Semantics of Netlogo

Sara Hartse

May 11, 2015

1 Introduction to Logo

Netlogo is a member of the Logo family, a collection of Lisp-like programming languages designed as educational tools. The first iteration of Logo was developed in 1967 and it's been mutated and re-imagined many times since. One of the most distinctive aspects of Logo is the notion of 'turtle graphics,' which allows the user to draw and create designs and patterns by providing instructions to the 'turtle' who moves around the screen, tracing its path as it goes.

One of the developers of Logo, Seymour Papert says that this introduces the concept of programming "through the metaphor of teaching the Turtle a new word" (Papert, 6)[1]. For example, the student can teach figure out how to get the Turtle to draw a square, repeating the commands `forward 10 right 90` four times. But this is tedious, ideally the turtle would know what you mean when you tell it to draw a square. Typing the command `square` initially produces the error message "`I don't know how to square`", but by binding the square keyword to the instructions for drawing, the Turtle's knowledge base is expanded. The procedure can be parameterized to accept the desired side length and more elaborate patterns can be developed.

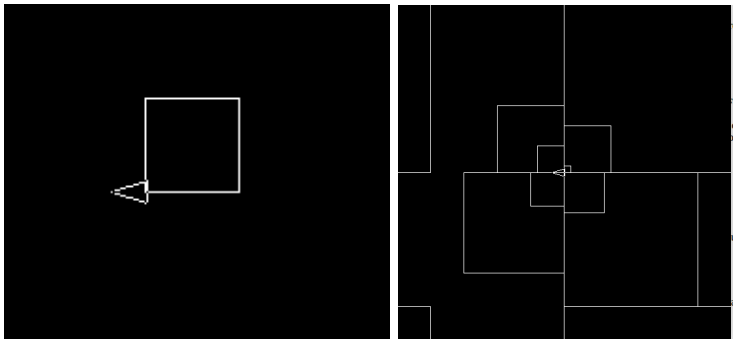

Figure 1: Progression of a square drawing program

In addition to developing programming skills, students can use the turtle as a simple model for reasoning and learning about the world with a constructivist approach. Constructvism is a philosophy of epistemology and education that emphasizes the need for learners to create knowledge about the world based on the interactions of their ideas and experiences. It's often associated with the idea of learning by doing. Papert claims Logo can act as a facilitator of this type of exploration and discovery. An example he gives is figuring out how to make a turtle draw a circle or a triangle with only forward, left and right commands. In a math class a student might see the equation for a circle and might be taught that the internal angles of a triangle must add up to 180 degrees, but Papert argues that by experimenting with and debugging turtle instructions, a student can construct geometric knowledge themselves.

Papert wrote his book in the 1980s. At the time he observed the growing prevalence of computers in peoples' day to day lives and rightly predicted their

[1]Papert, Seymour. Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic, 1980. Print.

future ubiquity. He also predicted the usage of computers in a classroom setting, but warned against using them simply as new tool for drilling and testing students. He believed that programs like Logo had imense value as an exploratory tool that could give students a greater ownership of their knowledge than the current educational systems, especially for subjects like mathematics, can facilitate.

The metaphor of the turtle is also an interesting feature as it gives the student an actor to with which to identify. Every line of Logo code written is designed to be thought of as directed at the turtle. Making the conceptual leap of writing code to be interpreted and executed by a computer is softened by the turtle figure as turtles are a lot less complicated and intimidating than a Java program, for instance. Further, the turtle offers a more physical model to reason about. When trying to get the turtle to a draw a certain pattern, it's easy to figure out what it needs to do based on what actions you take when stepping through the pattern.

The implementation of Logo I've been experimenting with will writing this is an opensource interpreter created at Berkley in 1993 and is available here: `https://www.cs.berkeley.edu/~bh/logo.html`

2 Netlogo

Netlogo is an agent-based modeling language which was first developed by Uri Wilensky at Tufts in 1999 and then moved to Northwestern. It was also designed as an educational tool and continues preserves the central turtle metaphor. However it makes several departures from Logo, most notably in that it was developed to be a tool for modeling and exploring multi-agent systems. Which basically means that instead of controlling one turtle, you're instructing many plus a few other types of agents. It's been used educationally and academically for simulating non-linear, non-deterministic systems, modeling everything from Ant Colony optimization to the spread of disease in Tasmanian devils.

