you share should not use this approach.
You can check whether the file exists with `file-exists`, which returns a boolean.

```
if (file-exists? "temp.txt") [
  ifelse (user-yes-or-no? "OK to delete temp.txt?") [
    file-delete "temp.txt"
  ][
    error "temp.txt already exists"
  ]
]
```

File Output Commands

NetLogo provides an unusual collection of commands for writing to files. Note that `file-print` and `file-show` append a carriage return (CR).

`file-type` *value* write *value*
: strings are written without quotes; backslashes escape control characters

`file-write` *value* write a space and then write *value*,
: strings are written quote delimited; backslashes are literal

`file-print` *value* write *value*, followed by CR

`file-show` *value* first write the agent description, then write *value*, followed by CR

Comment: Windows-centric text editors may not display CR as an end-of-line. For example, Notepad will not display then at all.

Example: Write Space-Separated Values (SSV)

Try this in the command center:

```
carefully [file-delete "temp.txt"] []
file-open "temp.txt"
file-print "minimum mean maximum"
file-type 10 file-write 15 file-write 20
file-print ""  ;;terminate line with CR
file-close
```

Example: Write Comma-Separated Values (CSV)

Try this in the command center:

```
carefully [file-delete "temp.csv"] []
file-open "temp.csv"
file-print "minimum,mean,maximum"
let vals [10 15 10]
file-type first vals
foreach but-first vals [
  file-type "," file-type ?
]
file-print ""  ;;terminate line with CR
file-close
```

Multiple Open Files

You must always use `file-open` to specify what file you want to interact with. E.g.,

```
file-open "log1.txt"
file-open "log2.txt"
file-write "this goes in log2.txt"
file-close
file-open "log1.txt"  ;;required!
file-write "this goes in log1.txt"
file-close
```

File-Based Input

In order to read external information into a program, the following commands are often useful.

`file-read-line:` read the next line and return it as a string (without terminators)

`file-read:` read the next "constant" (e.g., number, list, or string) and return it

`file-at-end?:` report true if last character of file has been read

Of course we will still need to open and close our files.

`file-open` *string*: open a file for reading or appending (but not both)

`file-close:` close an open file

Example: file-read-line

Try this in the command center:

```
file-open "temp.txt"
print file-read-line
file-close
```

Note use forward slashes in your paths.

Example: File-Based Input

Suppose the `nldata01.txt` looks like:

```
pxcor pycor n-turtles
0 0 5
1 0 3
```

You could handle this as follows:

```
file-open "nldata01.txt"
let trash file-read-line ;; discard header line
while [not file-at-end?] [
  ask patch file-read file-read [sprout file-read]
]
file-close
```

Example: More File-Based Input

Assume a 20x10 world of patches.
Suppose patches have a *foo* attribute. Suppose you have created `foo.txt` as:

```
1 2 3 4 ... 200
```

Suppose patches also have a *bar* attribute. Suppose you have created `bar.txt` as:

```
200 199 198 197 ... 1
```

Give each patch one of these values for its *foo* and *bar* attributes as follows:

```
to setupPatches
  let patch-list sort patches
  file-open "foo.txt"
  foreach patch-list [ask ? [set foo file-read]]
  file-close
  file-open "bar.txt"
  foreach patch-list [ask ? [set bar file-read]]
  file-close
```

Example: File-Based Input (Python)

```
fin = open('nldata01.txt', 'r')
trash = next(fin)
data = dict()
for line in fin:
    x, y, n = map(int, line.split())
    data[(x,y)] = n
fin.close()
```

CSV Extension

NetLogo provides a CSV extension for reading and writing CSV data:
`http://ccl.northwestern.edu/netlogo/docs/csv.html`
CSV stands for comma-separated values. This is an internationally recognized data-exchange format. See
`https://tools.ietf.org/html/rfc4180`
The CSV extension accommodates some common deviations from the CSV standard. For example, it allows specification of a different delimiter than the comma. However, the standard for scientific data exchange is a comma as the field delimiter and a point as the decimal separator.

