we want to answer with the model serves as a filter: all those aspects of the real system considered irrelevant or insufficiently important *for answering this question* are filtered out. They are ignored in the model, or represented only in a very simplified way.

Let us consider a simple, but not trivial, example: Did you ever search for mushrooms in a forest? Did you ask yourself what the best search strategy might be? If you are a mushroom expert, you would know how to recognize good mushroom habitat, but let us assume you are a neophyte. And even the mushroom expert needs a smaller-scale search strategy because mushrooms are so hard to see—you often almost step on them before seeing them.

You might think of several intuitive strategies, such as scanning an area in wide sweeps but, upon finding a mushroom, turning to smaller-scale sweeps because you know that mushrooms occur in clusters. But what does "large" and "small" and "sweeps" mean, and how long should you search in smaller sweeps until you turn back to larger ones?

Many animal species face similar problems, so it is likely that evolution has equipped them with good adaptive search strategies. (The same is likely true of human organizations searching for prizes such as profit and peace with neighbors.) Albatross, for example, behave like mushroom hunters: they alternate more or less linear long-distance movements with small-scale searching (figure 1.1).

The common feature of the mushroom hunter and the albatross is that their sensing radius is limited—they can only detect what they seek when they are close to it—so they must move. And, often the items searched for are not distributed randomly or regularly but in clusters, so search behavior should be adaptive: it should change once an item is found.

Why would we want to develop a model of this problem? Because even for this simple problem we are not able to develop quantitative mental models. Intuitively we find a search strategy which works quite well, but then we see others who use different strategies and find more mushrooms. Are they just luckier, or are their strategies better?

Now we understand that we need a clearly formulated purpose before we can formulate a model. Imagine that someone simply asked you: "Please, model mushroom hunting in the

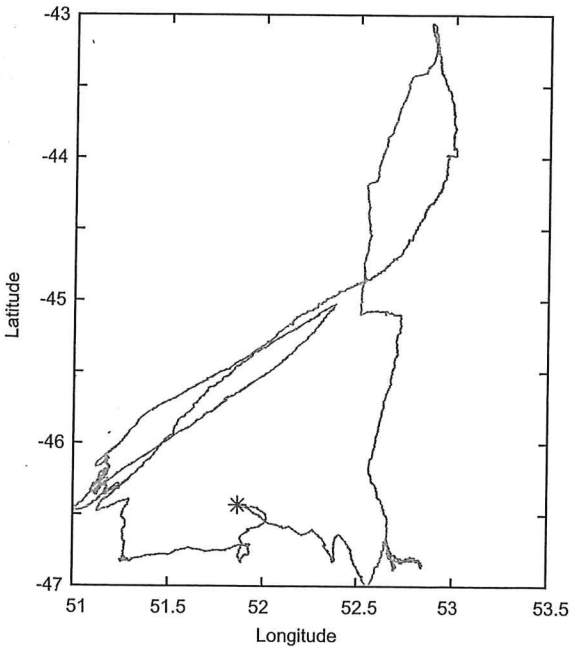

Figure 1.1
Flight path of a female wandering albatross (*Diomedea exulans*) feeding in the southern Indian Ocean. The flight begins and ends at a breeding colony (indicated by the star) in the Crozet Islands. Data recorded by H. Weimerskirch and colleagues for studies of adaptive search behavior in albatross (e.g., Weimerskirch et al. 2007).

forest." What should you focus on? On different mushroom species, different forests, identification of good and bad habitats, effects of hunting on mushroom populations, etc.? However, with the purpose "What search strategy maximizes the number of mushrooms found in a certain time?" we know that

- We can ignore trees and vegetation; we only need to take into account that mushrooms are distributed in clusters. Also, we can ignore any other heterogeneity in the forest, such as topography or soil type—they might affect searching a little, but not enough to affect the general answer to our question.
- It will be sufficient to represent the mushroom hunter in a very simplified way: just a moving "point" that has a certain sensing radius and keeps track of how many mushrooms it has found and perhaps how long it has been since it found the last one.

So, now we can formulate a model that includes clusters of items and an individual "agent" that searches for the items in the model world. If it finds a search item, it switches to smaller-scale movement, but if the time since it found the last item exceeds a threshold, it switches back to more straight movement to increase its chance of detecting another cluster of items. If we assume that the ability to detect items does not change with movement speed, we can even ignore speed.