Netlogo is implemented in Java and Scala. It's based on a framework in which you have collections of agents with internal variables and certain commands they can execute with modify these variables and the state of the world.

For example, a program that creates 10 agents (turtles) and has them move forward and draw as the go:

```
create-turtles 10
ask turtles [pendown
    forward random 10]
```

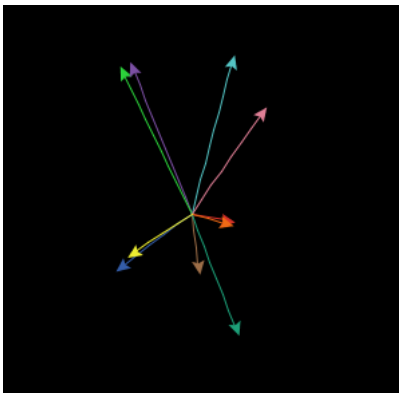

Figure 2: Multi-agent graphics

The fact that the user is controlling the action of many agents makes intricate patterns more simple to draw. For example, a program that has the agents draw a spiral pattern by slightly adjusting their heading and moving forward.

```
repeat 10 [
  ask turtles [set heading heading + 5
forward 1 ]]
```

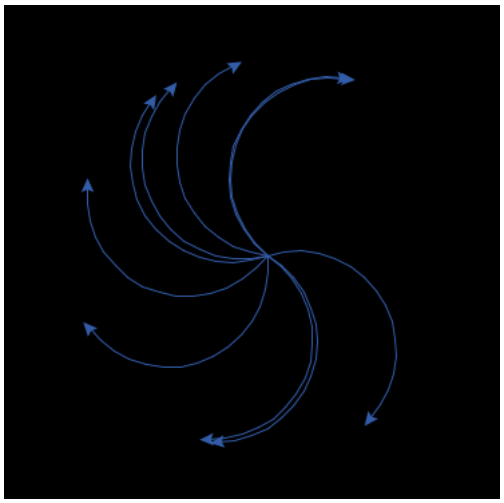

Figure 3: Spiral

Because there are now multiple agents, the agents can be programmed behave dependently, basing their actions on characteristics of one another. For example, the following program instructs every turtle to select the turtle with id value one higher than their own, adjust their heading to face that turtle and then move forward slightly. These simple instructions result in an interesting

global pattern. The exploration of emergent behaviors from multi-agent systems is one of the focal points of Netlogo.

```
repeat 10 [
  ask turtles [set heading towards turtle ((who + 1) mod 10)
forward 1 ]]
```

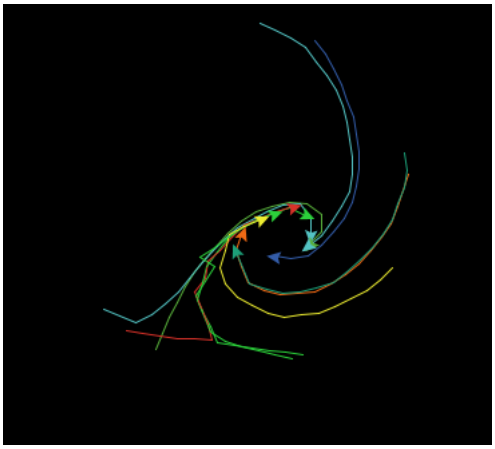

Figure 4: Turtles following each other

This pattern of programming is a little unfamiliar. Typically, if you invoking some kind of procedure that belongs to something, you'd expect a method call, something like:

```
<agent>.forward(1)
```

But this structure of `ask`-ing agents to perform an action makes sense because most of the time you're interested in collective behavior. So it makes sense to not be manipulating a single agent, but a set of them. Therefore, you pass an 'agentset' a command and if it's composed of the correct type of agent to execute that command, then they do, otherwise a type error.

2.1 Agents

There are four types of agents available in a Netlogo model.