Example: File-Based Input (CSV)

```
extensions [csv]

to setup
  file-close-all
  ca
  file-open "c:/temp/temp.csv"
  ;;if there is a header line, use it or discard it
  let _trash file-read-line
end

to get-one-line
  file-open "c:/temp/temp.csv"
  if file-at-end? [ stop ]
  let _line file-read-line      ;; read the line into a str
  let _data csv:from-row _line  ;; convert the string to a
  ;;now do whatever you want with the data
end
```

Example: File-Based Output (CSV)

```
extensions [csv]

to setup
  ca
  file-close-all
  carefully [file-delete "temp.csv"] []
  file-open "c:/temp/temp.csv"
  file-print "x,y,z"
  file-close
end

to write-one-line
  let _mylist (list x y z)
  file-open "c:/temp/temp.csv"
  let _mystr csv:to-row _mylist
  file-print _mystr
  file-close
end
```

BehaviorSpace and File Output

If you want to make your own output files during BehaviorSpace runs, use the `behaviorspace-run-number` primitive. Alternatively, produce filenames based on parameter values.

```
["globalA" 1 2 3]
["gloablB" 4 5 6]
file-open (word "myfile-" globalA "-" globalB ".txt")
```

Of course you can combine these two approaches.
If you needs even more flexibility, consider Charles Staelin's `pathdir` extension. It might still be here: `http://sophia.smith.edu/~cstaelin/NetLogo/pathdir.html`

Bundled Extensions

See: Help > NetLogoUser Manual > Extensions

A standard NetLogo installation bundles a few extensions, which are located in Extensions subfolder of the NetLogo installation folder. These include:

- `table` is often needed; `array` and `matrix` can also be useful `http://ccl.northwestern.edu/netlogo/docs/arraystables.html` `http://ccl.northwestern.edu/netlogo/docs/matrix.html`
- `nw` provids a collection of networkd-analysis primitives
- `profiler` provides an experimental but useful profiler
- `sound` provides MIDI sounds and sound file playback
- `gogo` interacts with a GoGo board for simple robotics
- `bitmap` and `qtj` (Qucktime) are useful for movie making and interacting with images
- `gis` provides basic GIS capabilities

Other Important Extensions

See https://github.com/NetLogo/NetLogo/wiki/Extensions

shell https://github.com/NetLogo/Shell-Extension/

stats

- https://github.com/cstaelin/Stats-Extension/releases
- http://sophia.smith.edu/~cstaelin/NetLogo/StatsExtension-v1.2.1.pdf

R

- http://r-ext.sourceforge.net/
- [thiele.grimm-2010-envsoft]

numanal Numerical Analysis (roots and optima) http://sophia.smith.edu/~cstaelin/NetLogo.html http://sophia.smith.edu/~cstaelin/NetLogo/numanal.html

web https://github.com/NetLogo/Web-Extension/

NetLogo-Mathematica Link

You can control NetLogo from Mathematica: `http://ccl.northwestern.edu/netlogo/docs/mathematica.html`
There is a tutorial: `http://ccl.northwestern.edu/netlogo/5.0/docs/NetLogo-Mathematica%20Tutorial.pdf`

What is an Array?

A NetLogo array is a **fixed-length** collection of objects, such as a collection of 100 numbers. Let us make an array of length 100, full of zeros.

```
array:from-list n-values 100 [0]
```

Indexing is zero-based. That is, as with lists, the first item has index 0. The second item has index 1. And so on.
Documentation: `http://ccl.northwestern.edu/netlogo/5.0/docs/arraystables.html`

Arrays Are Mutable

School-Lockers Analogy:
You can think of an array as a bit like a row of school lockers. Each student has an assigned locker, where s/he stores stuff. The stuff in a locker can change. Similarly, we can change the corresponding item in the array. Let us change the first value of `myarray` to `999`.

```
array:set myarray 0 999
```

Increment a Single Item

Let us increment the first value of `myarray` by `1`.

```
array:set myarray 0 (array:item myarray 0 + 1)
```

Let us decrement the second value of `myarray` by `1`.

```
array:set myarray 1 (array:item myarray 1 - 1)
```

When array items represent the values of an attribute of agents, then a transfer can be represented as a decrement of one item combined with an increment in another.
This is a bit like taking something out of one student's locker and putting it in another student's locker.

