Introduction to Computer Science with MakeCode for Minecraft

Lesson 7: Functions

Often in programming, there are tasks or procedures that are used frequently within the same program. Rather than rewrite the lines of code that perform a particular task each time you need it, you can group that set of instructions together as a function. Grouping frequently used instructions as a function makes your code more efficient. You can write the set of instructions once as a function and from then on simply 'call' the function from inside your program whenever you need that task done. A function is usually given a name that describes the task it will perform when called, making your code easier to read, too!

Example: Writing Your Name
Consider this example: As a student you are probably asked several times a day to write your name, on a paper, or a form, or on a sign-up sheet. Writing your name is a function that you do without thinking about it very much, yet the task is actually made up of a number of smaller steps. Let's take a look at what those steps would be for a person named 'Mary'.
The function could be called 'WriteMyName'.

The list of steps or lines of code performed when WriteMyName is called for Mary would be:

- Write a capital 'M'.
- Just to the right of the previous letter, write a lowercase 'a'.
- Just to the right of the previous letter, write a lowercase 'r'.
- Just to the right of the previous letter, write a lowercase 'y'.

Whenever Mary needs to write her name, she calls the function 'WriteMyName' and that set of instructions is carried out.

Note that the lines of pseudocode in this function have within them even smaller functions. When you were learning to write your name, you needed to be able to perform the set of instructions for writing each letter first!

WriteCapitalM, WriteLowercaseA, WriteLowercaseR, WriteLowercaseY are themselves functions that need to be called in order to complete the task. What sets of instructions make up each of these smaller functions?

Unplugged Activity: PB & J

Here's another example of how functions might be used to define a complex task. Imagine that a parent makes you lunch every day – wouldn't it be nice to delegate this task to a computer? Ask students to imagine a robot that would make them a peanut butter and jelly sandwich (PB&J) every day. What are the steps involved in making a peanut butter and jelly sandwich? Can they write a function that includes all the steps necessary to make a yummy sandwich? This is a fun activity to do in the classroom.

Show the students, but do not comment on the materials that you will be using for this activity.

- Materials:
  - A jar of peanut butter** (closed)
  - A jar of jelly (closed)
  - A bag of sliced bread (closed)
  - A plate
  - A knife
  - A paper towel

**Note: Check for allergies to peanuts. You can always substitute butter or cream cheese for the peanut butter.

Write pseudocode for the making of a PB&J sandwich:

- Have students work in pairs to write pseudocode (English language instructions) for making a peanut butter and jelly sandwich

Execute the instructions:

- You, the teacher, will pretend you are the robot.
- Collect all the pseudocode instructions and chose one to perform. The one with the least number of steps is usually a good one to start with, as they will have made some assumptions about what you, the robot, know how to do.
- Follow the instructions as written. This is your chance to have some fun, usually by following their exact instructions without using 'what you know they meant'.
- For example:
    - The first step is often 'Open the bread.' Feel free to just rip open the bag of bread as their instructions did not say anything about untying the knot or unclipping the bag opening.
    - If the instruction is 'Put two slices of bread on the plate.', you can put one slice on top of the other since that instruction did not specify how to place the slices.
    - If their next instruction is 'Put peanut butter on one slice of bread', you can put the whole peanut butter jar on the bread slice, since they gave no instructions about opening the jar first.
    - If an instruction tells you to do something you simply cannot, like 'Use knife to scoop out jelly' yet the jelly jar isn't even open, you can just report a 'runtime error' and stop the program.
- By the time you have gone through a few steps, or run a couple of programs to the point where they produce an error, the students have gotten the idea that they have left out important steps and also made assumptions about what functions you already know how to perform and are asking for their papers back so they can re-write their pseudocode.

Rewrite pseudocode into functions:

- Give them the chance to re-write their 'MakePB&J' functions and let them know some smaller functions you already know how to perform. For example: If they write 'Open jelly jar' or 'Take lid off of jelly jar', tell them you already know the 'OpenJar' function, so they do not need to write it, 'Grasp lid tightly. Twist lid to the left…'
- Students will then start asking, 'Do you know how to..?', checking to see what other functions you already know.
- Perform some of their revised functions. There are usually students who will happily consume the results!

Example:

Main Program Function	Helper Functions
MakePB&J ● 'Open' bread bag	Open

• Remove 2 slices of bread • Place each slice face-down side by side on the plate • 'OpenJar' Peanut butter • 'OpenJar' Jelly • Pick up knife • 'Spread' Peanut butter on one slice of bread • 'Spread' Jelly on the other slice of bread • Put knife down • Pick up one slice of bread and lay face-down on the other slice of bread • Wrap the bread in a paper towel	• Grasp the end of the bag with the opening • Unclip the plastic holder • Untwist the wrapping • Reach in OpenJar • Put one hand on the top lid of the jar, and grasp tightly • Put the other hand around the base of the jar • Repeat until lid is loose: Twist your top hand counter-clockwise • Remove the lid of the jar Spread • With a knife, reach into the Jar • Scoop out contents • Move knife backwards and forwards over bread until knife is clean • Repeat previous 3 steps until bread is completely covered

Challenge: (Can be given as homework.)

- Have each student choose a 'simple' task like tying a shoe or brushing their teeth and write in pseudocode a function to perform that task.
- Along with their pseudocode, each student should bring in whatever props are necessary to perform their function.
- Select a student's pseudocode and give that function and the props to another student to perform.
- After watching you the day before, the students are primed to follow the instructions as written!

These exercises help students realize the value of functions as a way to organize their programs, and also how each function can itself include 'calls' to smaller functions.

Additional Challenge: A Cleaning Robot

- How might you program a robot to clean the house?
- Are there tasks that are common for all rooms of the house? (Vacuuming, dusting)
- Are there tasks that are specific to certain rooms in the house? (Clean the toilet, make the bed)
- Which tasks can you assign to functions? (pickUpStuff, dust, sweep, vacuum)
- How might you break up the overall task of cleaning a house into specific functions? (cleanTheKitchen, cleanTheLivingRoom, cleanBathrooms, etc.)

Have students write two different examples for cleaning a room in the house using pseudocode.

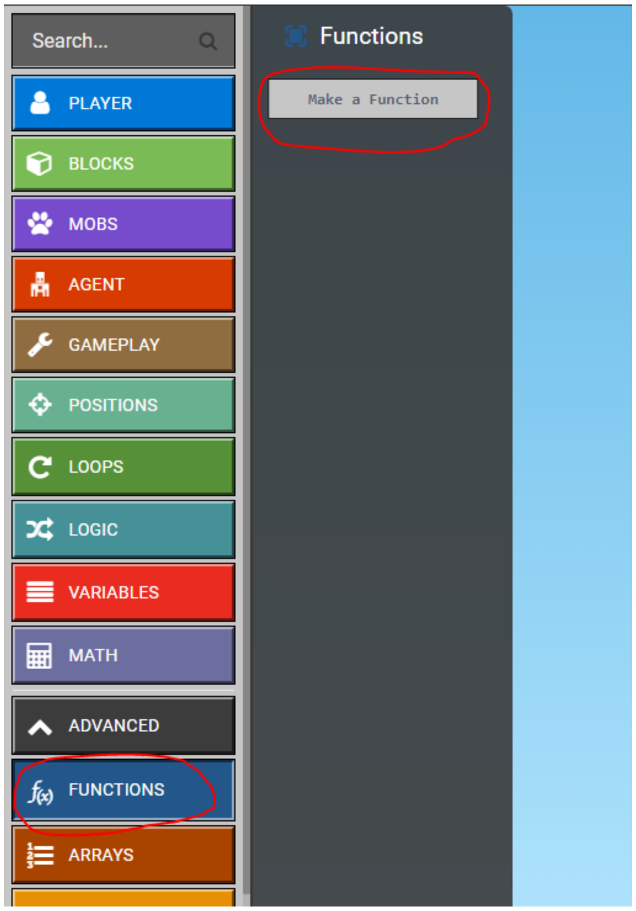
Activity: Leap of Faith Mini-Game
Students love to create mini-games in Minecraft for their friends to play. In this activity, we're going to create a simple mini-game that creates a tiny pool of water, then transports the player to a little platform 64 blocks high where the only way down is to jump and land in the pool of water! If you jump, and miss the pool of water, you lose. Luckily, we'll build this mini-game in MakeCode using functions, so you can try again and again.

