Procedures: Algorithms and Abstraction

5

5.1 Objectives

After completing this module, a student should be able to:

- Read and understand simple NetLogo models.
- Make changes to NetLogo procedures and predict the effect on the simulation.
- Name the control structures sufficient to express all components of programs.

5.2 Definitions

- Algorithm
- Procedural abstraction
- Procedure

5.3 Motivation

Part of calling science and engineering "disciplines" is the implication that they often follow structured processes and procedures for data collection, analysis, and evaluation of their respective topic areas. Thus, scientists and engineers often create and/or follow those processes and procedures, many times using computing. But before learning more about procedures and computers and their abstraction, we need to first take a brief look at the concept of an algorithm. An *algorithm* is a mathematical term for a clear set of instructions that, when followed, solves a particular problem. Examples include methods you may have learned in school and life:

- Multiplying large numbers by hand
- Constructing a perpendicular to a line using a ruler and compass

K. Brewer, C. Bareiss, *Concise Guide to Computing Foundations*,
DOI 10.1007/978-3-319-29954-9_5

- Constructing an angle bisector using a compass
- Brushing your teeth
- Preparing a hotdog in a microwave

Discuss It!
Consider how you multiply two large numbers by longhand. Could you teach someone this algorithm?

To get the correct result from an algorithm, you must understand each instruction and carry them out in the right order. You also need some basic knowledge such as a *times table* (for multiplication) or entering cooking time (on a microwave). In these examples, a human is performing the algorithm in order to solve a problem. The key to an algorithm (i.e. set of instructions) is that it can work with many different inputs (data values) and therefore be reused and repeated. Algorithms are used in science and engineering disciplines to solve common mathematical problems.

5.4 Procedures

Programmers sometimes use existing algorithms or they may design algorithms of their own to find solutions. Once a clear set of instructions to solve a problem exists, they must be written in a programming language so the algorithm can run on a computer. The entire algorithm written in a programming language is called a program or set of *procedures*. The computer must "know" some primitive things like numbers and simple math operations, as well as various *procedural* instructions.

A program can be separated into parts called subprograms. The *structured program theorem* (Böhm and Jacopini 2011) states that every algorithm can be implemented in a programming language that combines subprograms in only three specific ways, called "control structures". These three control structures are:

- Executing one subprogram, and then another subprogram (*sequence*).
- Executing one of two subprograms according to the value of a Boolean condition (*selection*).
- Executing a subprogram repeatedly while a Boolean condition is true (*repetition*).

We will learn about algorithms, procedures, and programming by using the NetLogo (Wilensky 1999) system and programming language. The objective is not for you to become a good NetLogo programmer, but to have some experience

and practical examples of a programming language using a defined set of built-in commands and control structures.

5.5 Control Structure Example

Say we want to create a repetitive spiral graphic that looks something like that shown in Fig. 5.1.

We can create this using NetLogo using a single turtle (the NetLogo term for an agent that can be programmed to move around the simulated world) with the pen down. All three control structures in the structured program theorem are needed, and the entire algorithm can be written in a single command:

repeat 72 [ifelse heading mod 10 < 5 [set color yellow] [set color green] repeat 4 [forward 10 right 90] right 5]

Answer It!

Q05.01: What part(s) of the command (algorithm) reflect a *sequence*?
Q05.02: What part(s) of the command (algorithm) reflect a *selection*?
Q05.03: What part(s) of the command (algorithm) reflect a *repetition*?

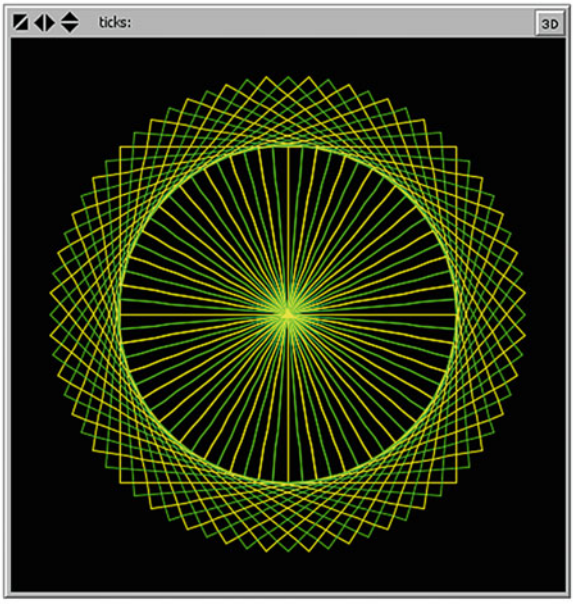

Fig. 5.1 Spiral graphic example

5.6 Procedural Abstraction

By now you have used examples of NetLogo control structures for each of the three types: *sequence, selection,* and *repetition*. There is one more powerful programming concept to cover – *procedural abstraction*. Because it is difficult to figure out exactly how to write a complex procedure and you often want to use them more than once, most programming languages have a way to give a procedure a name and to invoke it when needed. Once a name is associated with the procedure, you can run all the instructions in the procedure just by using the associated name. It's like a new command and can be used just like built-in commands. By giving a procedure a name, the set of commands is larger and the programming language is more expressive and more abstract.