Arrays vs Lists

NetLogo uses the term "array" substantially differently than many languages. In particular, a NetLogo array is a fixed length container of objects, which need not be of a common type. (E.g., they need not all be numbers.) Array items can be quickly accessed or replaced, using their indexes.

```
array:set myarr 0 (array:item myarr 0 + 1)
```

Since lists are immutable, the equivalent operation on lists is a bit more awkward.

```
set mylst replace-item 0 mylst (item 0 mylst + 1)
```

Array Limitations

While arrays can be useful when one needs a fixed-length container with changing contents, they are limited. For example, to copy an array, you need a list intermediary:

```
let acopy array:from-list array:to-list myarr
```

(See `http://ccl.northwestern.edu/netlogo/docs/arraystables.html` for details on `array:to-list`.)
Similarly, `max` and `min` work only on lists, so you will again have to use `array:to-list` if you want to use these commands.

Array Limitations: Filtering

We can only apply `filter` to lists. So if you want to filter an array `a`, you need to convert it first:

```
filter [? < 3] array:to-list a
```

What is a Table?

A NetLogo array is a mapping from keys to values.

Documentation: `http://ccl.northwestern.edu/netlogo/5.0/docs/arraystables.html`

Tables Are Mutable

```
to-report counts [#lst]
  let _t table:make
  foreach #lst [
    let _k ?
    ifelse table:has-key? _t _k [
      table:put _t _k (1 + table:get _t _k)
    ][
      table:put _t _k 1
    ]
  ]
  report _t
end
```

Colors Are Numbers

When we show the color of an agent, the result is a number. NetLogo represents colors by numbers in [0 .. 140).
`http://ccl.northwestern.edu/netlogo/5.0/docs/programming.html#colors`
The command `random-float 140` picks a random number in this range.
As a matter of convenience, NetLogo also defines named aliases for some colors. (E.g., white = 9.9.)
So the following are equivalent:

```
ask mypatch [set pcolor white]
ask mypatch [set pcolor 9.9]
```

(You can see this equivalent by entering `show white` at the command line.)

scale-color

For useful examples (including shading and tinting), see `http://ccl.northwestern.edu/papers/ABMVisualizationGuidelines/palette/doc/NetLogo%20Color%20Howto%201.htm`

NetLogo File Format

NetLogo files use a plain text file format.
`https://github.com/NetLogo/NetLogo/wiki/Model-file-format`
This means that all the widgets in the `Interface` tab can be edited with a text editor. The widget-format documentation is online.
`https://github.com/NetLogo/NetLogo/wiki/Widget-Format`

NetLogo with GIS

We do not cover the GIS capabilities of NetLogo. Simple example are in the NetLogo Models Library.
Here is an introduction:
https://simulatingcomplexity.wordpress.com/2014/08/20/turtles-in-space-integrating-gis-and-netlogo/
Also, see this interesting Artificial-Anasazi tutorial:

- http://modelingcommons.org/browse/one_model/2354#model_tabs_browse_info
- http://scientificgems.wordpress.com/2014/06/07/revisiting-artificial-anasazi-a-tutorial-part-1/
- http://scientificgems.wordpress.com/2014/06/10/revisiting-artificial-anasazi-a-tutorial-part-2/

NetLogo with Java

We do not cover using Java with NetLogo, but see these useful materials:

- http://scientificgems.wordpress.com/2013/12/11/integrating-netlogo-and-java-part-1/
- http://scientificgems.wordpress.com/2013/12/12/integrating-netlogo-and-java-2/
- http://scientificgems.wordpress.com/2013/12/13/integrating-netlogo-and-java-3/

References

Axtell, R., Axelrod, R., Epstien, J. M., and Cohen, M. D. (1996). Aligning simulation models: A case study and results. Computational and Mathematical Organization Theory, 1:123–141.

Resnick, M. (1997). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. MIT Press.

[thiele.grimm-2010-envsoft]

Legalities