- Observer - There is always a single observer, it is a top level agent that has fields that act as global variables and has access to most of the higher level commands
- Turtles - These are the agents that have mutable spacial data as well as other internal variables. They tend to move around, interact with each other and change their environment. The can be both added and removed from the model.

- Patches - These are agents that represent fixed spacial locations, chucks of space in the model. They cannot be created or destroyed but they also have internal variables that can be manipulated through the progress of the simulation.
- Links - These won't be discussed at length, but links are agents that do not have spatial information, instead they have turtles which they connect, representing edges of a graph and with their own mutable internal variables.

2.2 Formal Semantics

I'll be outlining big step operational semantics for a subset of Netlogo. I'm restricting it to only 3 agents (leaving out links and there are just way to many commands to include). I'm trying to highlight the language's core functionality

$M \in$ Model ::= $\langle$World, Program$\rangle$
$W \in$ World ::= $\langle$Observer, TurtleSet, PatchSet$\rangle$
$A \in$ Agent ::= Observer [`tick <globals>*`]
| Turtle [`who color heading xcor ycor shape breed size <breeds-own>*`]
| Patch [`pxcor pycor pcolor <patches-own>*`]
$AS \in$ Agentset ::= Turtle*
| `no-turtles`
| Patch*
| `no-patches`
| <breed>*
| `no-<breed>`
$AV \in$ Agent-Var ::= `globals` I_{var}*
| `turtles-own` I_{var}*
| `patches-own` I_{var}*
| `<breeds>-own` I_{var}*
$Prog \in$ Program ::= AV* $Proc$*
$Proc \in$ Procedure ::= `to` S_{name} I_{formal}* $C*_{body}$ `end`
$R \in$ Reporter ::= V
| (`list` R_{elm}*)
| binop R_1 R_2
| unop R
| TP
$TP \in$ TurtlePatch-Reporter ::= `self`
| `distance` R_A
| R_{AS} `in-radius` $R_{patches}$
| `other` R_{AS}
| `towards` R_A
$C \in$ Command ::= OTP
| O
| T
| P
$OTP \in$ ObserverTurtlePatch ::= $Proc$ R_{arg}*
| `ask` R_{AS} $C*$
| `ifelse` R_{cond} C_{then} C_{else}

$$
\begin{array}{rl}
 & \mid \texttt{show}\ R \\
 & \mid \texttt{set}\ I\ R_{val} \\
 & \mid \texttt{let}\ I\ R_{val} \\
 & \mid \texttt{repeat}\ R_N\ C*_{body} \\
 & \mid \texttt{while}\ R_{cond}\ C*_{body} \\
 & \mid \texttt{error}\ R \\
O \in \text{Observer} ::= & \texttt{clear-all} \\
 & \mid \texttt{create-<breed>}\ R_{count}\ C*_{action} \\
 & \mid \texttt{set-patch-size}\ R_{count} \\
 & \mid \texttt{tick} \\
T \in \text{Turtle} ::= & \texttt{hatch}\ R_{count}\ C*_{action} \\
 & \mid \texttt{forward}\ R_{steps} \\
 & \mid \texttt{back}\ R_{steps} \\
 & \mid \texttt{right}\ R_{degrees} \\
 & \mid \texttt{left}\ R_{degrees} \\
 & \mid \texttt{pen-down} \\
 & \mid \texttt{die} \\
P \in \text{Patch} ::= & \texttt{sprout}\ R_{count}\ C*_{action} \\
V \in \text{Value} ::= & A \mid AS \mid N \mid I \mid B \mid S \\
N \in \text{Number} ::= & \{\texttt{... -2 -1 0 1 2 ...}\} \\
B \in \text{Bool} ::= & \{\texttt{true false}\} \\
S \in \text{String} ::= & \{\texttt{x y set forward die ...}\} \\
I \in \text{Ident} ::= & \text{String - Keywords} \\
\text{binop} = & \{\ \texttt{+ - = * / \^{} mod map filter of with}\} \\
\text{unop} = & \{\texttt{not is-agent? is-agentset? is-boolean? is-list? is-number?} \\
 & \texttt{but-last but-first is-patch? is-string? is-turtle? empty?} \\
 & \texttt{sort one-of random count}\}
\end{array}
$$