Answer It!

Q05.04: Write a NetLogo procedure to have a turtle draw an equilateral triangle with sides of length 10.

5.7 Theater Lights Part 1

This model was inspired one day while driving past a local movie theater and watching the flashing lights moving around each sign. You will create the user interface and will program the algorithms that will perform the model behaviors. It is strongly suggested you read through the NetLogo Tutorial appendix prior to continuing in this module.

We will approach creating the model in two major efforts:

- we will first arrange turtles (our lights) around the edges of a 20×12 rectangle on every other patch.
- we will then make them move counter-clockwise to simulate the moving flashing.

Let us begin. Start NetLogo and create the basic *setup* and *go* buttons as instructed in the appendix. Ensure the *setup* procedure includes the "clear-all" and "create-turtles 1" commands.

The next thing to do in the setup procedure is to ask the only turtle to set its (x, y) location to (10, 6) the upper right corner of the rectangle. At the same time, we should set the color to *white* and make its heading north.

```
ask turtle 0 [
  set xcor 10
  set ycor 6
```

```
  set color white
  set heading 0
]
```

Try the code in the *setup* button and make sure it does what you've planned.

A turtle can make a copy of itself by using the command *hatch 1*. You'll use this command with the first turtle to leave lights along the path as it travels around a rectangle in a counter-clockwise direction. Remember that you plan to put a light on every other patch along the outside of a 20×12 rectangle.

To add the top row of lights, try adding these commands inside the *ask turtle 0* sequence (after the previous code).

```
left 90
repeat 10 [
 forward 2
 hatch 1
]
```

Test your modified procedure. Make sure you get it working (the top row lights are created) before you continue. Try sliding the speed slider left to slow the simulation down. Test your procedure again and watch it doing each command one at a time in response to pushing the *setup* button.

You should save your model periodically while you are modifying it. Do that now by selecting *File* and *Save* and naming your model (e.g. Lights1.nlogo).

Now you need to add commands to the *setup* procedure to complete the rectangle. The rectangle is 20×12. You repeated 10 times along the top because you want a light every other patch. You should therefore repeat 6 times on the sides. Each turn at the corners is 90° to the left. Don't continue until your *setup* procedure places all the lights around the rectangle. The corners of the rectangle should be at (10, 6), (−10,6), (−10, −6), and (10, −6).

You should add one more command to the end of the *setup* procedure. The turtle that traveled around the rectangle leaving copies of itself along the way using the *hatch 1* command will end up at its starting location in the upper right corner. But there is also a copy of it in that same starting location. To avoid duplication, we need to remove the turtle that hatched all the others. To remove turtles from the world, we use the command *die* to ask a turtle to leave this world. Therefore ask the turtle to "die" at the end of the "ask turtle 0" command. That should be all for the *setup* procedure.

Answer It!

Q05.05: Report your setup procedure to create the initial lights around the rectangle.

Now you need a way to have the lights change color and move around to the left (i.e. counter-clockwise) and you will create those commands in the *go* procedure. Remember that the procedure starts with "to go" and ends with "end".

Let us first modify the *go* procedure to blink the lights from white to yellow and back again. It possible to identify a set of agents (i.e., all the turtles) using the term "turtles". You previously gave commands to a single turtle agent with a command like ***ask turtle 0 [set color white]***. You can give commands to each turtle in the set of all turtles with a command like ***ask turtles [set color yellow]***. Add commands to set all the lights (turtles) to white and then a separate command to set all the lights to yellow.

Test *go* by selecting the *Interface* tab and pushing the *go* button several times. Try slowing the speed slider way down again until you can see what is happening to each light when you push the button. You should see that the turtles are in a random order when they are given commands by the ***ask turtles []*** command. It is a form of repetition control structure. Return the speed slider to normal now.

Now let us further modify the code to move the lights. Add a command to one of your ***ask turtles []*** commands to move your lights forward one distance. Test your code by running it.

