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 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"*
Copyright © 2012-2014 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[2], 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:

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 reported value. Since they are used inside blocks, rather than snapped on top

[2] The latest version of Scratch also supports building your own blocks, but not in as general or powerful a way as BYOB.

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:

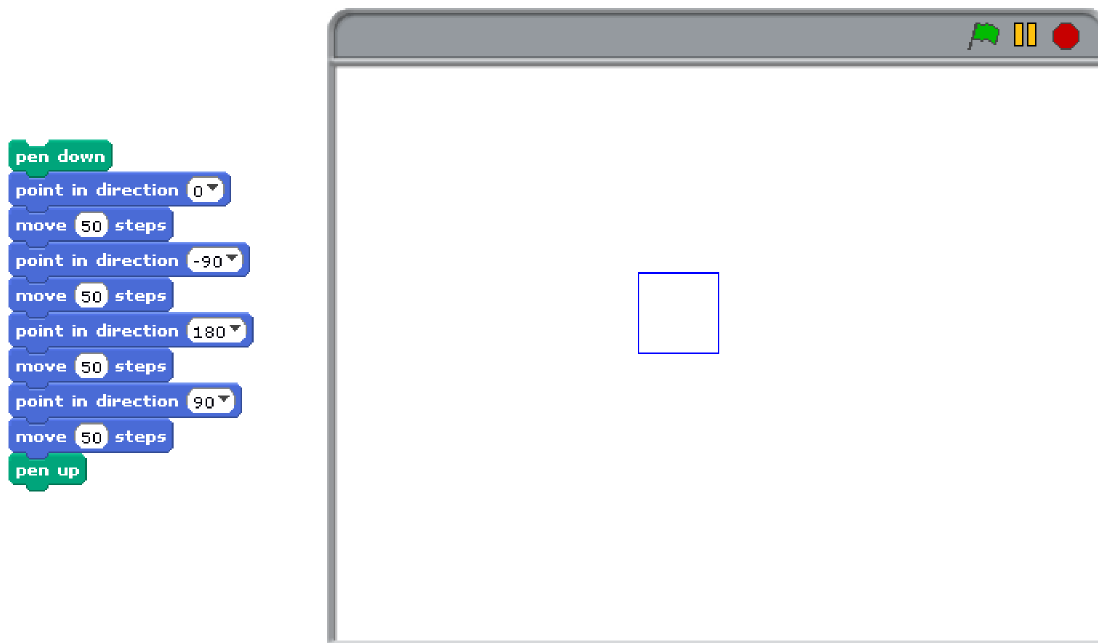
In the last lab you saw that a reporter block will "say" what its value is when you click on it (in the pre-lab reading, this was shown with the "mod" block). Since a predicate is just a special form of a reporter block, this will work with predicates the same way, but – being predicates – they will only say "true" or "false."

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, we usually say that a function will **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.

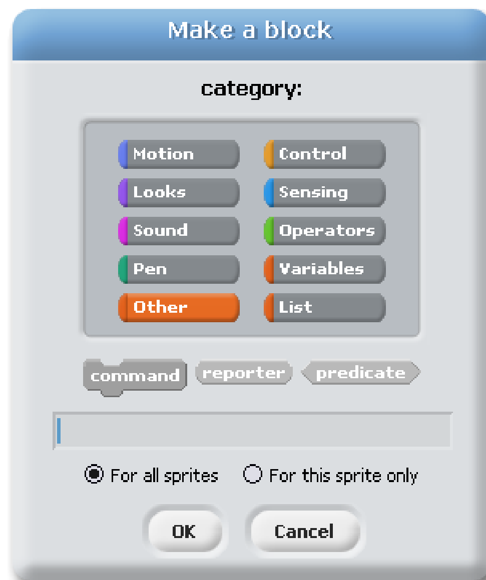
Building a Simple Command Block

In this lab you will create your own command blocks – we'll postpone creating reporter and predicate blocks until the next lab. As an example, let's see 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:



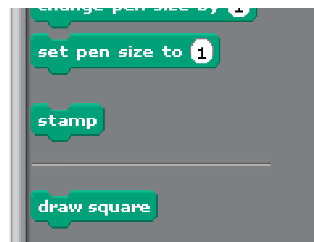
Now let's see how to turn this into a custom block, so that instead of having to create a 10-block sequence every time we want to draw a square we can just use our "draw square" block. There are two ways to define a new block: One is to right click on the scripts pane of any sprite and select "make a 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, 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:

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:

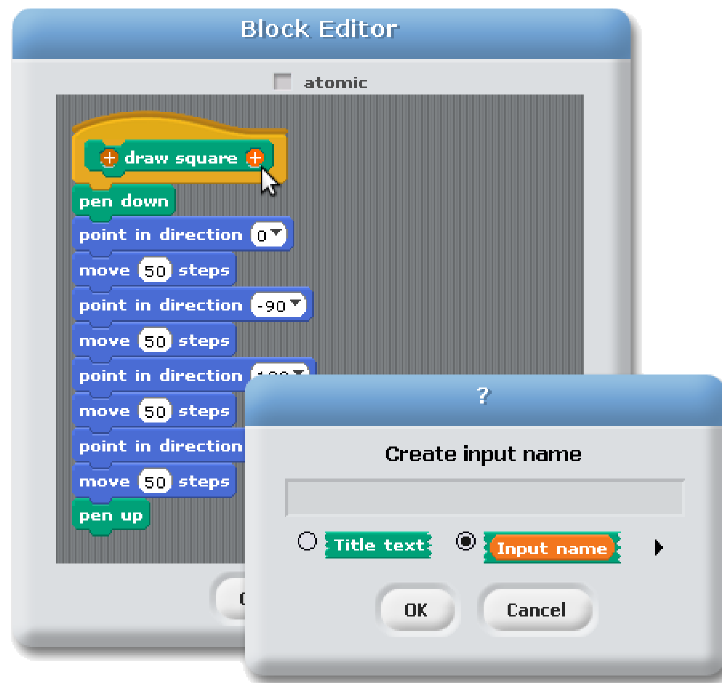


Our new block can dragged and used with any sprite, whenever the programmer wants to draw a square that is 50 units on a side – now that the work has been done to define the block, the multi-step process of drawing a square is abstractly represented as a single block!

Making Blocks With Parameters

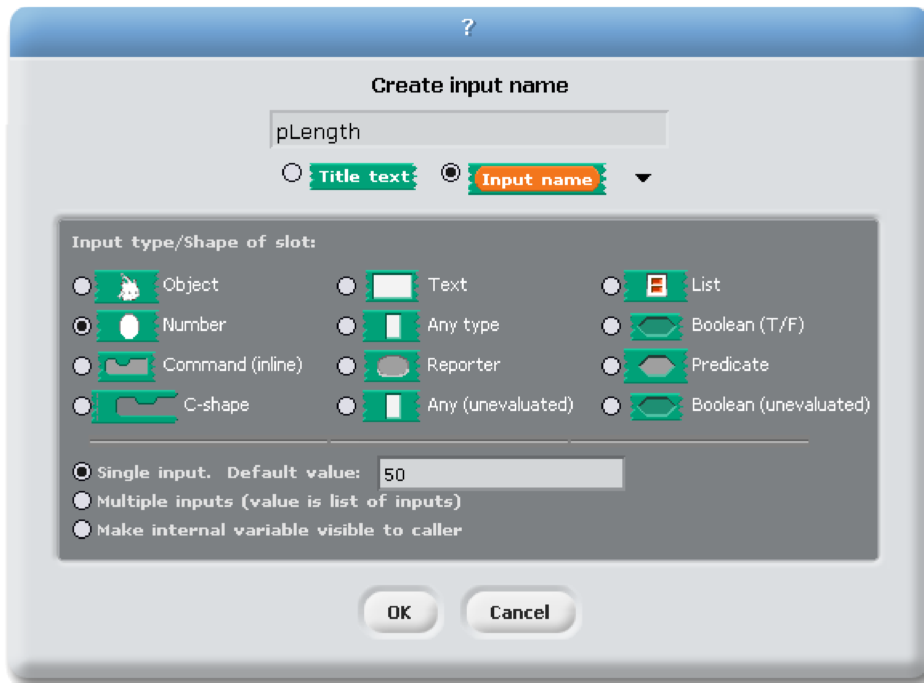
While making a block that can "stand in" for a fixed sequence of other blocks is useful in some situations, it's even better if we can generalize the operation so that it can do slightly different things based on a parameter. In this case, why limit ourselves to just squares with sides of length 50? 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"):