The basic idea behind the Netlogo semantics is that worldstates are transformed by programs. A world is represented by the Observer, a collection of Turtles and a collection of Patches and the world state consists of membership of these collections and the internal variables of the agents. A Program is applied to a World and a new World is produced. The initial step of Program application is the assignment of Agent-Vars. These are grouped based on agent and they expand the set of fields which can be modified internally by the individual agents.

$$\frac{\langle Prog, Agents\rangle \twoheadrightarrow_{Prog} Agents_{new}}{\langle Prog, W_{inital}\rangle \twoheadrightarrow_{M} W_{final}}$$

$$\frac{\langle AV*, Agents\rangle \twoheadrightarrow_{expand} Agents'; \langle Proc, Agents'\rangle \twoheadrightarrow_{Proc} Agents_{new}}{\langle AV, Proc*, W_{inital}\rangle \twoheadrightarrow_{Prog} W_{final}}$$

$$\frac{\langle C*, Agents\rangle \twoheadrightarrow_{C} Agents_{new}}{\langle Proc, W_{inital}\rangle \twoheadrightarrow_{Proc} W_{final}}$$

The next step is the evaluation of Procedures. This is started by a particular procedure being triggered, often by a 'start' button. A procedure consists of a sequence of commands. Some of these commands don't change the worldstate,

many are control-flow commands like `ifelse, while, repeat` and `ask`. `show` for example just prints the value of a reporter without modifying any state. `let` also does not affect the worldstate, only binding values locally. Other commands, like `create-turtles, die, sprout, clear-all` and `hatch` affect the world state by adding or removing turtles from the World's collection of Turtles. Finally, there are commands which change the internal values of agent variables. `set, forward, back, left` and `right` are examples.

The following are examples of some of the most imporant commands. *filter*, *map*, *remove*, and *add* are meta functions that work as you'd expect. *new(N, vars*)* creates N new Turtles their internal fields set to *vars*.

$$\frac{R \twoheadrightarrow_R B; if\ B\ C_{then},\ otherwise\ C_{else}}{\texttt{ifelse}\ C_{then} *\ C_{else}* \twoheadrightarrow_C C_{final}}$$

$$\frac{R \twoheadrightarrow_R AS; AS.map(\langle C_{body}*, A\rangle)}{\texttt{ask}\ R\ C_{body}* \twoheadrightarrow_C W_{final}}$$

$$\frac{R \twoheadrightarrow_R V; A_{final} = A[(I, V)]}{\texttt{set}\ I\ R \twoheadrightarrow_C W_{final}}$$

$$\frac{R \twoheadrightarrow_R N; AS = new(N, 0, 0)); \texttt{ask}\ AS\ C*; W_{final} = Agents.add(AS)}{\texttt{create-turtles}\ R\ C* \twoheadrightarrow_C W_{final}}$$

$$\frac{R \twoheadrightarrow_R N; AS = new(N, Turtle(T.vars)); \texttt{ask}\ AS\ C*; W_{final} = Agents.add(AS)}{\langle \texttt{hatch}\ R\ C*, T\rangle \twoheadrightarrow_C W_{final}}$$

$$\frac{R \twoheadrightarrow_R N; AS = new(N, (P.pxcor, P.pycor)); \texttt{ask}\ AS\ C*; W_{final} = Agents.add(AS)}{\langle \texttt{sprout}\ R\ C*, P\rangle \twoheadrightarrow_C W_{final}}$$

$$\frac{W_{final} = Agents.remove(A)}{\langle \texttt{die}, A\rangle \twoheadrightarrow_C W_{final}}$$

$$\frac{W_{final} = Agents.remove(Turtles)}{\texttt{clear-all} \twoheadrightarrow_C W_{final}}$$

The next type of expression is reporter. Reporters are expressions that evaluate to Values, which cannot stand alone but are often needed to carry out commands. Some reporters can be computed universally, by any Agent. For example `count`only needs to be provided an Agentset and it just needs to find its length. Reporters access local variables, evaluate boolean statements and restrict Agentsets.