Figure 1.2 shows an example run of such a model, our simple Mushroom Hunt model. In chapter 2 you will start learning NetLogo, the software platform we use in this book, by programming this little model.

This searching problem is so simple that we have good idea of what processes and behaviors are important for modeling it. But how in general can we know whether certain factors are important with regard to the question addressed with a model? The answer is: we can't! That is, exactly, why we have to formulate, implement (program in the computer), and analyze a model: because then we can use mathematics and computer logic to rigorously explore the consequences of our simplifying assumptions.

Our first formulation of a model must be based on our preliminary understanding of how the system works, what the important elements and processes are, and so on. These preliminary ideas might be based on empirical knowledge of the system's behavior, on earlier models addressing similar questions, on theory, or just on . . . imagination (as in the mushroom

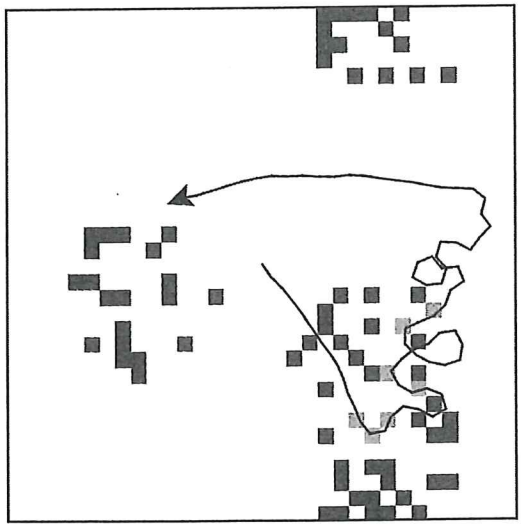

Figure 1.2
Path of a model agent searching for items that are distributed in clusters.

How do you know what context you are in, when writing code? If you are writing a procedure that the observer executes (e.g., `go` and `setup`; any procedure executed by a button on the Interface), then you are in observer context. If you are writing a procedure executed by turtles (or patches), you are in turtle (or patch) context. But—the context can change within a procedure because primitives such as `ask`, `create-turtles`, and `hatch` begin new contexts within their brackets that contain code for other agents to execute. If the `go` procedure includes the statement `ask turtles [move]` then the procedure `move` is in turtle context. See the NetLogo Programming Guide on `ask` for more explanation.

Similarly, agents can only directly access the variables defined for their own type; patches cannot do calculations using a variable defined for turtles, the observer cannot use patch variables, and so on (but a patch can obtain the value of a turtle's variable using the primitive `of`). There are two very important exceptions to this restriction: observer variables are global and can be used by all agents, and turtles automatically can use the variables of the patch they are currently on.

- To understand agents, variables, and commands and reporters, read the "Agents," "Procedures," and "Variables" sections of the Programming Guide carefully.

At this point you probably feel confused and overwhelmed, but don't panic! All the material thrown at you here will be repeated, reinforced, and explained in much greater detail as we move through parts I and II of this course. Perhaps slowly at first, but then rapidly, NetLogo will become intuitive and easy. For now, just follow along and learn by doing, teaching yourself further by continually referring to the NetLogo documentation.

2.3 A Demonstration Program: Mushroom Hunt

Now we are ready to program our first model, the mushroom hunting model introduced in section 1.2. We will describe the model as we go.

- Create, via "File/New" in the NetLogo menu, a new NetLogo program. Save it, via "File/Save as", in a directory of your choice and with the file name "MushroomHunt.nlogo".

- Click the Settings button.

From the Interface tab, click the Settings button to open the Model Settings dialog, where you can check the World's geometry. For now we will use the default geometry with the origin (patch 0,0) at the center of the World, and both the maximum x-coordinate (`max-pxcor`) and maximum y-coordinate (`max-pycor`) set to 16. This makes a square of 33 × 33 patches. Even though we use the default geometry now, it is an important habit to always check these settings. Click OK to close the Model Settings dialog.