Your model should be getting closer to what you want, but the lights are forgetting to turn left at the corners. You need a selection statement to see if a turtle is at one of the corners. If it is, have it make a 90° left turn. Then have it move forward 1. The way to see if a turtle is at the upper right corner (10, 6) would look like this:

```
if xcor = 10 and ycor = 6 [left 90]
```

Add a statement like this for each of the other three corners of the rectangle. Test your changes and see how it looks. Make sure you have also added the command ***tick*** to the end of the *go* procedure. This adds one to the ticks counter on the model to tell how many times the *go* procedure has run. Remember you can start the model over anytime by pushing the *setup* button. Adjust the speed slider until it looks like flickering theater lights. Make sure to save your model.

Answer It!

Q05.06: Give an example of a *sequence* of commands from your theater lights model.

Q05.07: Give an example of a *selection* control structure from your theater lights model.

Q05.08: Give an example of a *repetition* control structure from your theater lights model.

Discuss It!
Your model looks somewhat like flashing theater lights but you know that lights don't really move around the outside of a theater sign. How does a real sign achieve the look of moving lights?

5.8 Theater Lights Part 2

One way to make lights on signs appear to be moving is by turning them off and on in a sequence. We will change the previous model to model this technique. Open your previous theater lights model and use *File* and *Save As*... to make a copy of the model as Lights2.nlogo.

We will first modify the *setup* procedure to leave a turtle on every patch around the border instead of every other one. Make sure the lights create the same size rectangle. Test it and make sure it is working correctly before continuing.

After all the lights are on around the border, we now need to turn off every other light. Turtles have a Boolean condition variable ***hidden?*** which is either ***true*** or ***false***. Boolean variables end with a ***?*** to show that we are asking the question whether the turtle is hidden or not. Right click one of the lights and select the option to inspect whichever turtle you happened to click on. This should bring up the turtle monitor. Look to see that the ***hidden?*** variable has the value ***false***. Change it to ***true***. Did the light disappear? Change the value back to *false* and the light should come back on.

Remember that each turtle has a unique ***who*** number assigned when it is created (kind of like a social security number). Since the turtles are numbered in order during creation, the turtles with even *who* numbers will be every other one.

Even numbers are evenly divisible by 2. That means they have a 0 remainder when divided by 2. For each light that would be the condition ***who mod 2 = 0***. Now add commands to the end of the *setup* procedure to ask all turtles with an even who number to *set hidden? true*. (You may need to search the excellent NetLogo Dictionary online to determine the proper terminology to use.) Test your *setup* procedure until you get it working correctly (i.e. hides the even numbered turtles).

Answer It!
Q05.09: What was the command you used to hide the even numbered lights?

In order to give the illusion of moving lights, you want to keep turning lights on and off. For each light, if it is on, then turn it off. If it is off, then turn it on. One approach would be to use a statement like ***ifelse hidden? [set hidden? false] [set hidden? true]***. Study this code until it is clear.

Another method takes advantage of Boolean algebra operation ***not hidden?*** which gives the value *true* if ***hidden?*** is *false* and gives the value *false* if ***hidden?*** is *true*. This makes it possible to set the value of ***hidden?*** to its opposite. That would look something like ***set hidden? not hidden?***. Now modify the *go* procedure by removing the commands to move the turtles around the rectangle and replace with commands to turn the lights off and on each time *go* is run. Test your model until it works. Don't forget to save your model.

Answer It!

Q05.10: Does it look like the lights are moving around or just flashing on and off?
Q05.11: If the lights appear to be moving, which direction are they moving (clockwise or counter-clockwise)?
Q05.12: How might you improve the model to make it more realistic?

Discuss It!

Which theater lights model version do you think is better? Why? Which on was easier to implement? How much more effort and code do you think would be needed to make it more realistic?

5.9 Leaves on the River Part 1

This NetLogo model was inspired one beautiful day at the park from watching the river flowing by. Assume that it is fall and leaves are falling into the river on the left from where you sit watching. The river is flowing from left to right and you are viewing from above. All leaves in our section of the river will float down river (i.e. to the right) from where they enter on the left until they exit on the right.

Start NetLogo if it isn't already started and be sure to begin a *New* model. Begin by adding the basic, simple commands (see the appendix) and save the model with a good name (e.g. Leaves1.nlogo). Don't forget to periodically save your model as you modify it.

You have seen the expressive power of *procedural abstraction* in enabling us to name a set of commands. As you learn in Module 3, *data abstraction* provides power in naming data types. In the previous theater lights models, you represented lights with turtle agents. You may have found it difficult or at best annoying to always remember to call a light a turtle. Wouldn't it be nice to be able to call a light a *light*?