Our mini-game will have three parts:

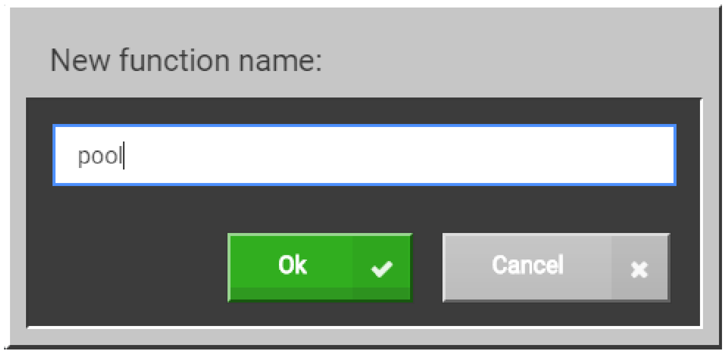
1. Creating the pool of water
2. Creating the platform
3. Teleporting the player to the top of the platform

<u>Steps:</u>

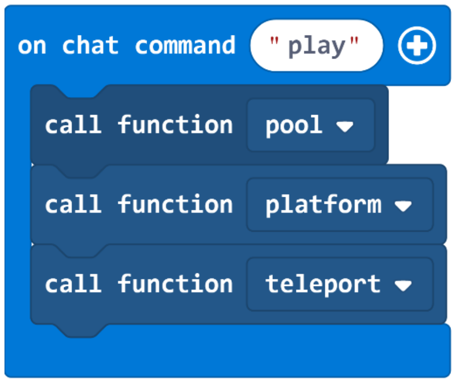
1. Create a new MakeCode project called "Leap"
2. Click the Advanced tab on the Toolbox to display more Toolbox categories
3. In the Functions Toolbox drawer, click on 'Make a Function' button



4. Name this function *pool*, and click Ok



5. Repeat steps 3 and 4 to create two more functions named: *platform* and *teleport*
6. From the Player Toolbox drawer, drag an On chat command block onto the Workspace
7. Rename this On chat command to "*play*"
8. From the Functions Toolbox drawer, drag the three blocks: Call function pool, Call function platform, Call function teleport into the On chat command block

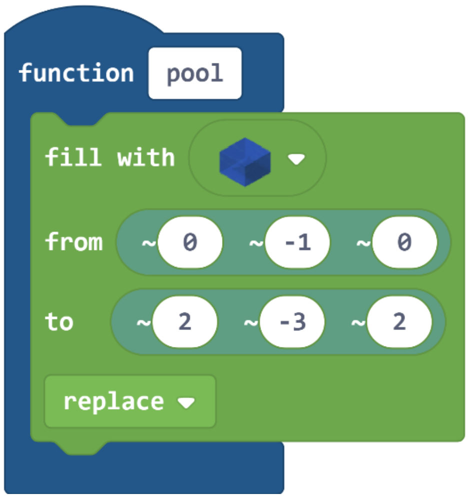


```
on chat command "play"
    call function pool
    call function platform
    call function teleport
```

This will be our main program to start our mini-game. Now, let's build out these three functions.

The first thing we'll do is create a pool of water.

1. From the Blocks Toolbox drawer, drag a Fill with block into the Function pool block. The Fill with block will fill a three-dimensional box from the first set of coordinates, to the second set of coordinates.
2. Using the drop-down menu in the Fill with block, select a block of water
3. In the Fill with block, type the following values for the first set of from coordinates: (0, -1, 0)
4. In the Fill with block, type the following values for the second set of to coordinates: (2, -3, 2)

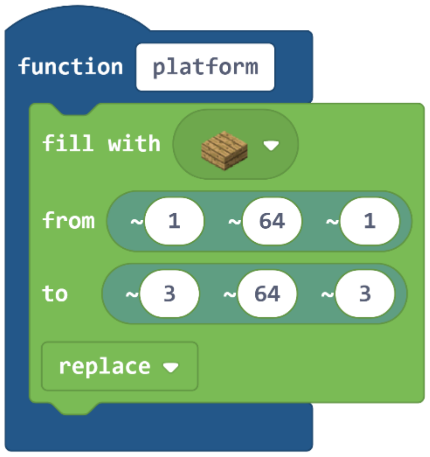


```
function pool
    fill with
    from ~0 ~-1 ~0
    to ~2 ~-3 ~2
    replace
```

This will make a 2 x 2 x 2 pool of water that is located below our player's feet. The Fill block is set to *replace* by default, which means that the existing blocks will be replaced with water.

Once the pool is built, a platform will need to be built high in the sky. There's no need for a tower, or a ladder -- there's only one way down! We want to create a wooden platform that is slightly offset from the pool below, so that you have at least a chance of jumping off the right side of the platform and landing in the pool.

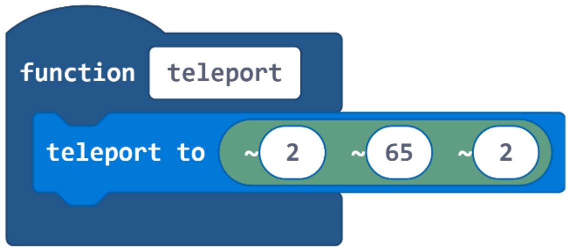
1. From the Blocks Toolbox drawer, drag another Fill with block into the Function platform block
2. Using the drop-down menu in the Fill with block, select a Wood Slab
3. In the Fill with block, type the following values for the first set of from coordinates: (1, 64, 1)
4. In the Fill with block, type the following values for the second set of to coordinates: (3, 64, 3)



```
function platform
    fill with [Wood Slab]
    from ~1 ~64 ~1
    to ~3 ~64 ~3
    replace
```

The last step is to teleport the player to a spot just above the middle of the platform.

5. From the Player Toolbox drawer, drag out a Teleport to block into the Function teleport
6. In the Teleport to block, type the following values for the coordinates: (2, 65, 2)



```
function teleport
    teleport to ~2 ~65 ~2
```

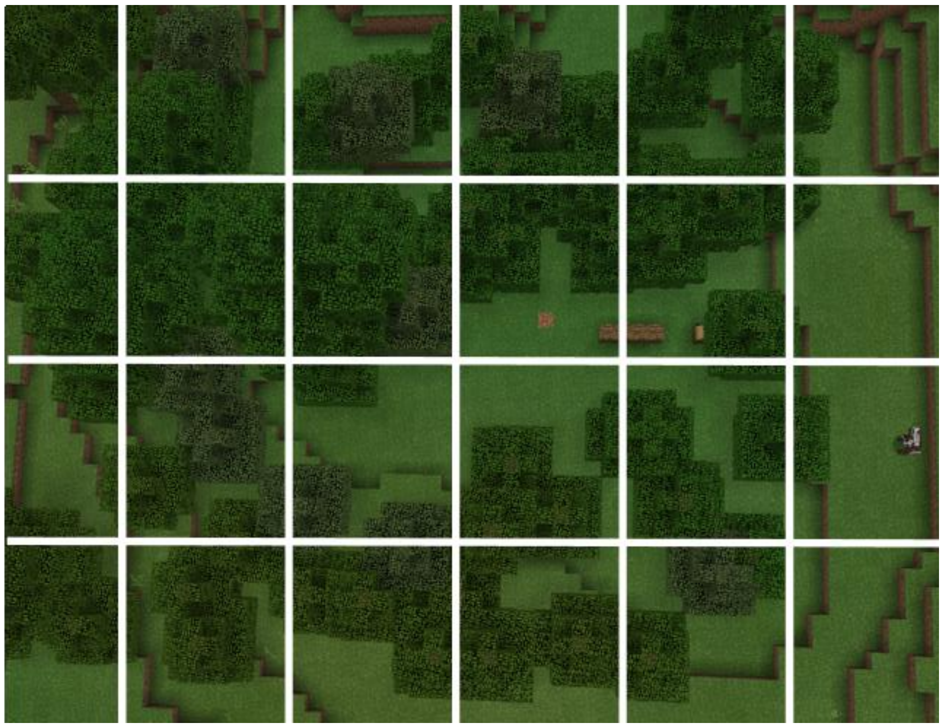
Now to try out your mini-game!
In a Minecraft world, set the game mode to Survival, then find a spot of open ground, and execute your mini-game by typing 'play' in the chat window. Then take a Leap of Faith! Remember that when you are up on the platform, holding down the shift key while moving around will keep you from falling off while you look for a good spot to jump. Have students try playing each other's games.