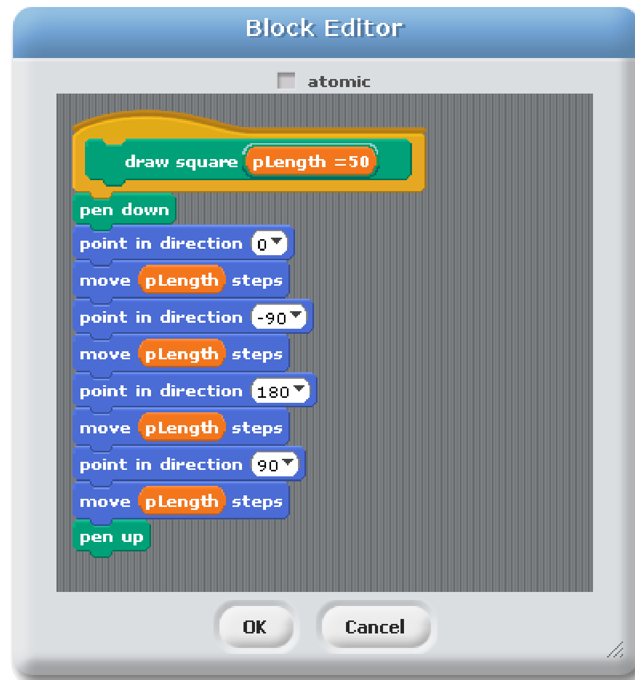


Clicking on the right arrow below the text entry field will open a window to set various properties of the parameter (see the screenshot below) – we need to give the parameter a name, and we do this according to some rules that we define. Such rules are referred to as "**naming conventions**" and are used to make programming easier and less error-prone. A common naming convention is to prefix a variable name with a code that indicates something about the variable – in our case, we will prefix every parameter name with a lowercase "p" (for "parameter") and then start the variable name (reflecting what it stands for) with a capital letter. Since this parameter represents the length of a side of our square, we'll call it "pLength." When we talk about other kinds of variables in a later lab, we'll return to the issue of naming conventions, but for now just remember to always start every parameter name with a "p" and to never name a non-parameter variable in this way. In addition to naming the variable, I have indicated that it will be a number (which makes the parameter space an oval) and gave 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:



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 "pLength" 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:



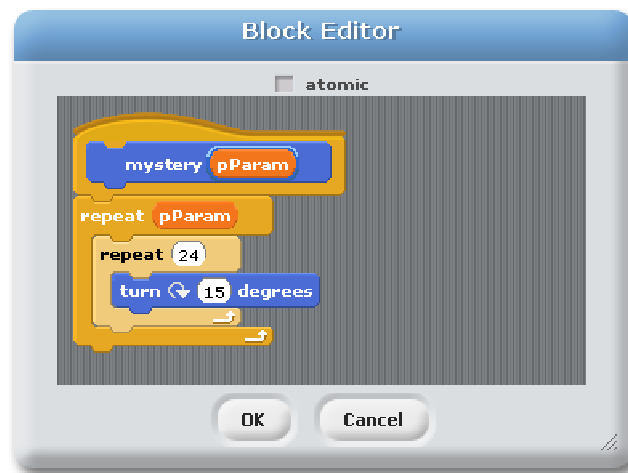
After these changes, the "draw square" block appears in "Pen" category of the blocks palette like this:

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, and a block that is defined by the system.

Self-Assessment Questions

Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. Why is there no tab at the bottom of the "forever" block?
2. Name the three basic types of blocks in BYOB.
3. If you wanted to create a block that played an animation, like a character doing a "You win!" dance, what type of block would you use?
4. If you wanted to define a block for a sprite that indicated whether or not it was an enemy in a game, what kind of block would you use?
5. Below is the definition of a "mystery" block – what does it do (a high-level description, not a step-by-step description)?