NetLogo provides data abstraction by having every turtle have variable *breed*. You can use the naming power of *abstraction* to create a new breed of turtle. If you add the statement: ***breed [leaves leaf]*** at the top of your code, this defines a new breed. Now you have a set of agents with the name ***leaves***, with individuals called

leaf. This one brief definition now makes it possible to use a command ***create-leaves 10*** instead of ***create-turtles 10***. You can also give commands to this new breed of turtles with ***ask leaves []*** instead of ***ask turtles []***. It merely makes it easier to read *leaves* and think *leaves* as well as think *leaves* and write *leaves*.

Lets uses this abstraction by creating leafs in the new model. The c*reate-leaves* command expects a number which will be the number of agents to be created. It can also accept a block of commands to give to each *leaf* as it is created. This is the same as creating some leaves and then asking them to all run a block of commands, but a little more convenient. Add these commands to your *setup* procedure:

```
create-leaves 10 [
  set color green
]
```

Now add commands to the *go* procedure to have all the *leaves* move with each *tick* of the simulation. Try running the model.

Abstraction has allowed the naming of procedures and data according to what they do and represent. You have seen the expressiveness of calling a procedure *draw-square* when that is what it does. Also, you can now call a leaf a *leaf*. Wouldn't it be better if in addition to calling a leaf a *leaf*, you could look at a leaf and have it look like a leaf instead of an arrowhead?

Select the *Tools* menu, then select *Turtle Shapes Editor*. There is a library of shapes available for turtles and you can create your own. Scroll down until you see the *leaf* shape. It is already included in the model. Select the *leaf* shape and push the *edit* button. Check the *rotatable* box near the bottom left and push the *OK* button to save the change to the shape. The arrowhead shape is the default shape of turtles. Close the *Turtle Shapes Editor*. Add a command to the *setup* procedure to ***set shape "leaf"*** for each of the *leaves* created. Doesn't that look better? It might be a little confusing with the word ***leaf*** meaning a type of turtle agent and ***"leaf"*** meaning a shape. Remember that the naming of the agent breeds can be anything and you choose them because they mean something to you.

Now when running the model, you should see all your leaves start at the middle and then move out in random directions. They continue to move in straight lines, and when they move past the edges of the world (the black window), they appear on the other side (top to bottom, right to left, etc.). This is not realistic for a river and we need to make some modifications to the placement and movement of our leaves.

Many real world phenomena occur with a particular probability. When observed, these phenomena appear to occur by random chance. Probability and statistics is the mathematical study of these types of occurrences. Mathematical models that represent random outcomes are often used to build what are called *Monte Carlo Simulations*. Much study of games with random chance, such as cards and dice, have contributed to this type of simulation model. You will use some commands to add realistic *randomness* to your model.

As leaves fall, they are usually a random set of colors. Try setting the color of the leaves with ***set color one-of [green yellow brown]***. ***one-of*** randomly selects one of the values in a list enclosed in brackets. Test it to make sure it works.

By default, when the leaves are created they are placed in the center of the world and have a random heading from 0 to 360. When they go f*orward,* they all go in the random direction they are headed. You want them to be randomly placed in the world but all flow "downstream" to the right. We will first place the leaves randomly in the world, then we will make sure they all move to the right.

The leaves are currently all being placed in the center of the world at the origin. We need to set the xcor and ycor of our created leaves to a random coordinate within the world. The commands ***random-pxcor*** and ***random-pycor*** provide a random x and y value. Specifically ***random-pxcor*** returns a uniformly distributed random number in the range of all the patches x coordinates on the world grid, same for the other command for y coordinates. Test your code.

Moving to the right can be accomplished by two different ways: changing the heading of all the leaves to 90° and moving them forward 1 (or more) distance; or by changing all leaves *xcor* variable to be 1 more (which is the patch to the right). Let us use the latter and modify the *go* procedure to ask the leaves to ***set xcor xcor + 1***. Test your code.

Answer It!

Q05.13: Does your model look like leaves moving down a river? List things that are unrealistic and should be improved.

Q05.14: Report your model code.

The model currently just causes the leaves that move off the right side of the world to reappear on the left (at the same vertical location). This is somewhat unrealistic. We could improve the model by randomly changing the vertical placement of the leaf when it wraps to the left side. We would need to use a conditional structure along with a reset of the y coordinate. Note that you need to test if you are going to go off the right side before you move the leaf, as the system will immediately place the leaf on the right side.

Answer It!