For Mushroom Hunt, we want to create a World of black patches, with a few clusters of red patches (representing mushrooms) sprinkled in it; create two turtles—our hunters; and then let the hunters search for the red patches. We therefore need to program how to initialize (create) the World and the hunters, and then the actions the hunters perform when the model runs.

We always use the NetLogo convention of using the name `setup` for the procedure that initializes the World and turtles, and the name `go` for the procedure that contains the actions to be performed repeatedly as the model runs. By now you know from reading the Programming

Guide that a procedure starts and ends with the keywords `to` and `end`, and contains code for one discrete part of a model.

To program the `setup` and `go` procedures, we first create buttons on the Interface tab that will be used to start these procedures.

- On the Interface tab, there is a drop-down menu, usually labeled "Button," that lets you select one of the Interface elements provided by NetLogo (Button, Slider, Switch, etc.). Click on this selector, which opens a list of these elements, then on "Button." Place the mouse cursor in the blank white part of the Interface, left of the black World window, and click. This puts a new button on the Interface and opens a window that lets you put in the characteristics of this new button (figure 2.1).

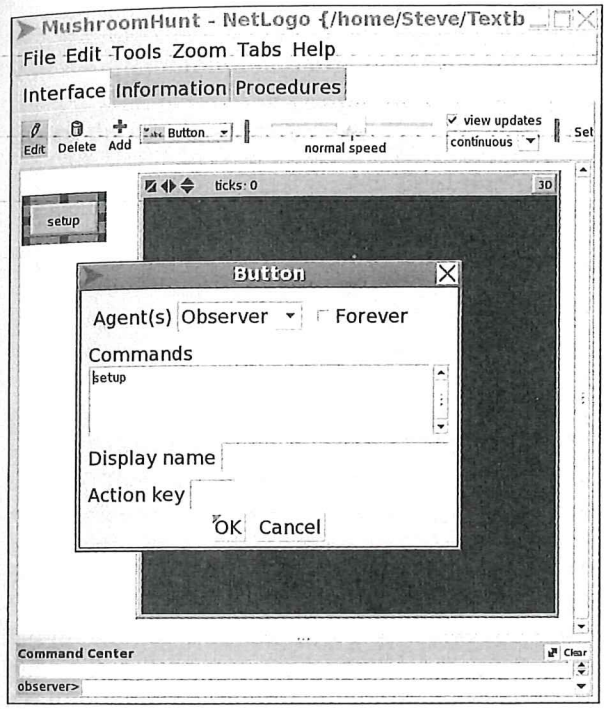

Figure 2.1
Adding a button to the Interface.

- In the "Commands" field, type "setup" and click OK.

Now we have created a button that executes a procedure called `setup`. The button's label is in red, which indicates that there is a problem: no corresponding command or procedure named `setup` exists yet on the Procedures tab. We will fix that in a minute.

- Create a second button, assign it the command `go`, click the "forever" checkbox in the Button window, and click OK.

This second button, `go`, now has a pair of circular arrows that indicate it is a "forever" button. "Forever" means that when the button is clicked, it will execute its procedure over and over until the button is clicked again.

Now, let us write the `setup` procedure.

- Click on the Procedures tab, where you will find a blank white space to enter your code. To start (figure 2.2), type:

```
to setup
end
```

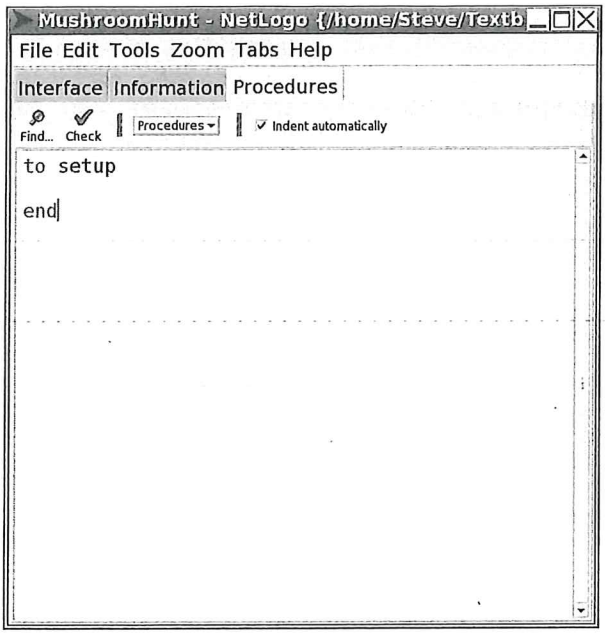

Figure 2.2
Starting to write code in the Procedures tab.