$$\frac{R \twoheadrightarrow_R AS; N = AS.size()}{\texttt{count}\ R \twoheadrightarrow_R N_{ans}}$$

$$\frac{R_1 \twoheadrightarrow_R AS; R_2 \twoheadrightarrow_R B; AS_{ans} = AS.filter(B)}{R_1\ \texttt{with}\ R_2 \twoheadrightarrow_R AS_{ans}}$$

$$\frac{R \twoheadrightarrow_R A; V_{ans} = A.lookup(I)}{I\ \texttt{of}\ R \twoheadrightarrow_R V_{ans}}$$

$$\frac{R \twoheadrightarrow_R AS; A_{ans} = AS.get(random\ AS.size())}{\texttt{one-of}\ R \twoheadrightarrow_R A_{ans}}$$

Other Reporters can only be executed by Turtles or Patches. `in-radius` requires that it be executed by a Turtle and finds all the other agents with the specified radius of that turtle's location. These are evaulated by passing along the agent executing the command into the evaluation.

$$\frac{R_1 \twoheadrightarrow_R AS; R_2 \twoheadrightarrow_R N; AS_{ans} = AS.filter(inradius)}{\langle R_1\ \texttt{in-radius}\ R_2, A\rangle \twoheadrightarrow_R AS_{ans}}$$

where *inradius* is:
$(A.xcor - N < xcor < A.xcor + N\ and\ A.yxor - N < ycor < A.ycor + N)$

$$\frac{R \twoheadrightarrow_R AS; AS_{ans} = AS.filter(not(x == A))}{\langle \texttt{other}\ R, A\rangle \twoheadrightarrow_R AS_{ans}}$$

2.3 Errors

One area that was not discussed earlier with the semantics is the way that Netlogo deals with errors. Netlogo has several types of errors that appear at compile and at runtime. The editor has a "check" button which compiles the code and does and certain amount of type and error checking, but it's confusing as to what exactly is being checked.

2.3.1 Constant folding

Programs like with errors along the lines of `print 1 + "string"`, `print 1 / 0` or `print first (list )` throw a runtime error when you hit the "check" button. Netlogo is finding many simple errors at compile time, before the simulation is even started.

However, things like `let x "string`, `print 1 + x` do not. So clearly the program is not actually being evaluated in its entirety. What's going on is an optimization called constant folding which recognizes and evaluates constant expressions at compile time. The complier does not use constant propagation, so identifier references are not simplified. However, identifier bindings are evaluated, `let x "string" + 1` does produce an error.

Netlogo is not lazily evaluated. A runtime error that appears in any branch of the program will be caught.

2.3.2 Type checking

The other type of error that appears at the "check" stage or compile time are the type-errors which pertain to the types of agents who are allowed to execute each command. Some commands like `print`are universal can be executed by any agent (the Observer, Turtles, Patches or Links). Most are restricted to a certain agent and can only be executed by 'asking' a certain type of agent to perform them. Any command not nested within an `ask` command is assumed to be directed towards the Observer. For example, the command `set pcolor green` cannot be executed by the Observer, but instead must be redirected to patches, `ask patches [set pcolor green]`.

Netlogo's complier includes a type checker which runs over the body of the program to determine what the "type" of each procedure is and whether or not it is being executed by the correct collection of agents each time it appears. The basic idea is to make multiple passes through the code and determine who the code should be 'usable by.' Initially, all procedures are assumed to be "OTPL" (Observer, Turtles, Patches, Links) usable, but then are restricted. These restrictions are propagated forward and downward and between procedures. The three main steps are:

1. Propagation within procedures

   - Forward propagation: `forward 1 sprout 1` fails because `forward` is a Turtle only usable command and `sprout` is only for Patches.
   - Downward propagation: Some commands have optional nested commands. For example, adding commands after `create-turtles` allows you to give commands to the turtles just created.

     ```
     create-turtles 1
     [sprout 1]
     ```

     Fails because the nested block is directed at turtles.

2. Restrictions on which agents can execute which procedures. For example, the following procedure would be markable as a Turtle usable procedure as the commands in it are Turtle usable.

   ```
   to triangle
     pendown
     forward 10
     left 120
     forward 10
     left 120
     forward 10
   end
   ```

3. Propagation in procedures which call other procedures. The following procedure would fail to typecheck because `triangle` is Turtle usable but `clear-all` is Observer only.