Q05.15: Implement code to cause the leaves to reappear on the right side in a different vertical location. Report your model code. (Be sure to save your model.)

Discuss It!
An alternate way to improve the look of the model by having leafs appear on the right differently than where they leave. This can be accomplished by turning off the world wrapping, killing the leaving leafs, and creating a new leaf (even with a new color) on the left side. Try modifying your model accordingly.

5.10 Leaves on the River Part 2

Most computational science models have one or more input variables which are changed within some set of possible parameters. These are the independent variables for the experiment. Your current leaves on the river model currently has 10 leaves on the river. In order to vary the number of leaves, we must add an input control.

Select the *Interface* tab and then choose a *slider* from the drop-down list of controls. Add the slider control to your model interface above the *setup* button. A slider dialog should pop-up. Complete the dialog: Global variable: ***leaves-in***, Minimum: ***1***, Increment: ***1***, Maximum: ***30***, initial Value: ***15***, and Units: ***leaves***. Then push the *OK* button to save the slider control. This has defined a new global variable names ***leaves-in*** we can refer to in our code. You can change the slider input parameters anytime by right clicking on the slider and selecting *Edit*.

Next, we need to connect the slider control to the procedures code. Change the leaf creation command by replacing the ***create-leaves 10*** with ***create-leaves leaves-in***. This now creates the number of leaves as indicated on our slider at the start of the model. Run the model with different initial leaves. Change the slider to allow 100 leaves in the river.

Sometimes we want to monitor and record information from our model/simulation as it is running, often with a graph. Let us add a graph to our NetLogo model and have it report the average vertical position of our leaves. Add a plot "control" and fill it out to be like Fig. 5.2. We will alter are code to store the current average y coordinate value in the variable called ***leaf-average-y***. After clicking OK to close the dialog box, right-click the plot control and choose select, then resize the plot to an appropriate size. Again right-click and unselect the control.

To calculate the average y position, we first need to add the leaf-average-y variable to our model by using the declaration ***globals [leaf-average-y]*** at the beginning of our code. We then need to calculate the value of the variable during each *go* iteration. The average is just the sum of the leaf y locations divided by the count of the leaves. We can therefore add the following at the end of the *go* procedure (just before the *tick* command).

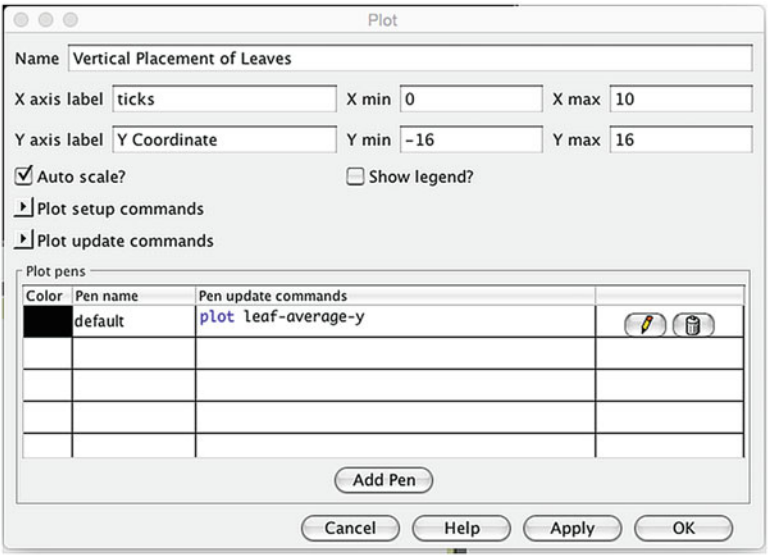

Fig. 5.2 Plot control dialog box

```
set leaf-average-y 0
ask leaves [
 set leaf-average-y leaf-average-y + ycor
]
set leaf-average-y leaf-average-y / count leaves
```

Answer It!

Q05.16: Run the model for at least 200 ticks and screen capture your plot.

Discuss It!

There are many possible improvements to the leaves on a river model. How might you implement an ability to change the number of leaves in the river during the run? With user intervention? Randomly? How might you expand the possible colors of the leaves? How might you make the "water" blue?

5.11 Related Modules

- Module 1: Introduction to Computational Science.
- Module 2: Types of Visualization and Modeling.
- Module 9: Procedures: Performance and Complexity.

Acknowledgement The original version of this module was developed by Dr. Larry Vail.

References

Böhm C, Jacopini G (2011) Structured program theorem. http://en.wikipedia.org/wiki/Structured_program_theorem. Retrieved July 2011

Wilensky U (1999) NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston