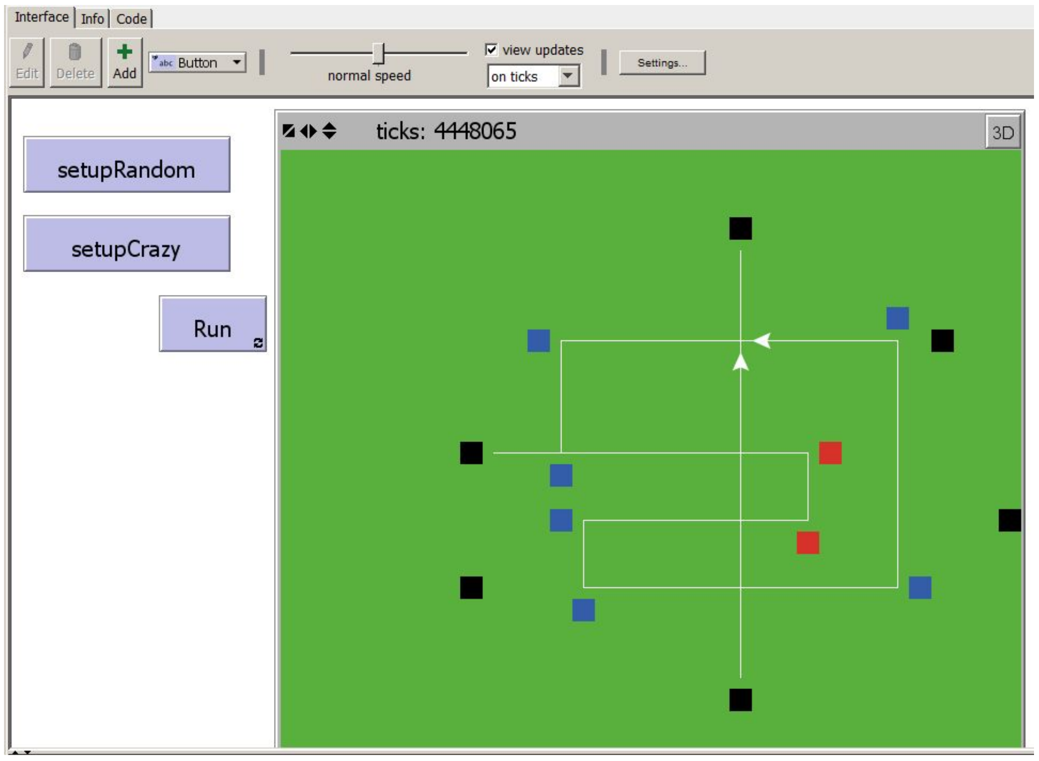
CS108L Computer Science for All
Module 4 Guide
Bumper Turtles



The Bumper Turtles model created in this lab requires the use of ***Boolean logic*** and ***conditional control flow***. The basic rules are:

1. Each turtle starts in the middle of a patch. This could be any patch, but not part in one patch and part in another.
2. Every tick, each turtle "looks" ahead one patch along its current heading.
    a. If the patch ahead is black then the turtle makes a U-Turn.
    b. If the patch ahead is blue, then the turtle makes a 90° left turn.
    c. If the patch ahead is red, then the turtle makes a 90° right turn.
    d. If the patch ahead is beautiful green grass, then there are two cases to deal with: If there is another turtle in that patch, then the turtle makes a U-Turn, otherwise, the turtle runs one step forward in the turf.

Note: the turtle should ONLY move when the patch ahead is green. This is because after the turtle turns, there might be another block in its new forward direction.

Your turtles' tracks can be as simple or as complex as you'd like, with the following requirements:

1. You have at least 2 turtles, which must start at the same unique locations each time.
2. There are at least 10 black, red or blue patches that affect turtle behavior.
3. There is at least one patch where two different turtle paths cross.
4. There is at least one large grouping of patches that is at least 3x3 patches made using the **if** conditional and the **AND** keyword. This group counts as 1 of the 10 in 2 above and maybe any color.
5. Your turtles must stay on the tracks you create. These tracks can change, but when a turtle turns and heads in a new direction, there must be another turtle or a patch that will push it back on track. Your turtles shouldn't wrap around the world endlessly.