It's a long way down!

Activity: Tree Hunter
You'll recall in the Conditionals Lesson, we coded a tree chopping Agent that works well for individual trees, but it requires an intelligent player to stand at the base of a tree and teleport the Agent to the tree each time. Imagine if you could turn a more intelligent Agent loose in a grove of trees and have it find trees and chop them down automatically, without involving the player. What would be involved?

When we see a grove of trees, it's always easy to see the next tree to chop down. However, the Agent can only inspect blocks that are immediately forward, to the left, or to the right of its current position. To imagine how to code an Intelligent Agent, picture yourself working your way through a grove of trees, blindfolded. How might you proceed?

One way to solve this problem is to divide the grove into a grid and proceed through the grid row by row. It's slow, but you will eventually inspect every square on the grid. To make this a little bit more efficient, by reducing the number of passes we will make, we'll test forward, left, and right at every step.



We also need to plan for the changing terrain. Trees rarely grow on a flat plain; there are usually numerous changes in elevation and we will want to make sure the Agent follows the natural ground terrain so that it always starts at the base of each tree.

Finally, we will need an actual tree chopping program. We can use the one we created in the Conditionals Lesson. Because this is a fairly complex program, let's break it down into the different parts:

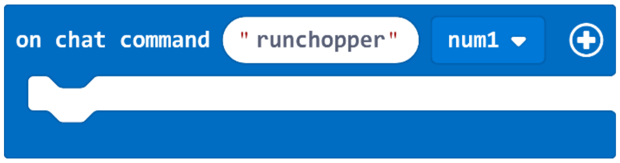
- Main RunChopper Program
- Searching for Trees (search function)
- Following the Terrain (follow function)
- Turning around at the end of a row (turn function)
- Chopping Trees (chop function)

Steps:

1. Create a new MakeCode program called 'Tree Hunter'
2. Rename the On chat command block to "runchopper"

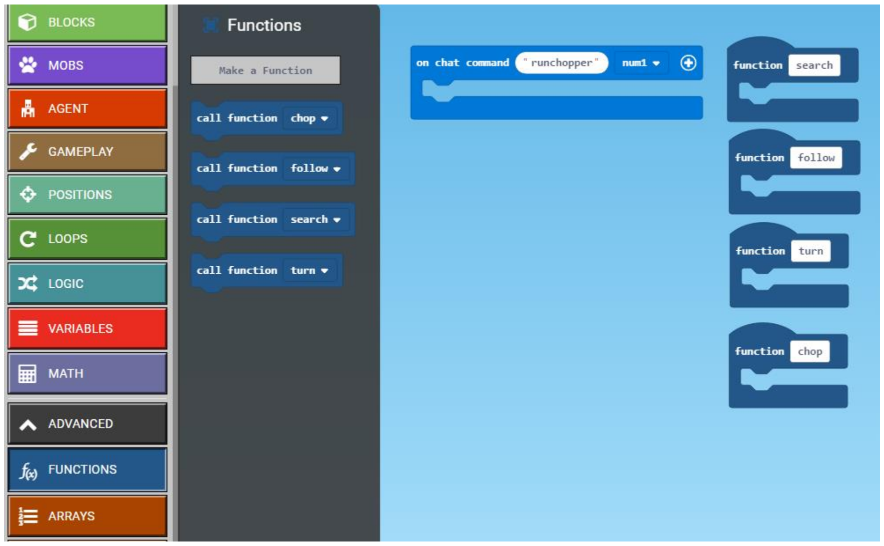
To determine the size of the area to search, let's attach a number as a parameter to our RunChopper command block that will represent the length of one side of the square search area. For example, if we call "runchopper 25" we want our Agent to search within a 25 x 25 block square area, which should encompass a good number of trees.

3. Click the plus sign (+) in the On chat command block to add the num1 parameter to our block



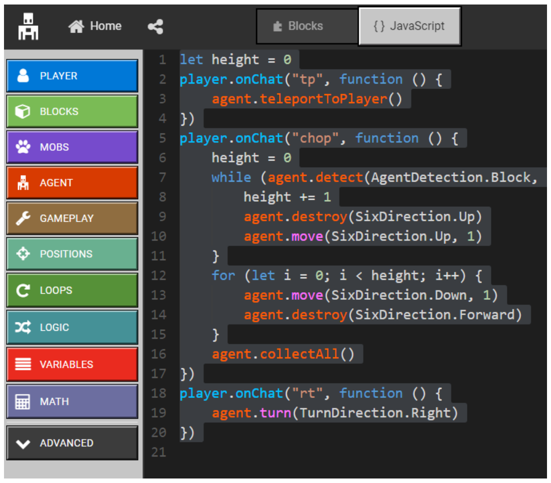
Let's first create the 4 functions that will make up our program.

4. Click the Advanced tab on the Toolbox to display the additional Toolbox categories
5. From the Functions Toolbox drawer, click the 'Make a Function' button 4 times
6. Name the functions: *search*, *follow*, *turn*, and *chop*



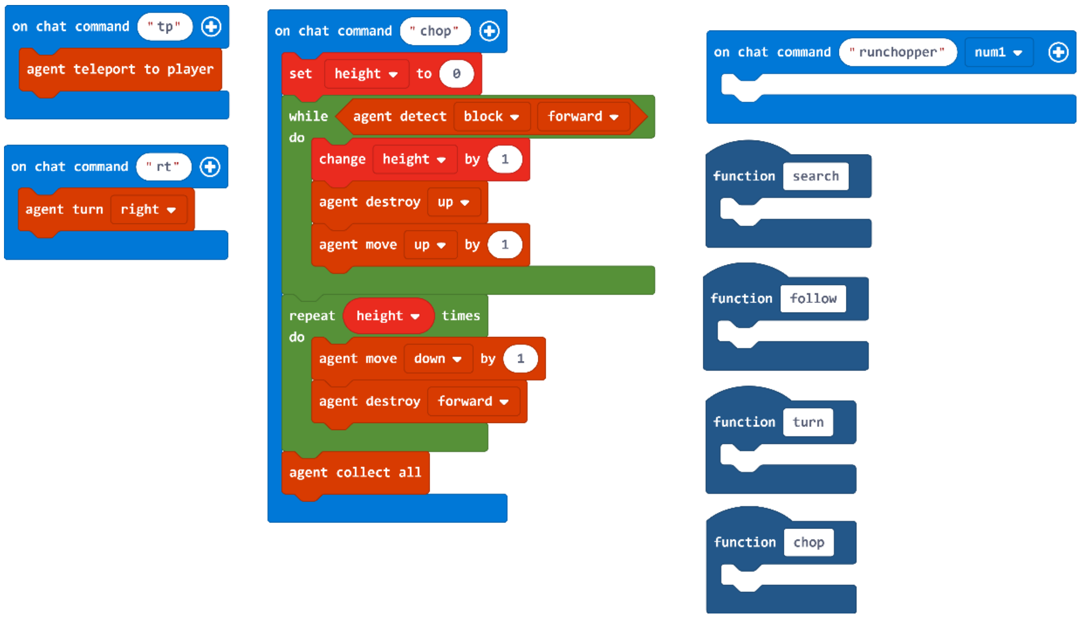
Let's start with the chop function, since we've already completed this code in the Conditionals Lesson.

