# The Beauty and Joy of Computing[1]

*Lab Exercise 3: Abstraction with Functions*

## Objectives

By completing this lab exercise, you should learn to

- Describe the different kinds of blocks that BYOB uses;
- Define your own command, reporter, and predicate blocks;
- Simplify scripts by replacing redundant sequences with custom blocks; and
- Draw geometric shapes and patterns using BYOB scripts.

## Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly can do them if it would help you follow along with the examples.

Abstraction is a powerful tool for dealing with complex processes - it allows a programmer or designer to define processes at a high level without having to be concerned with the details of underlying tasks, and enables a programmer to re-use code for common tasks without having to start from scratch. While abstraction is used in many different ways in computing, in this lab we'll look at one particular use of abstraction: using blocks in BYOB to represent operations that require multiple individual steps.

Consider the "move … steps" block in BYOB. While it's convenient to view this as a single action, think about what really happens: first, the sprite is erased from its current position (which requires re-drawing the background and anything that was behind the sprite), the new position for the sprite is calculated based on the direction and number of steps, and the sprite is drawn in its new position. Even that description is simplified: computing the new position based on an angle and distance requires computing trigonometric functions, which consist of other more basic operations, and drawing the sprite requires looping over the rows and columns to put the picture on the screen pixel-by-pixel. The Alonzo sprite is 104 by 128 pixels in size, and each pixel is made up of three colors - so as a conservative estimate, the "move … steps" block executes at least 50,000 individual instructions in order to complete its task. This is a good example of abstraction: those 50,000 (or more!) instructions are abstracted into the simple idea of moving a sprite a certain distance - isn't it great that you don't have to think about everything that goes on behind the scenes every time you just want to move a sprite?

Each of the built-in blocks in BYOB is really a representation (an abstraction) of a more complex sequence of instructions. The designers of BYOB and Scratch have thought of common things a programmer might want to do, and have made blocks to represent those actions. One of the great things about abstractions is that, generally, you can use abstractions to build up even

[1] *Lab Exercises for "The Beauty and Joy of Computing" [This is the final version for the Fall 2012 class]*
*Copyright © 2012 by Stephen R. Tate - Creative Commons License*
*See http://www.uncg.edu/cmp/faculty/srtate/csc100labs for more information*

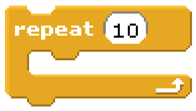
higher levels of abstraction. The ability to build up your own abstractions (i.e., make your own blocks) out of existing blocks is one of the main differences between BYOB and the earlier Scratch system, and is the source of it's name (BYOB stands for "Build Your Own Blocks"). If you disagree with the designers of BYOB, and really think they should have supplied a block that is not part of the basic BYOB system, you can make it yourself and use it just like the built-in blocks! To understand how to build your own blocks, let's first take a closer look at the types of blocks that BYOB uses.

## Types of Blocks in BYOB

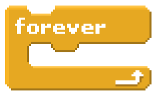
There are three basic types of blocks in BYOB, some of which can be further broken down into sub-types, and the shape of a block corresponds to what type of block it is. The most common type of block is a **command block**, which simply represents an action to perform, and is drawn as a block with an indentation on the top, and (usually) a tab on the bottom. The move block that we discussed above is an example of a command block:

move 10 steps

From the shape, you can imagine stacking these on top of each other to define a sequence of commands that control a sprite or cause some other kind of action. In a previous lab we used the term "C-block," since it had a shape like a C that could contain other blocks that it controls, such as the "repeat …" block:

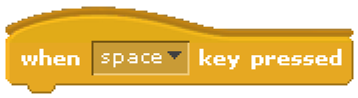


Notice that this block has the indentation at the top and the tab at the bottom, which makes this a command block - the C-block is just a sub-type of the command block type! One interesting thing to note about command blocks is that while all command blocks have the indentation on the top, not all have the tab on the bottom. For example, consider the "forever" block:



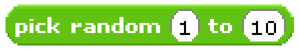
Can you guess why there is no tab at the bottom of this block?

A second type of block in BYOB is the **hat block**, which has a tab on the bottom and a curved top. This type of block specifies a starting condition for a script, indicating when the script should be executed. The most common hat block is one that "catches" an event and is at the top of the sequence of commands that define the corresponding event handler. An example of this type of block is the hat block that catches keypress events:



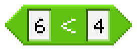
There are different hat blocks for different types of events that can trigger event handlers (keypress, broadcast message received, green flag clicked, etc.).

The final type of block in BYOB is the **reporter block**, which performs actions and reports a value as a result of those actions. Reporter blocks are always used inside other blocks which make use of the value reported as an argument. Since they are used inside blocks, rather than snapped on top of one another, they don't have tabs, but rather are either oval or hexagonal in shape. For example, the "pick random" block is an oval-shaped reporter, and reports a randomly chosen number in the range given by its arguments - the picture below shows a block that will report a random integer from 1 to 10:



This can then be snapped in as an argument to any block that has an integer parameter - for example, you could put the "pick random" block into the "repeat .." block shown above, and it would repeat the enclosed command sequence a randomly chosen number of times.

Hexagonal-shaped reporter blocks are special sub-types of reporter blocks called **predicates**. A predicate reports either true or false, usually depending on the outcome of some test. While this is the same concept as a basic reporter block, predicates are used in different situations than blocks that report a number or a string, which is why they have a different shape - they fit into different blocks, such as the "if …" block, which specifically require a predicate. An example of a predicate block is the "<" block, which reports true if the first parameter is less than the second one; therefore, this block will report false since 6 is not less than 4:

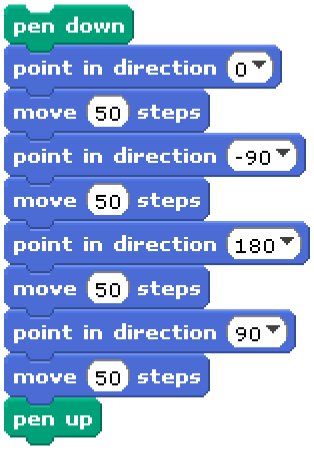


In the last lab you experimented with isolated reporter blocks by clicking on them and seeing what they "said" (you did this with the mod block). This works with any reporter block if you want to see what it does.