World Settings

Use the following settings for the interface:

- min-pxcor = -16, max-pxcor = 16, min-pycor = -16, max-pycor = 16

Setup Button

Your program has a setup button that when pressed must:

1. Clear the 2D world view.
2. Remove all the turtles (clear-all does both 1 and 2).
3. Set all patches to green. Then set specific, hardcoded patches to black, blue or red. You choose which specific patches to set black, blue or red so that the turtles you create in step (4) follow run around on a cool, creative, crisscrossing track. The screen capture on the first page shows an example of such a setup after the turtles have run around a bit. You may create more turtles and more interesting paths then that shown. Hint: Draw your pattern on graph paper before coding it.
4. Create at least 2 turtles each with a specific location and heading so that someplace along the path or that will enter the path created in step (3). Make sure the pen is down.

The example image has two setup buttons: setupRandom and setupCrazy. You only need one Setup button, and it should not create turtles or patches in random locations. However, if you would like to experiment with other setups, you can!

Hint: Setting a Particular Patch to a Specified Color
The NetLogo command: **ask patch 2 -4 [set pcolor blue]**

will set the color of the patch with coordinates (2, -4) to blue

Hint: Setting a Particular Turtle to a Location and Heading

Each turtle in NetLogo has a unique identification number. The identification numbers always start with 0 and count up to one less than the total number of turtles. The NetLogo **who** command reports a turtle's identification number.

These facts can be used to set properties of a particular turtle. For example:

```
 1)     create-turtles 2
 2)     [
 3)       if (who = 0)
 4)       [
 5)         setxy -3 0     ;; Lines 5 & 6 only execute for the turtle with ID number = 0
 6)         set heading 180
 7)       ]
 8)       if (who = 1)
 9)       [
10)         setxy 4 0     ;; Lines 10 & 11 only execute for the turtle with ID number = 1.
11)         set heading 0
12)       ]
13)     ]
```

Hint: Getting the Color of the Patch Ahead

See "Bumper Turtles" video for more details

We have seen before that within a turtle context, **pcolor** is the color of the patch the turtle is on. In this lab, we need to look at the color of the patch one *ahead* of the current turtle location. This can be done with the **patch-ahead** function as shown below:

```
ask turtles
[
  let colorOfPatchAhead green
  ask patch-ahead 1    ;;1 is the number of patches ahead to look
  [
    set colorOfPatchAhead pcolor    ;; stores color of the patch so you
                                    ;; can use it in Boolean expressions
  ]

  if (colorOfPatchAhead = blue)
  [
    ;; In this code block, code what you want to happen when
    ;; the color or the patch ahead is blue.
  ]
  ;; also write an if statement for each of the other possible
  ;; colors
]
```

Hint: Finding Out Whether there is a Turtle Ahead See "Bumper Turtles" video for more details
To determine whether there is another turtle on the patch ahead of the current turtle, we build a triple compound command. Within a turtle context, **patch-ahead 1**, reports the patch that is one patch ahead of the turtle's current location. The NetLogo function, **turtles-on *patch***, reports the set of all turtles that are on ***patch***. The Netlogo function, **any? *agentset***, reports true if ***agentset*** contains at least one agent. Otherwise, it reports false. Putting this all together, the statement: **any? turtles-on patch-ahead 1** Reports true if and only if there is at least one turtle on the patch that is one ahead of the current turtle's location. Since this function reports a Boolean (true or false) it can be used in an **if** or in an **ifelse** statement.

Understanding **OR** and **AND** keywords is critical to using conditional statements to their fullest. These keywords allow you to 'string' conditions together in a logical way that enables multiple conditions to be tested in an **if** statement. As with all if statements, the result of the condition must be true or false.