7. Open the Chopper project you created previously in MakeCode
8. Click the JavaScript button to see your program in the JavaScript editor
9. Select all the code (Ctrl-A) and copy it to your clipboard (Ctrl-C)



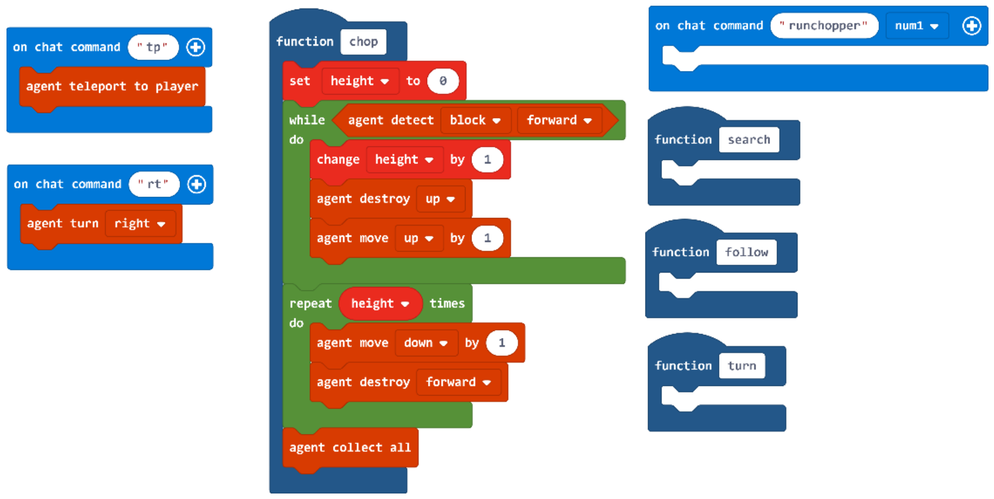
```
let height = 0
player.onChat("tp", function () {
    agent.teleportToPlayer()
})
player.onChat("chop", function () {
    height = 0
    while (agent.detect(AgentDetection.Block,
        height += 1
        agent.destroy(SixDirection.Up)
        agent.move(SixDirection.Up, 1)
    }
    for (let i = 0; i < height; i++) {
        agent.move(SixDirection.Down, 1)
        agent.destroy(SixDirection.Forward)
    }
    agent.collectAll()
})
player.onChat("rt", function () {
    agent.turn(TurnDirection.Right)
})
```

10. Click the Home button at the top of the screen to get back to the MakeCode home page
11. Open back up the Tree Hunter project in MakeCode
12. Click the JavaScript button to see your program in the JavaScript editor
13. Place your cursor in the last line, and paste the code you copied (Ctrl-V)
14. Click the Blocks button to get back to the Block editor. You should see the code you wrote for the Chopper project here.



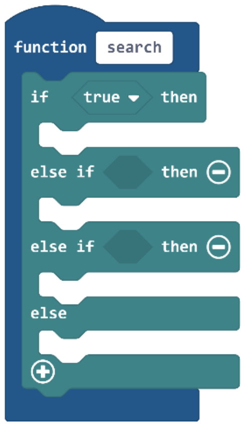
```
on chat command "tp"
    agent teleport to player

on chat command "rt"
    agent turn right

on chat command "chop"
    set height to 0
    while agent detect block forward
    do
        change height by 1
        agent destroy up
        agent move up by 1
    repeat height times
    do
        agent move down by 1
        agent destroy forward
    agent collect all

on chat command "runchopper" num1

function search

function follow

function turn

function chop
```

15. Drag all the blocks from under the On chat command “chop” to the Function chop
16. Delete the On chat command “chop”

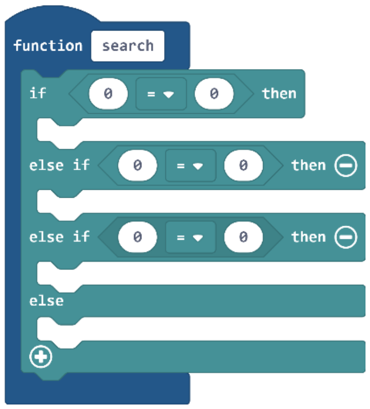

```
on chat command "tp"
    agent teleport to player

on chat command "rt"
    agent turn right

function chop
    set height to 0
    while agent detect block forward
    do
        change height by 1
        agent destroy up
        agent move up by 1
    repeat height times
    do
        agent move down by 1
        agent destroy forward
    agent collect all

on chat command "runchopper" num1

function search

function follow

function turn
```

Moving on to the search function, which will have the Agent check to see if there are any trees around it. The Agent will check to the right, in front, and to the left for wood blocks indicating a tree. If it finds a wood block, it will call the Function chop to chop the tree down.

17. From the Logic Toolbox drawer, drag an If Then Else block into the Function search block
18. In the If Then Else block, click the plus sign (+) two times to create two additional Else if clauses

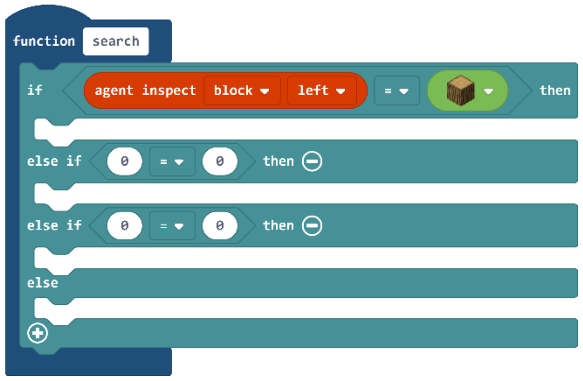

```
function search
    if true then
    else if then
    else if then
    else
```

19. From the Logic Toolbox drawer, drag 3 equals Comparison blocks into each of the If, and Else if slots (remember, you can also right-click on a block and select Duplicate)



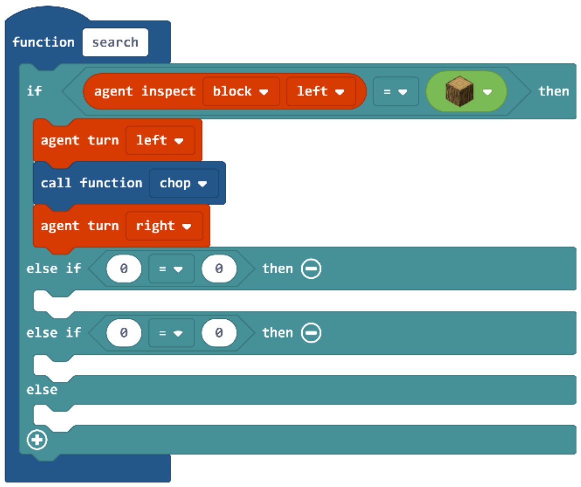
Let's first see if there is a block of wood to the left of the Agent.

20. From the Agent Toolbox drawer, drag an Agent inspect block into the first slot of the Equals comparison block in the first If clause
21. In the Agent inspect block, use the second drop-down menu to select '*left*' as the direction to look
22. From the Blocks Toolbox drawer, drag out a Block block and drop into the second slot of the Equals comparison block
23. In the Block block, use the drop-down menu to select an Oak Wood block



If we do find some wood to the left of the Agent, then we want to turn the Agent left and call our Chop function to chop down the tree, then turn the Agent back around to face forward again.