- Click the Check button.

This button runs NetLogo's "syntax checker," which looks for mistakes in the format of your code: are there missing brackets, do primitives have the right number of inputs, did you call a procedure that does not exist, etc.? Using the syntax checker very often—after every statement you write—is essential for writing code quickly: you find mistakes immediately and know exactly where they are. However, this checker does not find all mistakes; we will discuss it and other ways to find mistakes in chapter 6.

Now, there should be no error message. If you switch back to the Interface tab, the label of the `setup` button is now black, indicating that it now finds a `setup` procedure to execute—even though this procedure is still empty and does nothing.

- Go back to the Procedures tab and delete the word `end`. Click the Check button again. You will obtain an error message, saying that a procedure does not end with `end`.

- Undo your deletion of `end` by typing CTRL-Z several times.

Programming note: Shortcut keys for editing

For editing code, the following control key (or command key, on Apple computers) combinations (invented in the stone age of computing) are still essential: CTRL-C (copy), CTRL-V (paste), CTRL-X (delete and copy to clipboard), CTRL-F (find or replace), CTRL-Z (undo), and CTRL-Y (redo).

(These programming notes appear throughout this book to provide helpful general advice. There is a separate index of them at the back of the book.)

- Just as we created the `setup` procedure (with two lines saying `to go` and `end`), write the "skeleton" of the `go` procedure.

Now, we will tell the `setup` procedure to change the color of the patches.

- Change the `setup` procedure to say

```
to setup
  ask patches
  [
      set pcolor red
  ]
end
```

The `ask` primitive is the most important and powerful primitive of NetLogo. It makes the selected agents (here, all patches) perform all the actions specified within the brackets that follow it. These square brackets always delineate a block of actions that are executed together; you should read `[` as "begin block" and `]` as "end block." (Notice that NetLogo automatically indents your code so it is easy to see which code is within a set of brackets or the beginning and end of a procedure.)

The statements we just wrote "ask" all patches to use a new value of their built-in color variable `pcolor`. Patches by default have `pcolor` set to black (which is why the World looks like a big black square), but in our new `setup` procedure we use the primitive `set` to change it to red. The primitive `set` is called an "assignment" operator: it assigns a value (red) to a variable (`pcolor`). (In many programming languages, this statement would be written `pcolor = red`, which absolutely does not work in NetLogo!)

- Test the new `setup` procedure by going to the Interface tab and clicking the `setup` button. All the patches should turn red.

However, we only want only a few clusters of patches to turn red. Let's have NetLogo select four random patches, then ask those four patches to turn 20 nearby patches red. Modify the `setup` procedure to be like this:

```
ask n-of 4 patches
[
  ask n-of 20 patches in-radius 5
```

```
    [
      set pcolor red
    ]
  ]
```

The primitives `n-of` and `in-radius` are new to us.

- Look up `n-of` and `in-radius` in the NetLogo Dictionary. You can find them by going to the dictionary and clicking on the "Agentset" category (a good place to look for primitives that seem to deal with groups of agents). But—an important trick—you can go straight to the definition of any primitive from your code by clicking within the word and pressing F1.

The dictionary explains that `n-of` reports a random subset of a group of agents, and `in-radius` reports all the agents (here, patches) within a specified radius (here, five patch-widths). Thus, our new setup code identifies four random patches, and asks each of them to randomly identify 20 other patches within a radius of 5, which are then turned red.

- Go to the Interface tab and press the `setup` button.

If all your patches were already red, nothing happens! Is there is mistake? Not really, because you only told the program to turn some patches red—but all patches were already red, so you see no change. To prevent this kind of problem, let's make sure that the model world is always reset to a blank and default state before we start initializing things. This is done by the primitive `clear-all` (or, abbreviated, `ca`), which you should add as the second line of the `setup` procedure:

```
to setup
  ca
  ask n-of 4 patches
  [
    ask n-of 20 patches in-radius 5
     [
       set pcolor red
     ]
  ]
end
```

Programming note: `Clear-all`

Forgetting to put `ca` (`clear-all`) at the beginning of the `setup` procedure is a common error for beginners (and, still, one author of this book). Almost always, we need to begin `setup`'s initialization process by erasing everything left over from the last time we ran the model. As you build and use a NetLogo program, you hit the `setup` and `go` buttons over and over again. `Clear-all` removes all turtles and links, sets patch variables to their default values, and clears any graphs on your Interface. If you forget `ca` in `setup`, there will be junk (agents, colors, graph data) left from your previous run mixed in with the new stuff that `setup` creates. Learn to make `ca` automatically next after `to setup`.

If you click the `setup` button now, you will see that a World consisting of black and red patches appears. If you click the `setup` button several times, you see that four random clusters of red patches are indeed created.

Sometimes it looks like fewer, larger clusters are created, because clusters overlap. Other times, it looks like part of a cluster is created at the edge of the World; to understand why, go to the NetLogo Interface Guide section on the Views and read about "world wrapping," and see the Programming Guide's section called "Topology." Understanding NetLogo's world wrapping is extremely important!

- From the main NetLogo menu, hit "File/Save." Save your work often! NetLogo will not recover your unsaved work if something bad happens. (And, from here on, your code should match the full program we provide at the end of this section; use that full program we provide if you get confused.)

Now let's make the number of clusters of red patches a *model parameter*. In our models and programs, we usually try not to write numbers that control important things directly into the deep code—which is called "hardwiring" them. Instead, we make them parameters that are easy to find and change. To do this, we need to create a global variable named `num-clusters` and give it a value of 4, then use `num-clusters` in the `ask` statement that creates the clusters. You can give a variable a value by using the primitive `set`, but first you need to define that variable (as you learned from the Variables section of the Programming Guide).

- In the Procedure tab, go the top of all your code and insert

```
globals
[
  num-clusters
]
```

- In the `setup` procedure, add this near the beginning, just after `clear-all`:

```
set num-clusters 4
```

and replace `4` in the `ask n-of 4 patches` statement with `num-clusters`.

Now, to change the number of clusters, we know exactly where to find and change the parameter's value, without digging through the program. (Later, we will use *sliders* to change parameters from the Interface.)

Now that we have set up the world of our Mushroom Hunt program, we need to create the agents, in our case two hunters. This is done by the primitive `create-turtles`, or `crt` for short. The statement `crt 2 []` creates two turtles and then makes them execute the code within the brackets.

- Add the following code at the end of the `setup` procedure. Click Check to look for mistakes, then try it using the `setup` button on Interface:

```
crt 2
[
  set size 2
```

```
  set color yellow
]
```

`Size` and `color` are built-in turtle variables, so we can set them without defining them. If you click the `setup` button several times, you see that the hunters are placed, by default, in the center of the World, but their heading varies randomly.

Figure 2.3 shows what the Interface should look like.

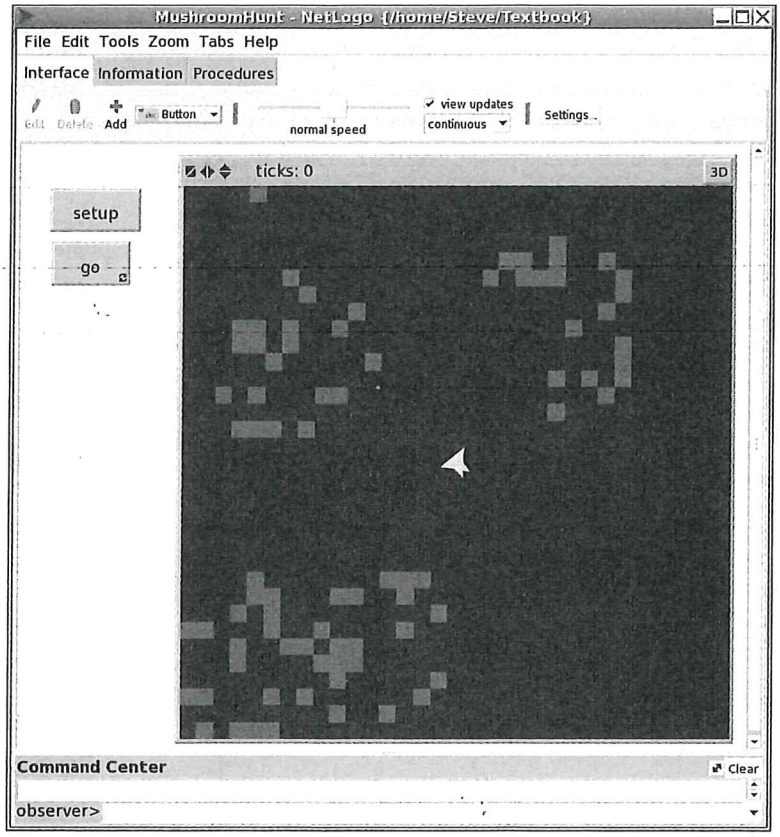

Figure 2.3
Mushroom Hunt interface with the model initialized.

Now we are going to program the `go` procedure. The `go` procedure defines a model's schedule: the processes (procedures) that will be performed over and over again, and their order. To keep the `go` procedure simple and easy to understand—which is very important because it is the heart of the program—we almost never program actual processes or actions within it. Instead, `go` simply calls other procedures where each action is programmed.

In Mushroom Hunt, we have only one action, the hunter's search. Thus, the `go` procedure needs to include only one statement, which tells all the turtles (our two hunters) to execute a turtle procedure where searching is programmed:

```
ask turtles [search]
```

For this to work, of course, we need to write a procedure called `search` for the hunters to execute.

■ Add the above `ask turtles` statement to the `go` procedure. Then create an empty procedure so you can check the code so far:

```
to search
end
```

Now let's program how the hunters search. Each time `search` executes, we want the hunters to turn, either a little bit if they have found nothing recently (e.g., in the past 20 patches searched), or sharply to keep searching the same area if they did find a mushroom recently. Then they should step ahead one patch.

■ Add this code to the `search` procedure, looking up any parts you do not understand in the NetLogo Dictionary:

```
ifelse time-since-last-found <= 20
  [right (random 181) - 90]
  [right (random 21) - 10]

forward 1
```

Can you figure out from the NetLogo Dictionary's entry for `random` how the statement `right (random 181) - 90` causes a turtle to turn a random angle between -90 and +90 degrees?

Note how we use the `ifelse` primitive (or, sometimes, the related `if` and `ifelse-value`) to model decisions. If the *boolean condition* following `ifelse` is true, then the code in the first set of brackets is executed; if false, the code in the second set of brackets is executed. A boolean condition is a statement that is either true or false. Here, the boolean condition `time-since-last-found <= 20` consists of a comparison: it is true if the value of `time-since-last-found` is less than or equal to 20.

But when you now hit the Check button, there is a problem: what is `time-since-last-found`? We want it to be a variable that records how long it has been since each hunter found a mushroom. That means three things. First, each hunter must have its own unique value of this variable, so it must be a turtle variable. We need to define `time-since-last-found` as a turtle variable.

■ Just after the `globals` statement at the start of your program, add

```
turtles-own
[
  time-since-last-found
]
```

Second, we need to set the initial value of this variable when we create the hunters. We will set it assuming the hunters have not found a mushroom yet, by using a value larger than 20.

■ In the `setup` procedure, change the `create-turtles` statement to this:

```
crt 2
[
  set size 2
```

```
    set color yellow
    set time-since-last-found 999
  ]
```

Finally, the hunters must update `time-since-last-found` each step. If they find a mushroom, they need to reset `time-since-last-found` to zero (and pick the mushroom!); otherwise, they need to add one to `time-since-last-found`. We define finding a mushroom as landing on a red patch.

■ Add these statements to the end of the `search` procedure:

```
ifelse pcolor = red
  [
    set time-since-last-found 0
    set pcolor yellow
  ]
  [
    set time-since-last-found time-since-last-found + 1
  ]
```

Note that these statements only work because NetLogo lets turtles read and change variables of the patch they are on; here, the turtles use `pcolor`, their patch's color. Also note that to add one to the value of `time-since-last-found`, we had to use `set` to assign its new value to its old value plus one. (Be aware that in NetLogo you must use spaces around arithmetic operators like "+"; otherwise NetLogo thinks they are part of the variable name.)

Make sure you understand that patch coordinates are discrete variables: they only have integer values, in units of patch-widths. In contrast, turtle coordinates are continuous variables: they can take any value within a patch, for example 13.11 units. It is therefore important to distinguish between primitives that refer to turtle coordinates and those that refer to patch coordinates. For example, `move-to` moves a turtle to the center of a patch, but `forward` can place a turtle anywhere.

If you now test the program by clicking the `go` button, you may not be able to see very much because the hunters move too fast. If so, adjust the execution speed controller on the Interface. (Click `go` a second time to stop execution.)

Voila! Here we have our first complete NetLogo program! (Make sure you save it!) Despite its simplicity, the program contains the most important elements of any NetLogo program. But before we stop, let us look at three more important NetLogo tools: Agent Monitors, the command center, and `ticks`.

Agent Monitors are tools to see, and change, the variables of an agent (turtle, patch, or link).

■ Move the cursor over one of the hunters and right-click. A panel appears that, at the end, lists "turtle 0"; move the cursor to this entry and select "inspect turtle 0."

An Agent Monitor opens that includes two panels: a zoomed view on the inspected turtle and its environment, and a list of all variables of the turtle: its position, color, heading, etc. The turtle variable we created, `time-since-last-found`, is also listed. (Both panels can be closed and reopened by clicking the small black triangle at their upper left corner.) You can restart the model by clicking `go` and observe how the variables change.

Not only can you see an agent's variables with an Agent Monitor; you can also change their value whenever you want, just by entering a new value in the monitor.

- Stop the program and change the turtle's size to 5, and its `time-since-last-found` to -99. (Your changes become active when you hit the Enter key or move the cursor out of the dialog box where you enter them.)

There are more things you can do with Agent Monitors, including giving commands directly to the selected agent by typing in the same kinds of NetLogo statements you would use in your code. Read the "Agent Monitors" section of the User Manual's Interface Guide.

The Command Center appears at the bottom of the Interface tab. Here, you can send commands to the observer or to all patches, turtles, or links, and get the results of output primitives such as `show`. Read the "Command Center" section of the User Manual and try some commands such as telling the patches to `set pcolor blue`.

The ability to type in commands in Agent Monitors and the Command Center may not seem useful at first, but as you become a better NetLogo programmer you will find these tools extremely helpful. As a fun example, use the Command Center to do the following.

- After setting up the Mushroom Hunt program and letting it run for a second or two, stop it. Click on the text "observer>" in the Command Center. From the pop-up list, select "turtles>" so you can now issue a command to all turtles. In the adjacent window, enter `hatch 1 [right 160]`, which tells each hunter to create a second hunter, which then turns 160 degrees (figure 2.4).

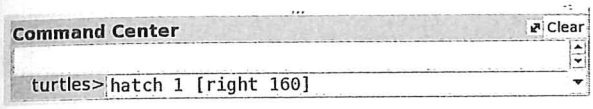

Figure 2.4
Giving a command to all turtles via the Command Center.

- Run the program and see how the now four hunters search for red patches. You might repeat the `hatch` command to create even more hunters: just place the cursor in the command line and hit the Up arrow to retrieve the previous command, then Enter.

- After hatching some more turtles, have the Command Center tell you how many there are. Select the observer to send commands to, and enter `show count turtles`.

So far, our program does not deal with time explicitly: we do not keep track of how many times the `go` procedure has executed, so we cannot determine how much time has been simulated if we assume, for example, that each time the mushroom hunters move represents one minute. To track time, we call the primitive `tick` every time the `go` procedure is called. Read the Programming Guide section "Tick Counter"; then:

- At the end of the `setup` procedure, insert a line with only the statement `reset-ticks`. At the beginning of the `go` procedure, insert a line with only the statement `tick`.

If you now click `setup` and then `go`, you can follow the tick counter at the top of the World display; it shows how many times the `go` procedure has executed. (In previous versions of NetLogo, the `clear-all` statement reset the tick counter to zero, but now we need to do it by putting the `reset-ticks` statement at the end of `setup`.)