   ```
   to go
     clear-all
     triangle
   end
   ```

2.3.3 Dynamic errors

The final type of errors are those which appear when code is executed fully executed. These include commands that result in things like trying to reference a dead turtle or dividing by the length of an empty agentset. Another trying to

modify a local variable that belongs to the wrong "breed" of turtle. A breed is really just syntactic sugar for an agentset of turtles that have the local variable breed set to the same value.

Because of the way this is implemented, there is no static distinction between breeds - you can ask an agent of any breed to set any breed's local variable which errors at runtime. This makes sense from a structural perspective, a breed is really just another local variable. It's by default turtle, and an agent remains a turtle even if you assign it another breed.

Somewhat confusingly, you can also instruct a turtle to set a patch local variable, which executes by setting that variable of the patch the turtle is on. Asking a turtle to set a link's variable errors as expected. A turtle is free to modify globals

2.4 Operational Reasoning

2.4.1 Determinism

One important feature of Netlogo is that it is not deterministic. This is due to the fact that evaluation order of Agents is not fixed. For example, if we write a program telling all Turtles to print their id number, the order we'll see them is random.

```
create-turtles 5
ask turtles [show who]
```

Adding the id number of each Turtle to a global variable, asking them to ask the Turtle one greater than it to die and then printing the variable is will not be the same every time.

```
globals [x]
to go
  create-turtles 10 [set x x + who
                     ask turtle ((who + 1) mod 10) [die]]
  show x
end
```

Or, if we write a program telling Turtles to change their size and pause mid-step, we can see that Turtles are acting in a non-deterministic fashion.

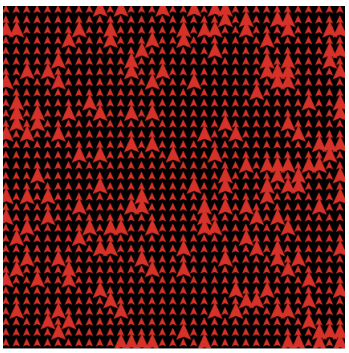

Figure 5: Turtles execute commands in a random order

This rule holds true for Patches and Links as well and makes sense for a couple reasons. First is that the datastructure representing these agents is a set, which is inherently unordered. Second, these models are designed to be used for random simulations and imposing an evaluation order for agents is a implicit parameter that could lead to artifacts unrelated to the the concept being modeled.

There are several consquences of non-determinism. One is the partical consideration that Netlogo programs can be harder to debug. It's difficult to employ the "starting at the top of your code and then stepping through each line in your head" strategy because you have so many different agents executing them and not nesscarily in the same order that they did last time you ran the code. For example, the following program passes the inital check by throws a division-by-zero error after it's run for a moment:

```
ask patches [sprout 1]
ask turtles [show 1 / (count other turtles in-radius 1)
             die]
```

What happens is that eventually enough of the turtles have excuted the `die` command that other turtles have no one in with a radius of 1 around them. The problem is that this error will be thrown by a different turtle every time and there's no information about that specific turtle than can help us track down the bug, it's a feature of the global environment. Figuring out the worldstates that lead to errors can be a delicate process, especially as these worldstates are non-determistically generated.

This means that Netlogo programs cannot guarantee confluence.

2.4.2 Static Anaylsis

As was previously discussed, there are a certain about of errors that a program can be statically checked for. Simple constant reporter computations are folded before runtime and the body of the program is passed over to check that that all commands and reporters are being executed by the correct type of Agent.

3 Conclusions

Netlogo is a very large, interesting language that brings a new approach to how people typically think about programming. Like Logo, it embraces a very physical, tactile interface with a Turtle actor for the user to identify with and reason through. It also focuses on the notion of Agent-based Modeling and the behavior of large, non deterministic systems. The Netlogo language is huge. The syntax outlined above reflects a small kernel and even leaves out one of the types of agents (Links). Netlogo has an active contributing community, presumably responsible for it's immense size as a quote from the Github page indicates: "The first law of adding primitives is: DON'T. The language is much too big already. In the name of all that is holy, write an extension instead!"