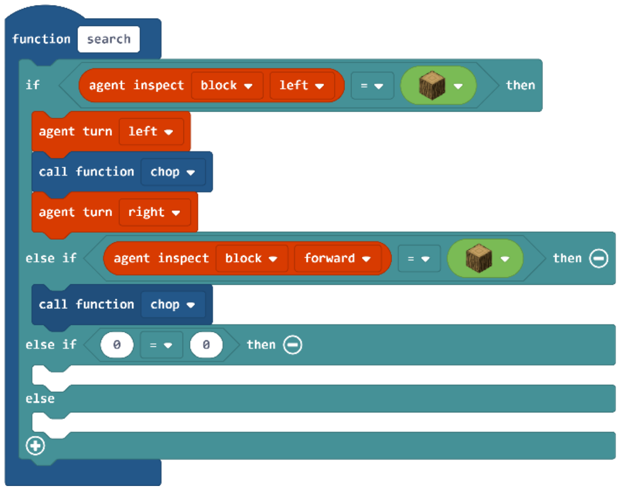
24. From the Agent Toolbox drawer, drag an Agent turn block under the If Then clause
25. From the Functions Toolbox drawer, drag a Call function chop block under the Agent turn block
26. From the Agent Toolbox drawer, drag an Agent turn block under the Call function chop block
27. In the Agent turn block, use the drop-down menu to select the direction as '*right*'



```
function search
  if agent inspect block left = then
    agent turn left
    call function chop
    agent turn right
  else if 0 = 0 then
  else if 0 = 0 then
  else
```

Now, let's check for wood to the front of our Agent. In this case, we don't need to turn the Agent.

28. From the Agent Toolbox drawer, drag an Agent inspect block into the first slot of the Equals comparison block in the first Else if clause
29. From the Blocks Toolbox drawer, drag out a Block block and drop into the second slot of the Equals comparison block
30. In the Block block, use the drop-down menu to select an Oak Wood block
31. From the Functions Toolbox drawer, drag a Call function chop block under the first Else if clause



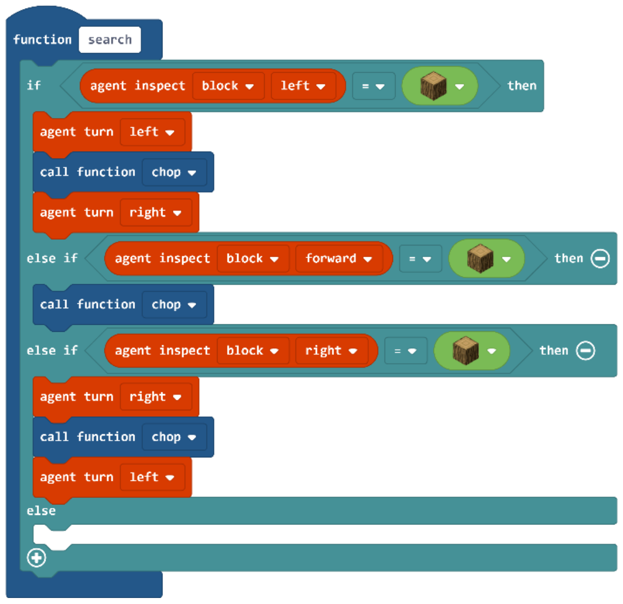
```
function search
  if agent inspect block left = then
    agent turn left
    call function chop
    agent turn right
  else if agent inspect block forward = then
    call function chop
  else if 0 = 0 then
  else
```

Finally, let's see if there is a block of wood to the right of the Agent.

32. From the Agent Toolbox drawer, drag an Agent inspect block into the first slot of the Equals comparison block in the second Else If clause
33. In the Agent inspect block, use the second drop-down menu to select '*right*' as the direction to look

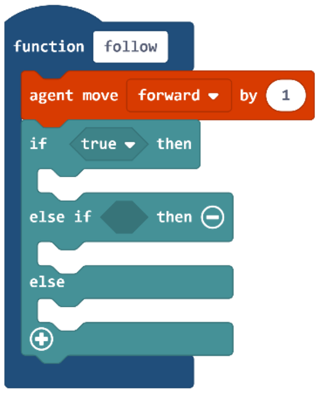
34. From the Blocks Toolbox drawer, drag out a Block block and drop into the second slot of the Equals comparison block
35. In the Block block, use the drop-down menu to select an Oak Wood block
36. From the Agent Toolbox drawer, drag an Agent turn block under the Else if clause
37. In the Agent turn block, use the drop-down menu to select '*right*' as the direction to turn
38. From the Functions Toolbox drawer, drag a Call function chop block under the Agent turn block
39. From the Agent Toolbox drawer, drag an Agent turn block under the Call function chop block

If there is no wood to the left, forward, or right of the Agent, then we do nothing. So the last Else clause will be empty.

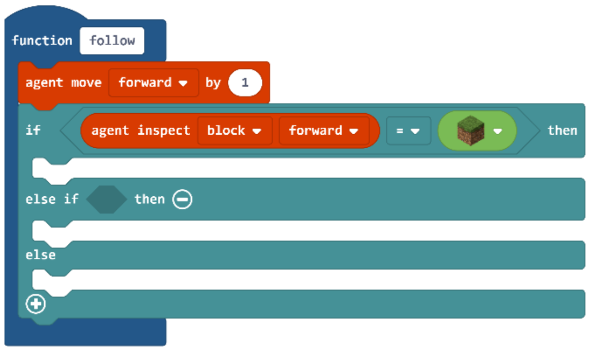


```
function search
  if agent inspect block left = [oak wood] then
    agent turn left
    call function chop
    agent turn right
  else if agent inspect block forward = [oak wood] then
    call function chop
  else if agent inspect block right = [oak wood] then
    agent turn right
    call function chop
    agent turn left
  else
```

Now, let's focus on the Follow function to make our Agent move along the ground following the terrain. In order to keep our Agent searching at ground level, we'll need to move the Agent up if it detects a block in front of it and move the Agent down if it detects no block below it.
40. From the Agent Toolbox drawer, drag an Agent move block into the Function follow
41. From the Logic Toolbox drawer, drag an If Then Else block under the Agent move block
42. In the If Then Else block, click the plus sign (+) to create an Else if clause

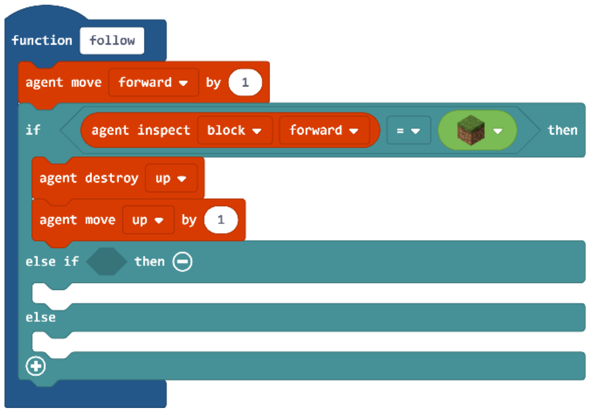


43. From the Logic Toolbox drawer, drag an Equals (=) comparison block into the If clause replacing 'true'
44. From the Agent Toolbox drawer, drag an Agent inspect block into the first slot of the Equals comparison block
45. From the Blocks Toolbox drawer, drag a Block block into the second slot of the Equals comparison block



If the block in front of the Agent is a grass block, then we need to move the Agent up. But we also need to make sure there's nothing blocking the Agent from moving up (like leaves).