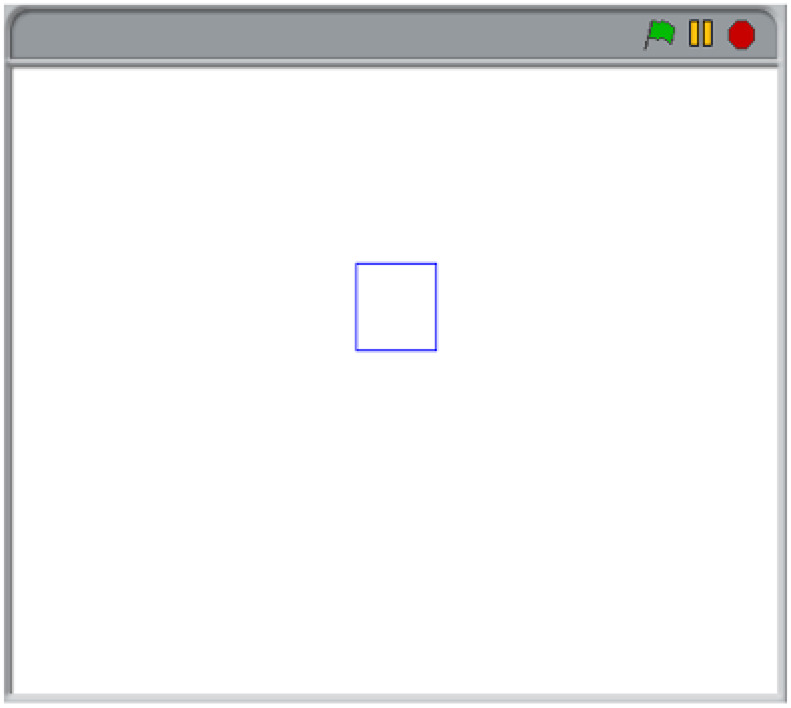
A few more words about terminology: The concepts described above are present in pretty much all programming languages, but in some cases different terminology is used. A reporter block is often called a **function**, and instead of saying it reports a value, a function will typically **return a value**. A command block, which defines a sequence of actions, is often referred to as a **procedure** or a **subroutine**, or sometimes as a function which does not return a value. That might be a little confusing now, but don't worry about it - these terms are used constantly in programming, so it becomes second-nature very quickly.

## Building a Simple Command Block

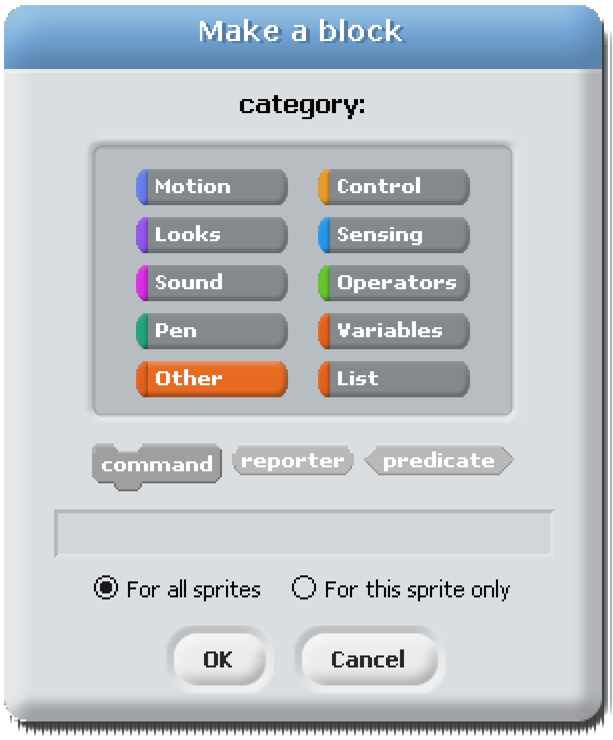
Let's see how to make a block, by showing how to create a block that draws a square, 50 units on a side: we'll start from the current sprite location, move up 50 units, left 50 units, down 50 units, and right 50 units to close the square. We start with the default project, with Alonzo in the center of the screen, and hide Alonzo since we're really not interested in seeing Alonzo. The script to draw the square and its output on the stage are shown below:



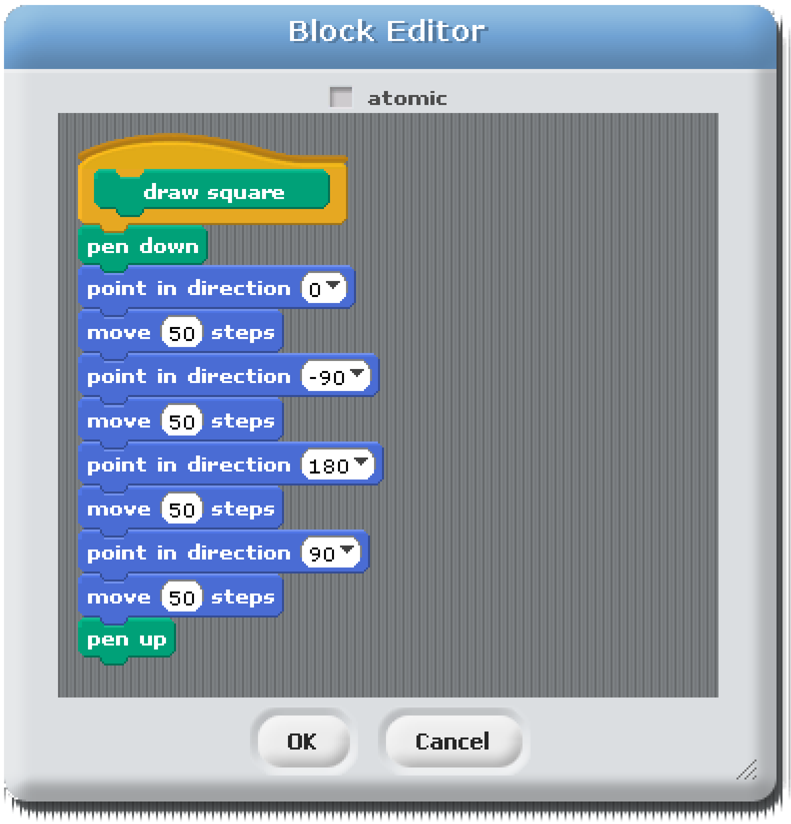
```
pen down
point in direction 0
move 50 steps
point in direction -90
move 50 steps
point in direction 180
move 50 steps
point in direction 90
move 50 steps
pen up
```



Now let’s see how to turn this into a custom block, so that instead of having to 10-block sequence every time we want to draw a square we can just use our “draw square” block.  There are two ways to define a new block:  One is to right click on the scripts pane of any sprite and select “make a block,” and the other is to click the “Make a block” button in the “Variables” category of the blocks palette. Both of these methods will cause the following window to pop up:

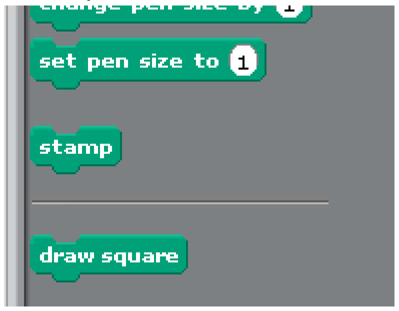


This window allows the programmer to select a category for the new block by clicking on it, and select what type/shape it should be (command, reporter, or predicate). Our example (drawing a square) draws with the pen, and is a command block, so we select those two items, keep "For all sprites" selected, type "draw square" in the text field as the name of the block, and click "OK". After doing this, the **block editor** window pops up, which has an area to build a script that starts with a hat block. There's also an "atomic" checkbox at the top, but that can be safely ignored for now (we'll get back to it later in the course). Putting our "square drawing" script in the block editor window results in the following:



```
draw square
pen down
point in direction 0
move 50 steps
point in direction -90
move 50 steps
point in direction 180
move 50 steps
point in direction 90
move 50 steps
pen up
```

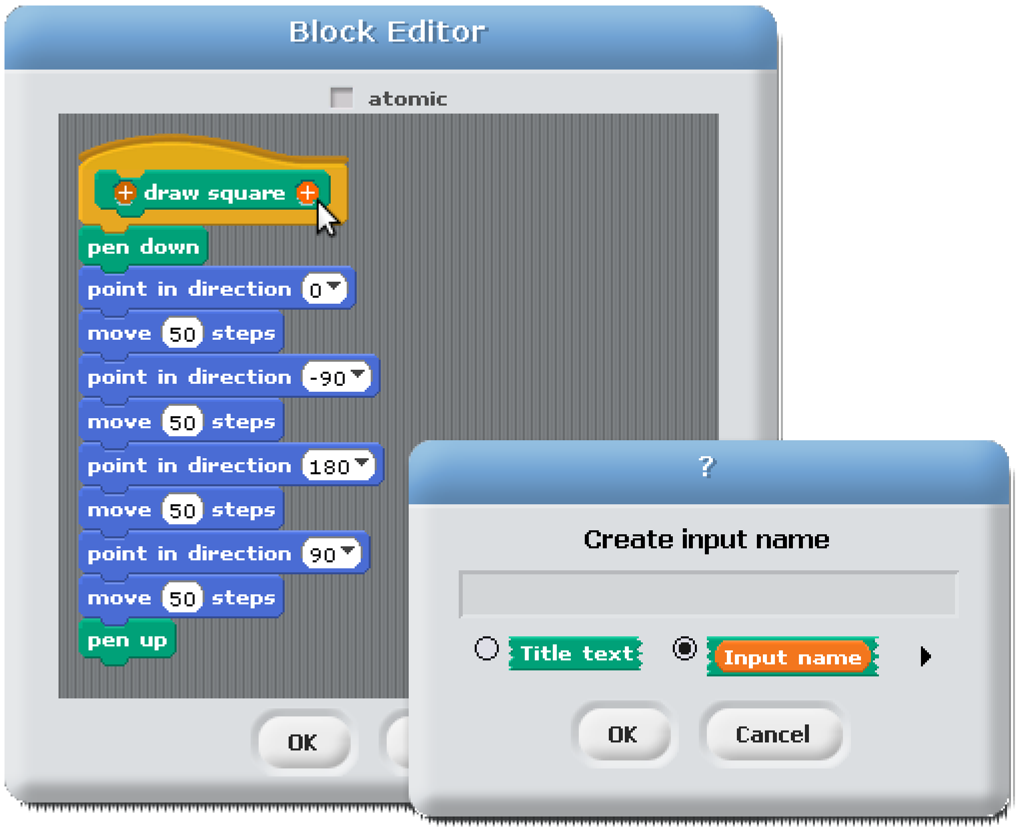
Once the block definition is finalized by clicking "OK" it will appear in the "Pen" category of the blocks palette - the bottom of the block palette now looks like this:



Our new block can dragged and used with any sprite, whenever the programmer wants to draw a square that is 50 units on a side - now that the work has been done to define the block, the multi-step process of drawing a square is abstractly represented as a single block!

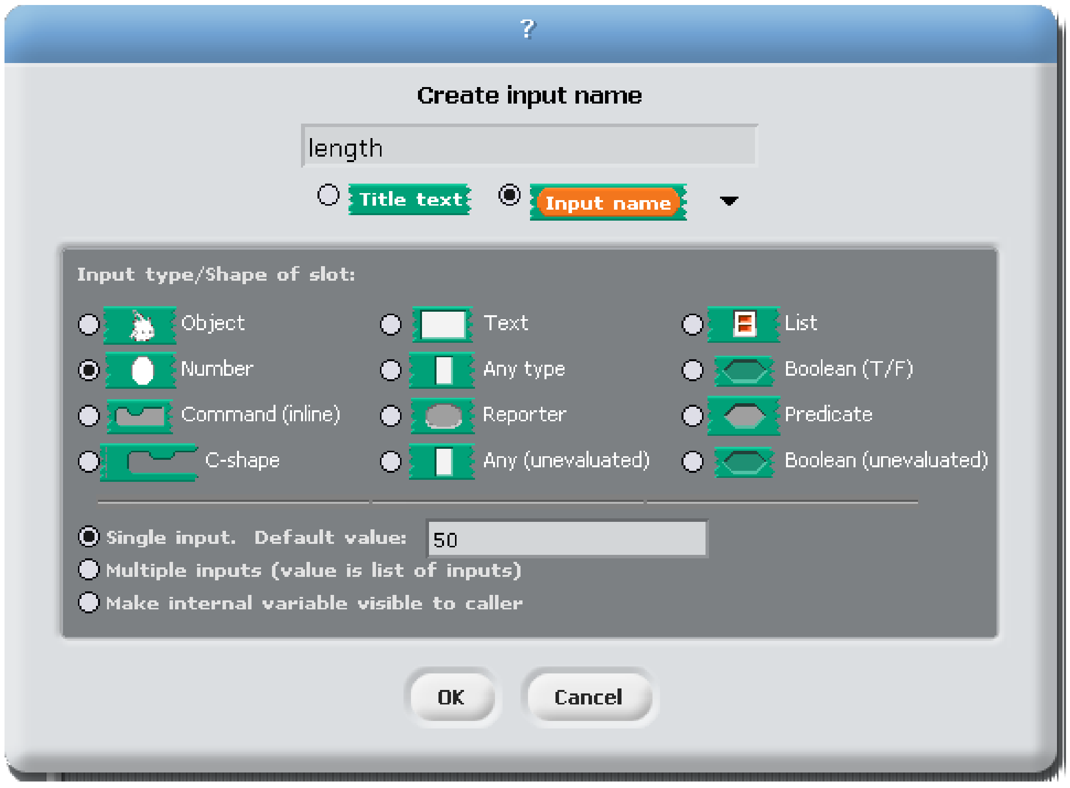
## Making Blocks With Parameters

While making a block that can “stand in” for a fixed sequence of other blocks is useful in some situations, it’s even better if we can generalize the operation so that it can do slightly different things based on a parameter.  In this case, why limit ourselves to just squares with sides of length 50? A programmer can modify the definition of any block she has defined by right clicking on that block in the blocks palette and selecting “edit” to open back the block editor.  When the mouse is over the hat block at the top of the block definition, some “+” signs appear to indicate where things can be inserted - to either add more words to the name of the block or add parameters.  We click on the plus sign at the right to add a parameter on the right and this pops up a window to create the parameter (this parameter provides input to the block, and we want to give it a name, so it’s labeled as “Create input name”):



```
+ draw square +
pen down
point in direction 0
move 50 steps
point in direction -90
move 50 steps
point in direction 180
move 50 steps
point in direction 90
move 50 steps
pen up
```

Clicking on the right arrow below the text entry field will open a window to set various properties of the parameter - we’ll give it the name “length”, set it to be a number (which makes the parameter space an oval) and give the parameter a default value of 50 (in other words, the value that appears with the block in the blocks palette until you change it).  With these properties set, this is what the parameter properties window looks like:



?

Create input name

length

○ Title text (● Input name)

Input type/Shape of slot:

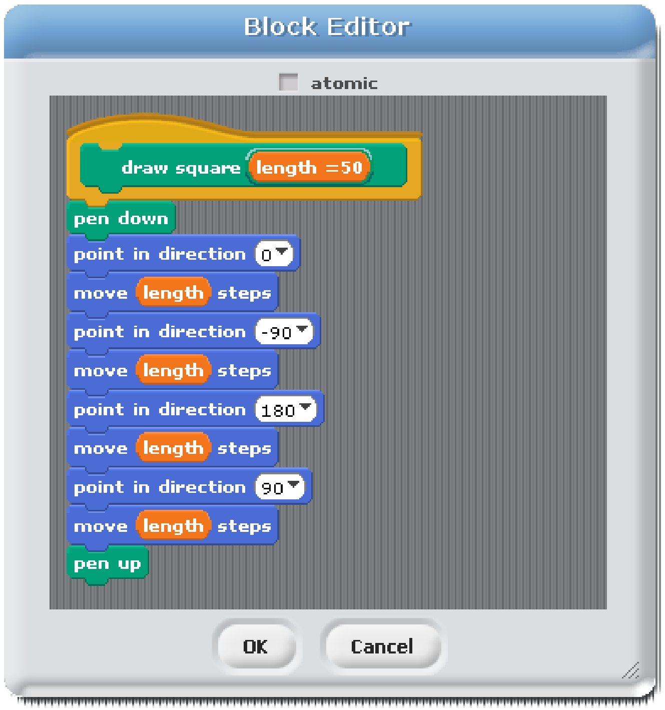
| | | |
|---|---|---|
| ○ Object | ○ Text | ○ List |
| ● Number | ○ Any type | ○ Boolean (T/F) |
| ○ Command (inline) | ○ Reporter | ○ Predicate |
| ○ C-shape | ○ Any (unevaluated) | ○ Boolean (unevaluated) |

● Single input. Default value: 50

○ Multiple inputs (value is list of inputs)

○ Make internal variable visible to caller

There's a lot more to defining parameters, but we'll save the rest of these properties for later. In the block editor window, the "length" parameter appears in the hat block, and it can be dragged to other places as if it were a variable (or a reporter block). We'll drag it down to each "move block" to end up with this block definition:

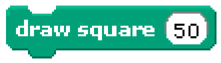


Block Editor

☐ atomic

```
draw square (length =50)
pen down
point in direction 0
move length steps
point in direction -90
move length steps
point in direction 180
move length steps
point in direction 90
move length steps
pen up
```

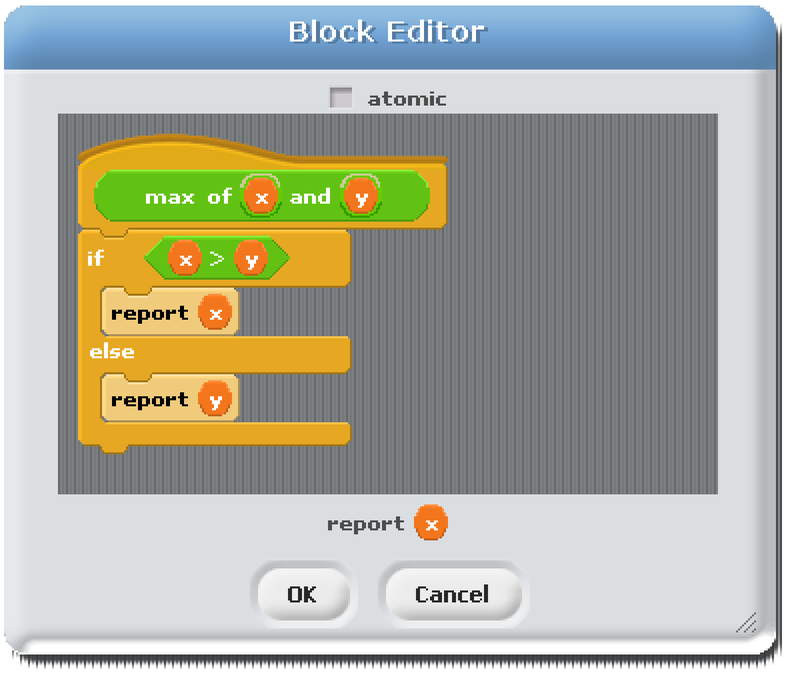
After these changes, the “draw square” block appears in “Pen” category of the blocks palette like this:



Now it’s possible to change the parameter to draw different size squares with a single block! The other important part to notice is that this block looks like all of the other blocks that come built-in to BYOB - there’s really no difference between one you define from a sequence of other blocks, and a block that is defined by the system.

## Examples of Reporter and Predicate Blocks

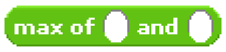
There are several functions that are useful, but are not provided by BYOB.  For example, the maximum function, typically written as “max(x,y)” in mathematics, is useful in many situations, but is not provided as a block. As an example of defining a reporter block, we will define a block that reports the maximum of two numbers.  We start the definition exactly as before, but add two parameters and some additional title text.  Like the built-in operators of “+”, “-”, etc., we won’t give the parameters default values.  The definition looks like this:



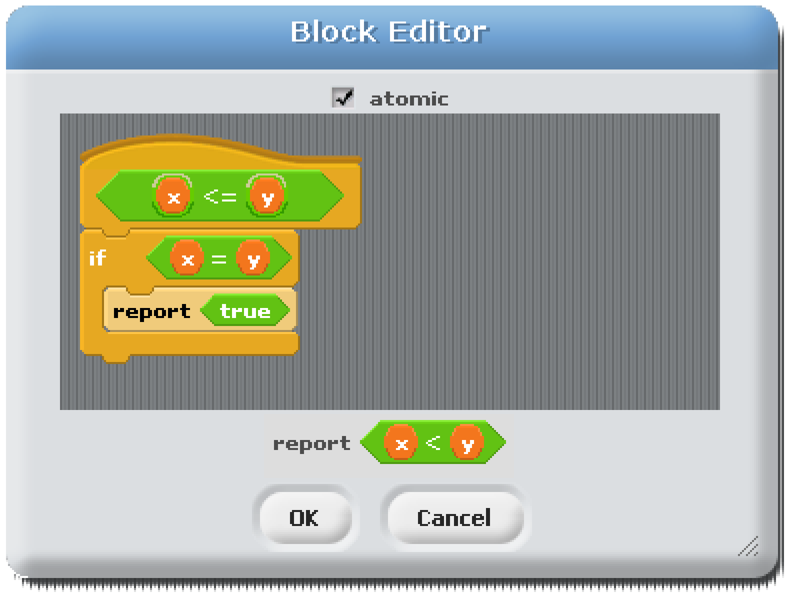
```
max of x and y
if x > y
    report x
else
    report y

report x
```

Notice the “report” blocks that are used in the script and the “report” statement at the bottom of the window.  The internal ones directly report a value (if x is larger it reports x; otherwise, it reports y).  The “report x” at the bottom is the default return value - if execution “falls off” the bottom of the script without a report block being found, this is the value that is reported.  Notice that in our block there is no way this could happen, since we handle all cases directly with “report …” blocks, but it is always a good idea to put a value here “just in case.”  Now we have a max block that can be used just like any other operator block, which appears in the blocks palette like this:



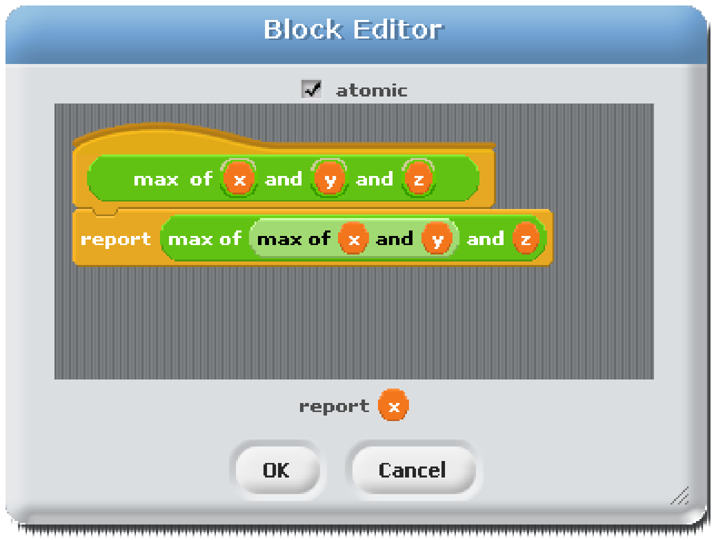
Predicate definitions are the same, but you have to select “predicate” when making the block so that it has the right shape. The following definition is an example of defining a predicate for “less than or equal”:



```
x <= y
if x = y
  report true
report x < y
```

The use of the default “report” value at the bottom is trickier than the previous example, and is vital to the correct functioning of this definition. It is worth your time to think this through until you understand how this definition works.

We’ll illustrate one more point before wrapping up the background reading section. Above we said that there is no difference between blocks you define and blocks that are built into the system. This includes using blocks in definitions! We will define a block that reports the maximum of three numbers using the block we defined for the maximum of two values: we take the maximum of the first two values, and then the maximum of this value and the remaining input value. Here’s the definition - make sure you understand how this works:

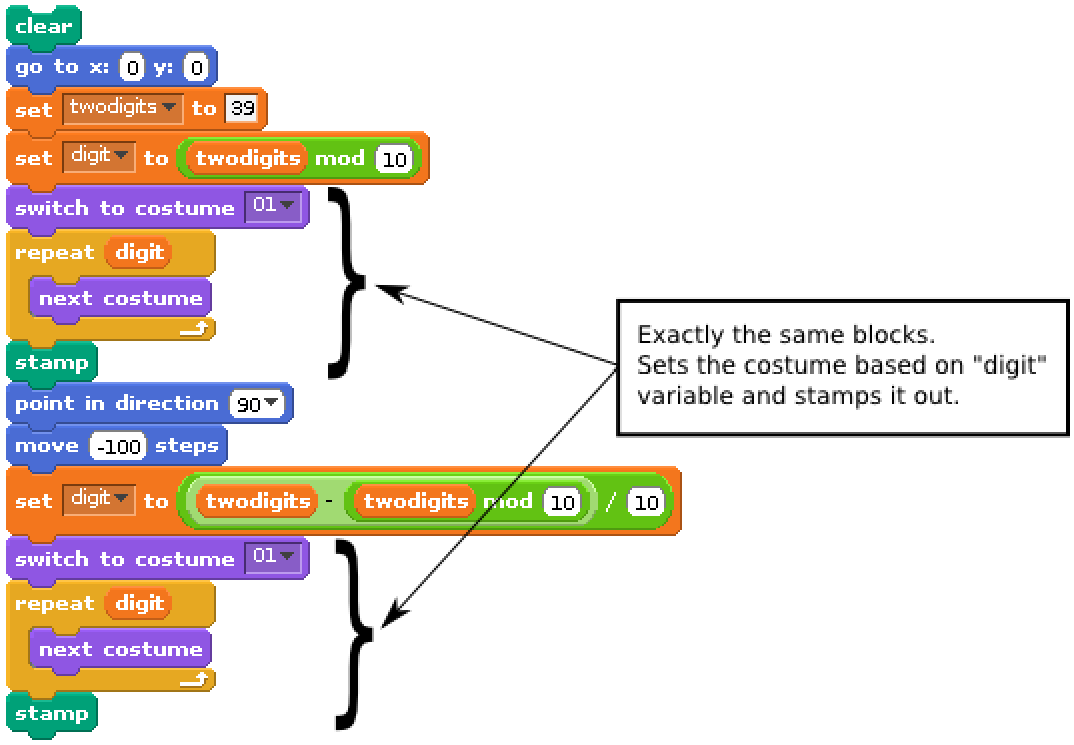


```
max of x and y and z
report max of (max of x and y) and z
report x
```

You have now seen examples of how you can “Build Your Own Block” to create a command, reporter, or predicate block. The activities you will do in the lab give you practice with this, making blocks that allow you to create a more elegant solution to the number stamping script from the previous lab.

## Activities (In-Lab Work)

**Activity 1:** A solution to the “number stamping” activity in the previous lab is shown below:



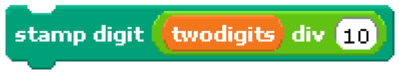
```
clear
go to x: 0 y: 0
set twodigits to 39
set digit to (twodigits mod 10)
switch to costume 01
repeat digit
    next costume
stamp
point in direction 90
move -100 steps
set digit to ((twodigits - (twodigits mod 10)) / 10)
switch to costume 01
repeat digit
    next costume
stamp
```

Exactly the same blocks.
Sets the costume based on "digit" variable and stamps it out.

Your solution may or may not have looked quite like this, but you probably had a section of your script that stamped out a single digit, and if you followed the directions (what a concept!) that section will be repeated twice in your script. For your first activity in this lab, create your own block named “stamp digit” that does the tasks marked above (sets the correct costume and stamps the digit) - it should take the digit to stamp as a parameter, and then you should change the main solution to use your new block. As a starting point, you can either use your solution from the previous lab, or you can use the solution shown above. Once you have the basic functionality correct, think about improvements - for example, do you want to include “hide” and “show” blocks so that the stamping process looks nicer? For full credit on this part you must do more than just the minimum - make it look nice!

When you have completed this activity, save your work as “Lab3-Activity1”.

**Activity 2:** In the last lab, we commented that while BYOB provides a block that gives the remainder after a division, it doesn’t have a block to report the integer quotient. Let’s correct that oversight! In the solution given above, we made a formula that uses subtraction, mod, and division to get the right answer. You should define a block named “div” that abstracts this process so that you can use this block to find an integer quotient. This will be a reporter block that takes two parameters just like “mod”, but returns the quotient rather than the remainder. The reason for this abstraction isn’t that the formula is particularly complicated, but it’s nice that we don’t have to see the details - it allows us to focus on the “big picture” of what we want to do rather than get bogged down in a messy formula. You can now use this block directly as the argument to the “stamp digit” block you created in Activity 1, so stamping the second digit your code might look something like this:

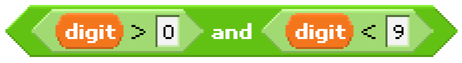


```
stamp digit (twodigits div 10)
```

Once you have this block defined and have modified your overall solution (stamping two digits) to use it, save the result as “Lab3-Activity2”.

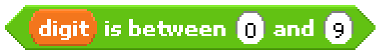
**Activity 3:** In this activity you will make the “stamp digit” block more robust using a block you create. In programming, we say that code is **robust** if it works acceptably no matter what inputs are provided to it - even ones (or maybe especially ones) that are not valid inputs. In the block we defined for “stamp digit,” what would happen if we called this with an argument of 50000? What about with an argument of -6? Try it - but be prepared to hit the red stop sign over the stage if you try it with an argument of 50000! Those aren’t valid inputs for a procedure that expects a single digit in the range 0..9, but you should consider this possibility because... well... stuff happens. And you don’t want such stuff to crash your entire program, just because one thing messed up somewhere.

The easiest way to handle this is to simply check at the beginning of the block whether the inputs are within the bounds of what they should be. This is called **input checking**, and incomplete (or missing) input checking is far and away the leading cause of instability and security problems in software. This is a good skill to practice and use regularly! In our case, we want to make sure that the input is in the range 0..9 - in other words, the input needs to be greater than or equal to 0 and less than or equal to 9. When we want to combine two conditions (i.e., two predicates) like this, we can use a **Boolean operator**. Boolean operators are things like “and”, “or”, and “not”, and their use corresponds to regular English usage (we’ll say a little more about Boolean operators in the “Discussion section” below). We used “and” when describing what we want to do in English, so we use “and” in the condition that we test. The following example shows *almost* what we want to do, but not quite (in other words, the following code is not correct - make sure you understand that, and don’t just copy this code!):



```
<<(digit) > [0]> and <(digit) < [9]>>
```

Your task for this activity is to create a predicate block that does a test like this, but does it correctly: testing whether a value is in a specific range (including the endpoints in the range). You should use your block inside your “stamp digit” block to make sure that block works reasonably, regardless of the inputs provided - using your block might look something like this:



```
<(digit) is between (0) and (9)>
```

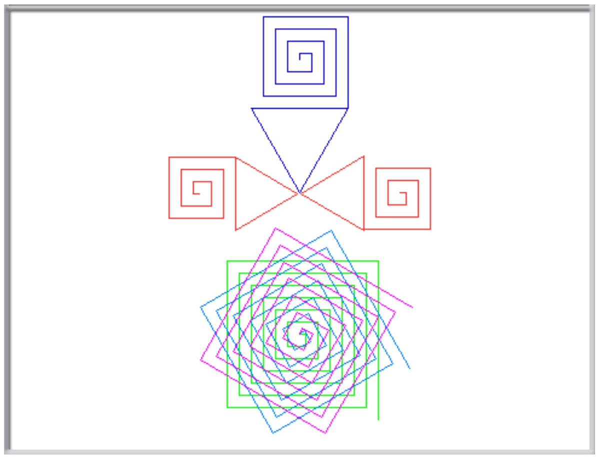
There are a few decisions you'll have to make when defining your block: Do you want to define additional blocks for "less than or equal to" and "greater than or equal to", or do you just want to put the logic for these tests directly into the "... is between ..." block? When you use this block in the "stamp digit" script, what do you want to happen if the test fails? You could stamp a 0, or even have a special character like an "X" that you stamp. Deciding what to do when you are given invalid inputs is called **error handling**, and it can sometimes be quite tricky to decide what you want to do!

If you do what was just described, you can get most of the credit for this activity. To get full credit, you need to make one additional adjustment to the script that stamps both digits: you should handle digits at different sizes. In other words, you should be able to set the size to 20% and stamp out the number (two digits) correctly at that size. It almost works if you just put in a block to set the size at the beginning; however, if the "move …" goes 100 pixels to the left, you'll end up with too much space between the digits. An important point: You should have only one place in your code where you set the size - you should not require a change in the programming for the "move …" block when you change the size at the top of the script. The key is the "size" variable in the "Looks" category, which allows you to use the size that was set at the top of the script when you figure out how far to move.

When you finish this activity, save your work as "Lab3-Activity3".

**Activity 4:** *This activity does not build on the previous ones - after you're sure the previous activity has been saved, start a new project in BYOB to clear out your previous work.*

For this activity, you should show some artistic creativity! You should make a picture out of geometric shapes that you draw on the stage. Here are the requirements: You should create at least two different custom blocks, drawing shapes that can vary using a parameter (the most obvious parameter is size, but maybe there are other options that you can probably come up with), and then you should use your blocks to draw multiple copies of each shape in various positions. By defining your own block, you don't have to make redundant scripts, so hopefully your code will be clean and elegant! I'm not expecting artistic masterpieces here - just make it interesting, with varying shapes and colors. Here's an example:



This is made up of two basic shapes:  a triangle and a squiral (that's a combination of a square and a spiral).  Both shapes were defined as blocks that took a size parameter, and then another block was defined that drew a squiral with a triangle on top, looking a little like a house (so this block definition used the other blocks that were created).  Finally, the main script drew three "houses" in two different sizes and colors, and then just to be more interesting drew three different squirals on top of each other, with angles varying by 30 degree increments.

Don't just copy this example - be creative!  Try different shapes - make pentagons, or squares, or octagons, or octirals (think about it) or...  the sky's the limit!

When you finish this activity, save your work as "Lab3-GeometricArt".

## Discussion (Post-Lab Follow-up)

**Parameters versus Arguments:** In Lab 1, we introduced the terms "parameter" and "argument" for values that affect how a block works, and said that we would explain the subtle difference in these terms later.  Now that you have seen how to define blocks, we can describe this difference.  The difference is subtle, but not difficult:  parameters appear in block definitions, and arguments are what you have when the block is used by a running program (another way to say this is "when the block is **called**"). For example, in the "max of (x) and (y)" block defined earlier, x and y are parameters, but when it is used you have something like "max of (9) and (3)", where 9 and 3 are arguments.  Another way to view this is that parameters are defined by the programmer when the block is defined, and given names which do not change.  However, when a running program uses a block it might provide different values for the arguments each time the block is used.  In some situations when you are discussing blocks it is unclear whether you are talking about an argument or a parameter, and in such cases either term will usually be fine.  But as a general rule, if you are talking about ***using*** a block you are talking about arguments, and if you are talking about a block's ***definition*** you are talking about parameters.

**Input Checking and Robustness:**  When you design a block and expect only certain inputs to be sent to the block, it is important to be clear about what you are assuming, and it's even better to enforce that the inputs are legitimate.  For example, if the "stamp digit" block is used correctly, it will always be sent an integer in the range 0 through 9, inclusive (when we say a range is "inclusive," it means we include the endpoints, so 0 and 9 are valid values). Unfortunately, the way we wrote the block definition, if the person using the block doesn't send a proper value then the block can go off into a long loop that disrupts the functioning of your program. It's also not entirely clear what will happen in this block if the argument is not an integer.  For example, what will happen if a program calls the "stamp digit" block with an argument of 2.5? The fact that a proper value wasn't sent doesn't necessarily mean that the person providing the input (or writing the program that provides the input) was being malicious or even careless - the argument to the block might be some formula that is complicated enough that it's easy to miss certain cases that give invalid values for the argument.

The two biggest problems that create security vulnerabilities in software today are "buffer overflows" and "SQL injection," both of which are caused by not checking inputs.  A buffer overflow occurs in languages where storage for certain kinds of variables, like strings or arrays, is pre-allocated and of fixed size. Let's say you allocate 100 bytes of memory to store a parameter passed to a function, and yet the function is passed 200 bytes of data - what do you do? In programming languages like C and C++, if you don't do anything to stop it, the program will happily go on writing into memory past the end of your allocated space, writing over anything that just happened to be stored in memory after that. This is called a buffer overflow, and is simply a matter of not checking inputs to make sure they fit into the space that is allocated.

"SQL" stands for "Structured Query Language," and is the most common language for querying a database. SQL injection is a problem with certain web applications that use a database, and occurs when a user types something into a web-based form (like a name or comment or login id or... really anything), and instead of the input being a simple string of characters that the programmer was expecting, the user puts in special characters that inject SQL commands into the query. Now what should be passive data turns into the remote user actually sending programming commands to the database, which can cause all sorts of problems. Again, notice that this is a case of the programmer not checking an input for these special characters that trigger an SQL command, and since web applications on the Internet are accessed by remote users who may be malicious, all web application programmers should be very good about checking all inputs that comes from remote users.

In most programming languages (certainly all industrial-strength programming languages) you can add comments to function definitions, which leaves a note to anyone using the function explaining what you expect. If you expect some condition to be true before the function is called, then you can make clear by documenting this function **pre-condition** with a comment. You shouldn't completely rely on a note left for other users about what you "expect" to hold about the inputs, and you should always check inputs to make sure they are valid, but documenting pre-conditions is very good practice so that users of a function or block are aware of your expectations with the function/block. Unfortunately, the current version of BYOB does not have a way to put comments in block definitions, so you can't do this - hopefully this oversight will be corrected in future versions of BYOB!

**Boolean Operators:** Boolean operators are operators that work on truth values: true or false. The term "Boolean" (also used in "Boolean Algebra" and "Boolean Logic") comes from George Boole, a mathematician who lived in the 1800's and developed much of the logic that is used in computers today. The main Boolean operators provided by BYOB are "and", "or", and "not". We can define what these operators do using **truth tables**, which give the value of a Boolean operator for any possible value of the inputs. For example, the truth table for the "and" operator is shown below:

| ***x*** | ***y*** | ***x AND y*** |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

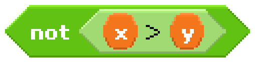
So if x is true and y is false, then “x AND y” is false. In BYOB terms, if the “and” block is called with the true as the first argument and false as the second argument, then the “and” block reports false. In fact, the only way for “x AND y” to be true is if both x and y are true - matching the common English understanding of the word “and.”

The truth table for “or” is as follows:

| ***x*** | ***y*** | ***x OR y*** |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

This matches our intuition about the word “or” except for one possible case. In some cases, the English language use of “or” means one thing or another, but not both. For example, I might say “I will go to the movies or I will study,” which means that I will do exactly one of those actions, but not both. In a Boolean “or” the answer is true if either of the arguments is true but it is also true if both arguments are true! There is a special kind of “or” that more closely matches the “either-one-or-the-other” meaning, and in Boolean logic that is called an **exclusive or**. While the exclusive or is very useful in some situations, it is not provided by BYOB and is not something we will need in this class.

Finally, the “not” operator takes a single parameter and flips it: true becomes false, and false becomes true. This can be very useful in some situations. For example, while we noted in the activities above that there is no “less than or equal to” in BYOB, the following is entirely equivalent to a predicate that tests if x is less than or equal to y - can you see why?



Boolean operators can be very powerful, but the basics are simple. If you study discrete mathematics or logic you will learn much more about how these operators work together.

## Terminology

- *block editor*: A window in BYOB that provides space to define a script that executes whenever a block is called.
- *Boolean operator*: An operator that works on true/false values, such as “and”, “or”, and “not”.
- *calling a block (or function)*: This means that a running program executes the actions that make up a block - for example, when a running program reaches the “draw square” block, we say that it “calls draw square to do the actions in the draw square definition.”
- *command block*: A block which represents a sequence of actions to perform, but does not report a value.
- *error handling*: Code which is executed when something unexpected happens in a program, such as a parameter having an illegal value.

- *exclusive or*: A special form of the “or” Boolean operator that is false if both arguments are true (unlike a regular Boolean “or” which is true in that situation).
- *function*: In general, a function takes arguments and produces a value, and is a general term that applies to BYOB reporter and predicate blocks. Since most programming languages are not built around “blocks” like BYOB, this is a more common term in computer science. Note that in some programming languages, it is possible for a function not to return a value (a type of function which is sometimes called a procedure).
- *hat block*: A type of block in BYOB that starts off a script, and can either be based on an event that triggers the script or can be at the top of a programmer-defined block.
- *input checking*: Testing whether values that are provided are acceptable, where the input can come from arguments provided to a block, typing from the user, or data received from the network.
- *predicate*: A special type of reporter block that always reports either true or false.
- *pre-condition*: A condition that is expected to hold before a script (typically a function or block definition) is executed.
- *procedure*: Using the more common “function” terminology, a function that does not return a value is often referred to as a procedure.
- *reporter block*: A block that provided a value when it finishes execution, where the value is typically a number or a string (if it is a Boolean value, the block is referred to as a predicate).
- *return a value*: In “function” terminology, this is a value produced by a function, just as we would say a block “reports” a value in block terminology.
- *robust*: A program that doesn’t do unexpected things, even when provided unexpected or illegal inputs.
- *subroutine*: Synonym for “procedure” - a function that does not return a value.
- *truth table*: A table that shows the value of a Boolean expression for all possible values of inputs.

## Submission

In this lab, you should have saved the following files: Lab3-Activity1, Lab3-Activity2, Lab3-Activity3, and Lab3-GeometricArt. Turn these in using whatever submission mechanism your school has set up for you.