So in the statements shown below, x is set to 0 so the condition is true (x = 0) and the code between the brackets is executed. Had x been set to a number other than 0 the statement would be false and the code would not be executed.

```
let x 0
if (x = 0 ) [
 ;Do something
]
```

But what if we wanted more than 1 condition to result in the if statement executing? In that case we might use the **OR** keyword between the two statements. So if we wanted to execute when x is either 0 or 1 then we could do this;

```
let x 0
if (x = 0 or x = 1) [
 ;Do something
]
```

Now if x = 0 or if x = 1 we execute what is between the brackets. In the case of the **OR** keyword if any statement is true then we execute what is between the brackets (note that you must write the full condition, you cannot write if x = 0 or 1 for instance. Note that you don't have to limit yourself to just two conditions write as many conditions as you need and they don't have to depend on the same variable, like below.

```
let x 2
let y 1
if (x = 0 or x = 1 or y = 1) [
 ;Do something
]
```

Here x = 2 so it does not meet the condition, but y = 1 which does meet the condition since with **OR** only 1 of the list of conditions needs to be true, the code in the brackets executes.

The **AND** keyword works similarly to the **OR** keyword except that all conditions must be true for the code in the brackets to execute.

The replacing the **OR** with an **AND** in the two condition statement we make it to where the code between the blocks can never be executed! This is something to watch out for carefully.

```
let x 1
if (x = 0 and x = 1) [
 ;Do something
]
```

This is because x may have a value of 1 or 0, it cannot be both thus at least one of the conditions is always false. Since all conditions associated with the **AND** must be true, the code within the brackets will never execute.

We could use the **AND** keyword like this though;

```
let x 0
let y 1
if (x = 0 and y = 1) [
 ;Do something
]
```

In this case, since x is 0 and y is 1, then the code in the brackets executes. If either of these variables are something else then the code within the brackets will not execute.

Lastly you may combine the keywords in any fashion that produces a final true or false result. It is important to remember when doing this that NetLogo reads left to right and does those calculations first then moves to the next item in the list. It is often much easier to use a set of parentheses to indicate which set of values to do first, just like you would do in a math equation.

```
if (x = 0 and y = 1 or x = 1)
if ((x = 0 and y = 1) or x = 1)
```

So both of the above equations are equivalent to each other. We first look at the terms on the left and evaluate it (or in parentheses). If x is 0 and y is 1 then it is true, then we can read the statement as true or x = 1. Since x cannot be both 0 and 1 the second statement is false in this case. Thus the statement is true or false. Since or only requires one of them to be true, the statement is considered true and code within brackets will then execute.

Likewise if x = 1 it would be false or true, so again the code would execute. But if x was anything else it would not execute. Likewise if x was 0 but y was not 1, it would not execute.

By moving the parentheses, the statement changes though.

```
if (x = 0 and (y = 1 or x = 1))
```

For this one if x is set to 0 and y to 1 then we evaluate within the parentheses first and find y = 1 is true and x = 1 is false so the statement is 'true or false' which evaluates to true since it is an **OR** keyword. This leaves us with x = 0 and true, since x is set to 0 this evaluates 'true and true' which meets the **AND** keyword requirement for true so the statement is true and the statements in the brackets will execute.

However, unlike the example above with the parentheses switched, if x = 1 this can never execute the code in the brackets regardless of what y is set too. This is because x = 0 will always be false when x = 1 and with the **AND** keyword that means you always have 'false and ….' which is false. Thus the code within the brackets only executes when x = 0 and y = 1.

Always make sure you statements evaluate to the correct true/false condition for what you are trying to do. Also keep in mind these are simple examples, you could for instance use the > (greater than) or < (less than) symbols instead of the equal (=) symbol and a larger range of numbers and operations are available to you.

Like bounding a variable. Below we execute the code if the value stored in x is bigger than 5 but less than 10. You can use variations of this to color code your patches instead of specifying each patch (though typically you must also bound y as well and use pxcor/pycor instead of x/y).

```
if (x > 5 and x < 10) [
  ;Do something
]
```