46. From the Agent Toolbox drawer, drag an Agent destroy block into the If Then clause
47. In the Agent destroy block, use the drop-down menu to select '*up*' as the direction
48. From the Agent Toolbox drawer, drag an Agent move block under the Agent destroy block
49. In the Agent move block, use the drop-down menu to select '*up*' as the direction



```
function follow
    agent move forward ▾ by 1
    if agent inspect block ▾ forward ▾ = ▾ 🟩 ▾ then
        agent destroy up ▾
        agent move up ▾ by 1
    else if then ⊖
    else
    ⊕
```

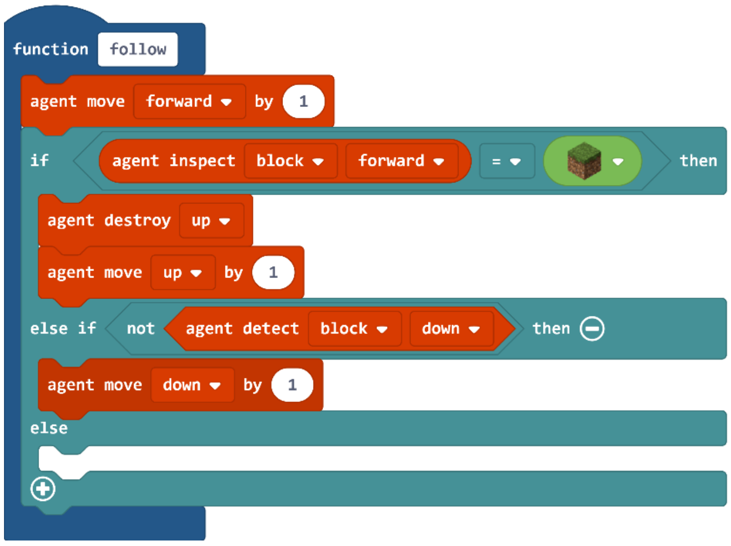
Now, let's take care of the case where there is a dip in the ground. If the Agent doesn't detect a block beneath her, then we'll need to move the Agent down.

50. From the Logic Toolbox drawer, drag a Not block into the Else If clause
51. From the Agent Toolbox drawer, drag an Agent detect block into the Not block
52. In the Agent detect block, use the second drop-down menu to select '*down*' as the direction

This essentially means that the Agent does NOT detect a block below her.

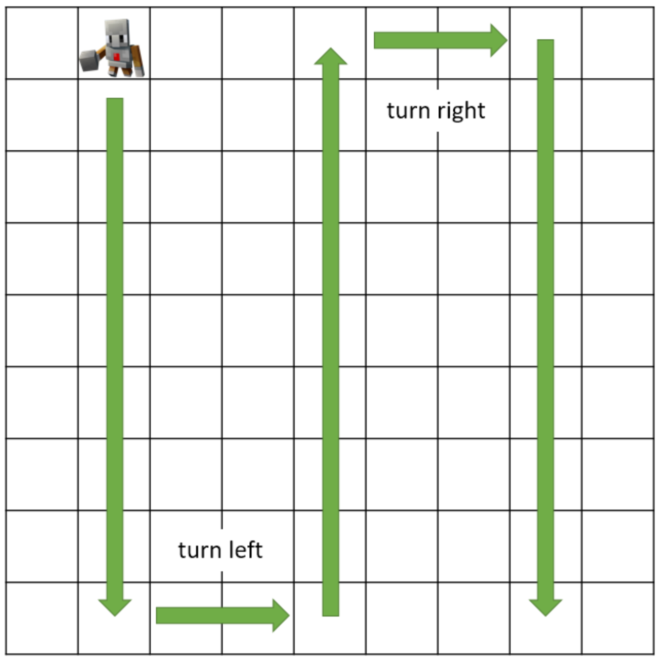
53. From the Agent Toolbox drawer, drag an Agent move block under the Else If clause
54. In the Agent move block, use the drop-down menu to select '*down*' as the direction

The Follow function will move the Agent forward and check if there is a block in front of it → if there is, the Agent will move up. If there isn't a block in front, it will check if there is a block below it → if there is not, the Agent will move down. If there isn't a block in front, and there is a block below it, then it won't do anything.



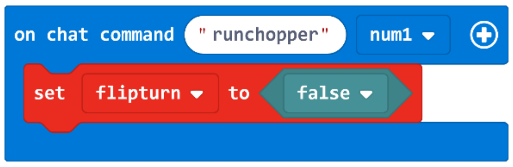
```
function follow
    agent move forward ▾ by 1
    if agent inspect block ▾ forward ▾ = ▾ 🟩 ▾ then
        agent destroy up ▾
        agent move up ▾ by 1
    else if not agent detect block ▾ down ▾ then ⊖
        agent move down ▾ by 1
    else
    ⊕
```

For the final Turn function, we need the Agent to turn around at the end of each row. But she needs to turn alternately left and then right at the end of each row.



What is the best way to ensure the Agent alternates its turns?
Let's create a boolean variable to keep track of which direction to turn. Recall from the Variables Lesson, a Boolean variable has only two possible values: True, and False. We'll use True to denote a Right turn, and False to represent a Left turn. After we make our turn, we'll change the boolean variable to its opposite value using the Not block.

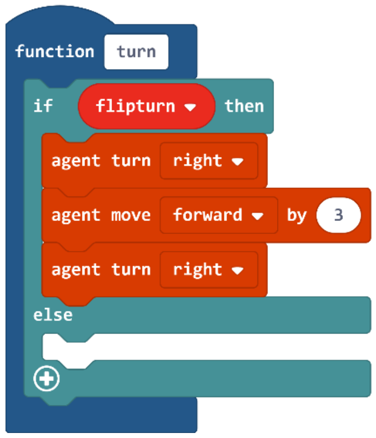
55. In the Variables Toolbox drawer, click on the 'Make a Variable' button
56. Name this variable 'flipturn' and click Ok
57. From the Variables Toolbox drawer, drag a Set block under the On chat command "runchopper"
58. In the Set block, use the drop-down menu to select the 'flipturn' variable
59. From the Logic Toolbox drawer, drag a False block into the Set block slot replacing 0



```
on chat command "runchopper" num1 ⊕
    set flipturn to false
```

60. From the Logic Toolbox drawer, drag an If Then Else block into the Function turn block
61. From the Variables Toolbox drawer, drag the flipturn variable block into the If clause replacing 'true'
62. From the Agent Toolbox drawer, drag an Agent turn block under the If Then clause
63. In the Agent turn block, use the drop-down menu to select '*right*' as the direction
64. From the Agent Toolbox drawer, drag an Agent move block under the Agent turn block

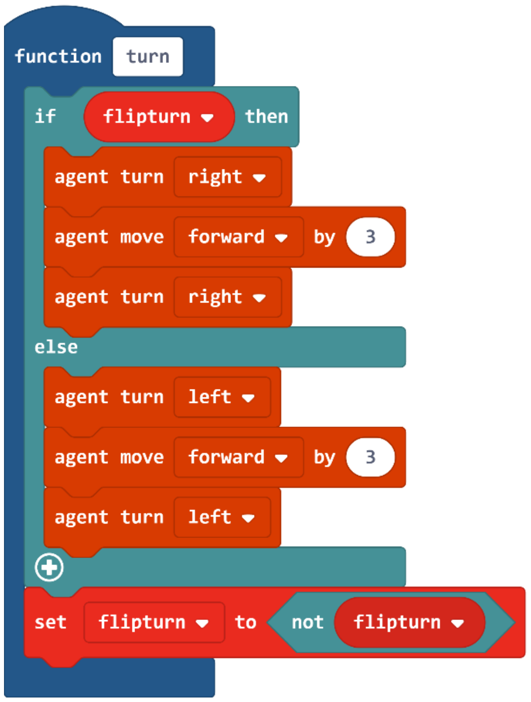
65. Type '3' into the Agent move block slot to move forward 3 blocks
66. From the Agent Toolbox drawer, drag another Agent turn block under the Agent move block
67. In the Agent turn block, use the drop-down menu to select '*right*' as the direction



```
function turn
  if flipturn then
    agent turn right
    agent move forward by 3
    agent turn right
  else
```

68. From the Agent Toolbox drawer, drag an Agent turn block under the Else clause
69. From the Agent Toolbox drawer, drag an Agent move block under the Agent turn block
70. Type '3' into the Agent move block slot to move forward 3 blocks
71. From the Agent Toolbox drawer, drag another Agent turn block under the Agent move block
72. Lastly, from the Variables Toolbox drawer, drag a Set block after the If Then Else clause in the Function turn
73. In the Set block, use the drop-down menu to select the flipturn variable
74. From the Logic Toolbox drawer, drag a Not block into the Set block slot replacing 0
75. From the Variables Toolbox drawer, drag a flipturn variable block into the Not block

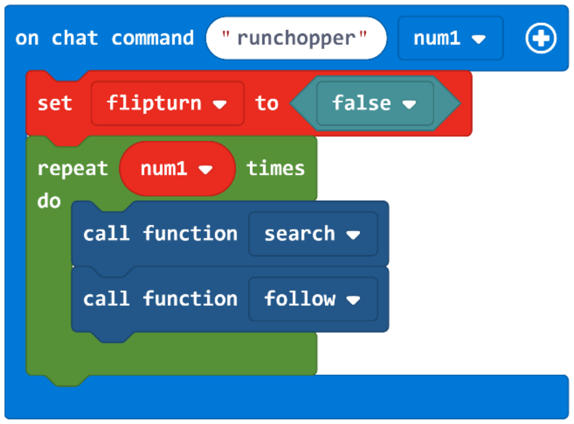
If the flipturn variable is True, then the Agent will turn right 90° move forward 3 blocks then turn right 90°. If the flipturn variable is False, then the Agent will turn left 90° move forward 3 blocks then turn left 90°. Then we will set the value of the flipturn variable to the opposite of what it currently is.



```
function turn
  if flipturn then
    agent turn right
    agent move forward by 3
    agent turn right
  else
    agent turn left
    agent move forward by 3
    agent turn left
  set flipturn to not flipturn
```

Now that we have all our functions built, let's bring it all together. For each step, we want our Agent to Search for trees, and Follow the ground terrain. And remember, that we set our num1 parameter to be the number of blocks in each row.

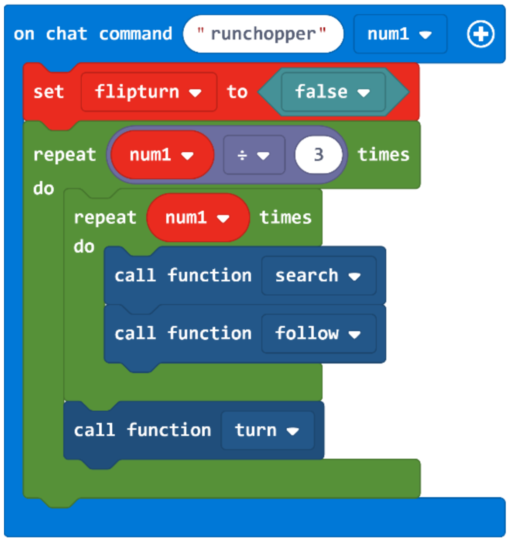
76. From the Loops Toolbox drawer, drag a Repeat block into the On chat command "runchopper" under the Set block
77. From the Variables Toolbox drawer, drag the num1 variable block into the Repeat block replacing '4'
78. From the Functions Toolbox drawer, drag the Call function search, and the Call function follow blocks under the Repeat loop



```
on chat command "runchopper" num1
  set flipturn to false
  repeat num1 times
  do
    call function search
    call function follow
```

Now, at the end of each row, we want to turn our Agent around. Let's imagine we have called "search 25" which should search a 25 x 25 block area. But on one pass we are checking three blocks at a time. So we can divide 25 by 3 and only make that many passes to cover the entire area. If we express that in terms of *num1*, we have *num1* divided by 3, which represents the outer loop.

79. From the Loops Toolbox drawer, drag another Repeat loop, and place it around the outside of the existing Repeat loop in the On chat command “runchopper”. Make sure the Set block is still outside the Repeat loop.
80. From the Math Toolbox drawer, drag a Division (÷) block into the Repeat loop replacing ‘4’
81. From the Variables Toolbox drawer, drag the num1 variable block into the first slot of the Division block replacing ‘0’
82. In the Division block, type ‘3’ in the second slot
83. From the Functions Toolbox drawer, drag a Call function turn block into the outer Repeat loop after the inner Repeat loop

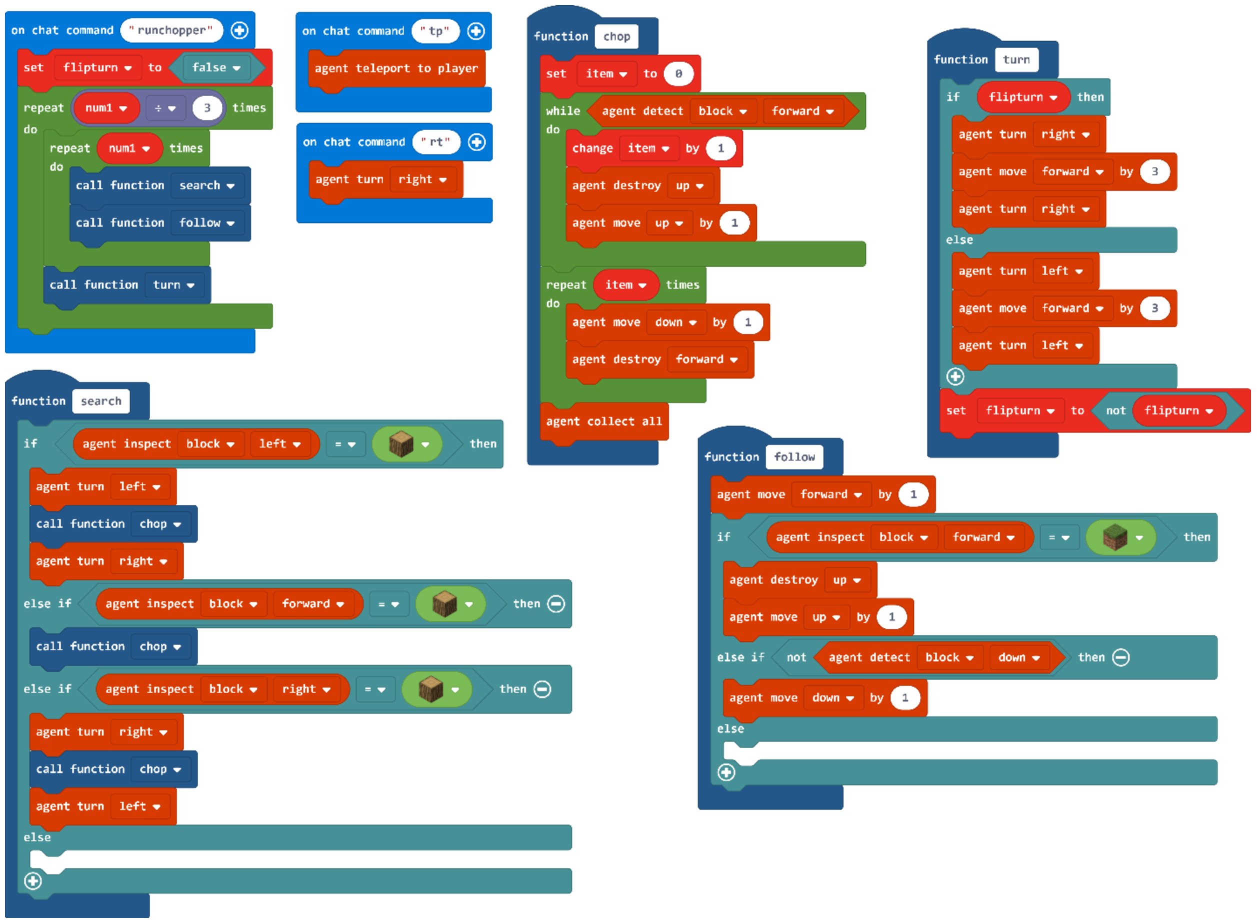


```
on chat command "runchopper" num1
    set flipturn to false
    repeat num1 ÷ 3 times
    do
        repeat num1 times
        do
            call function search
            call function follow
        call function turn
```

Optional Extensions

- Notice that the Follow function will only work on grass blocks, and only for dips and rises 1 block high. Modify the code to work for both grass and dirt blocks, and for dips and rises in the terrain more than 1 block high or low.
- Modify the code to work on 2 x 2 trees such as giant jungle trees and dark oak trees.
- Modify the code to work on acacia trees, which have a diagonal trunk.
- Modify the code to work on tree branches.
- If there are multiple trees around the Agent, the Agent will only chop down one of the trees – in the order of left, forward, right. Find a way to search and chop down all trees around the Agent.

Your final program may look like the following:



```
on chat command "runchopper"
    set flipturn to false
    repeat num1 ÷ 3 times
    do
        repeat num1 times
        do
            call function search
            call function follow
        call function turn

on chat command "tp"
    agent teleport to player

on chat command "rt"
    agent turn right

function chop
    set item to 0
    while agent detect block forward
    do
        change item by 1
        agent destroy up
        agent move up by 1
    repeat item times
    do
        agent move down by 1
        agent destroy forward
    agent collect all

function turn
    if flipturn then
        agent turn right
        agent move forward by 3
        agent turn right
    else
        agent turn left
        agent move forward by 3
        agent turn left
    set flipturn to not flipturn

function search
    if agent inspect block left = then
        agent turn left
        call function chop
        agent turn right
    else if agent inspect block forward = then
        call function chop
    else if agent inspect block right = then
        agent turn right
        call function chop
        agent turn left
    else

function follow
    agent move forward by 1
    if agent inspect block forward = then
        agent destroy up
        agent move up by 1
    else if not agent detect block down then
        agent move down by 1
    else
```

JavaScript:

```
let item = 0
let flipturn = false
player.onChat("runchopper", function (num1) {
    flipturn = false
    for (let i = 0; i < num1 / 3; i++) {
        for (let i = 0; i < num1; i++) {
            search()
            follow()
        }
        turn()
    }
})
player.onChat("tp", function () {
    agent.teleportToPlayer()
})
function chop() {
    item = 0
    while (agent.detect(AgentDetection.Block,
SixDirection.Forward)) {
        item += 1
        agent.destroy(SixDirection.Up)
        agent.move(SixDirection.Up, 1)
    }
    for (let i = 0; i < item; i++) {
        agent.move(SixDirection.Down, 1)
        agent.destroy(SixDirection.Forward)
    }
    agent.collectAll()
}
player.onChat("rt", function () {
    agent.turn(TurnDirection.Right)
})
function turn() {
    if (flipturn) {
        agent.turn(TurnDirection.Right)
        agent.move(SixDirection.Forward, 3)
        agent.turn(TurnDirection.Right)
    } else {
        agent.turn(TurnDirection.Left)
        agent.move(SixDirection.Forward, 3)
        agent.turn(TurnDirection.Left)
    }
    flipturn = !(flipturn)
}
function search() {
    if (agent.inspect(AgentInspection.Block,
SixDirection.Left) == blocks.block(Block.LogOak)) {
        agent.turn(TurnDirection.Left)
        chop()
        agent.turn(TurnDirection.Right)
    } else if (agent.inspect(AgentInspection.Block,
SixDirection.Forward) == blocks.block(Block.LogOak)) {
        chop()
    } else if (agent.inspect(AgentInspection.Block,
SixDirection.Right) == blocks.block(Block.LogOak)) {
        agent.turn(TurnDirection.Right)
        chop()
        agent.turn(TurnDirection.Left)
    } else {

    }
}
function follow() {
    agent.move(SixDirection.Forward, 1)
    if (agent.inspect(AgentInspection.Block,
SixDirection.Forward) == blocks.block(Block.Grass)) {
        agent.destroy(SixDirection.Up)
        agent.move(SixDirection.Up, 1)
    } else if (!(agent.detect(AgentDetection.Block,
SixDirection.Down))) {
        agent.move(SixDirection.Down, 1)
    } else {

    }
}
```

Shared Project: https://makecode.com/_e0h6sHUx1XWj

Independent Project
In this chapter, we learned how organizing blocks of code into separate functions makes your code more readable and saves space. Functions generally contain sections of code that go together logically and accomplish one thing. The function's name usually tells you what it does, and it's generally a verb. Functions can call other functions. Although functions in MakeCode do not accept parameters, you can use an On chat command block to pass in a parameter, and then update the value of an existing variable that any function can access.

Now it's your turn to practice using functions as you build something in Minecraft. One of the first things to do in Minecraft is build yourself a house. Your house can be decorated in different kinds of wood, use big windows to let in lots of natural light, or be dark and dungeony with traps to catch unwary visitors. It could have tall castle-like turrets, or go several levels underground and blend in with the natural landscape. You might also use the land around you to grow carrots and wheat to feed yourself and the animals around you. Your house, and the grounds around it, are a chance for you to express yourself through creatively building. MakeCode can help you direct the Agent to build for you, so that your house also becomes an expression of your growing ability to code!

For this independent project, build your dream house in Minecraft, or build a replica of an actual building. Either way, use functions to automate creating as many different sections of the house as possible. You might start with creating a floor command that creates a floor by filling blocks in a square area at your feet. Or, you might have a carpeting Agent that creates long carpet runners that run the length of your house. How about a Builder that can create a farm for you, automatically?

Come up with something original that meets the following criteria:

- Creates, or helps to create, a building, temple, monument, or other piece of architecture in Minecraft
- At least three separate and distinct functions
- Descriptive names that represent what each function does

Some ideas for functions:

- Create a carpet
- Create the walls of a room
- Create a decorative fountain
- Create a castle turret
- Level an area of terrain so you have a flat place to build

- Create a swimming pool

Minecraft Diary

- Compose a diary entry addressing the following:
- What did you decide to build? Why?
- Describe each of your three functions and what each of them does
- What kinds of building tasks did you decide needed to be done by hand? Why?
- Include at least one screenshot of your finished building or piece of architecture
- Share your project to the web and include the URL here

NOTE: If you decided to improve one of this lesson's activities, please talk about the new code you wrote in addition to what was already provided in the lesson.

Assessment

	1	2	3	4
Diary	Minecraft Diary entry is missing 4 or more of the required prompts.	Minecraft Diary entry is missing 2 or 3 of the required prompts.	Minecraft Diary entry is missing 1 of the required prompts.	Minecraft Diary addresses all prompts.
Project	Project is largely ineffective and/or inefficient.	Project is missing 2 of the required elements or is somewhat ineffective and/or inefficient.	Project is missing 1 of the required elements or is mostly effective and efficient.	Project addresses all required elements effectively and efficiently.

Functions	No functions logically separate nor appropriately named.	One function logically separate and appropriately named.	Two separate functions, logically separate and appropriately named.	At least three separate functions, logically separate and appropriately named.

CSTA Standards

- 2-AP-13 Decompose problems and subproblems into parts to facilitate the design, implementation, and review of programs.
- 2-AP-14 Create procedures with parameters to organize code and make it easier to reuse.
- 3A-CS-01 Explain how abstractions hide the underlying implementation details of computing systems embedded in everyday objects.
- 3A-AP-13 Create prototypes that use algorithms to solve computational problems by leveraging prior student knowledge and personal interests.