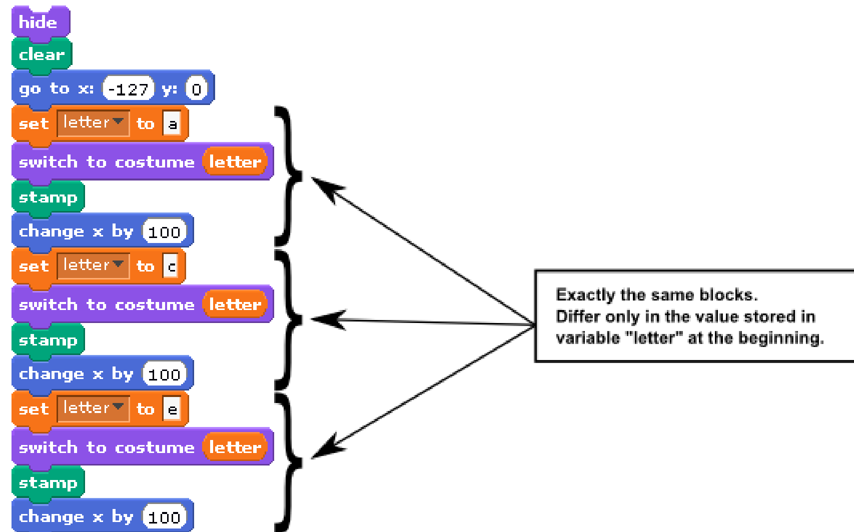


6. The block and parameter names in the previous question were purposely vague, which is not what you want to do when you are really programming. How could you change the definition so that it makes more sense to both a user of the block (the name of the block is most important for this) and to programmers understanding how the block is implemented (names of parameters and variables are important for this)?

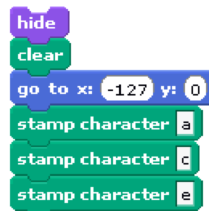
Activities (In-Lab Work)

Activity 1: Making a general-purpose letter-stamping command block.
Project name to use: Lab3-Activity1

A solution to the “stencil graffiti” activity from the previous lab is shown below:



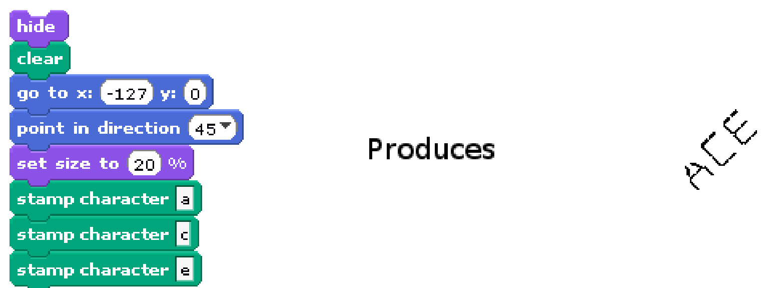
Your solution may or may not have looked exactly like this at the beginning (the hide/clear/goto sequence), but if you followed the directions then the rest should have looked like this. The only part of the repeating four-block sequence that might be different in your code are the values assigned to the “letter” variable (to write out different words) and the value in the “change x by …” block, which depends on the style of letter you chose. For your first activity in this lab, create your own block that does the common tasks marked by the brackets above, and name it “stamp character.” Your block definition should should take the character to stamp as a parameter, and then it should set the correct costume, stamp the character, and change x by the appropriate amount. *Important: Remember the “naming convention” for parameters that was described in the pre-lab reading – follow the naming convention when you give your parameter a name!* If you do this correctly, you can (and should!) change your main program so that it just looks like this:



Isn’t that a lot easier to read and understand than the original solution? To write out a word now, you don’t need to worry about costumes, or the amount to move, or any of the details that aren’t necessary to what you really want to do. That’s part of the power of abstraction!

There is one more thing to do in this activity: Make it so your “stamp character” block can write at any size and any angle. The size of a sprite can be adjusted using the “set size to …” block in

the “Looks” category, where “100” percent is the natural size. If the costumes tab says that the costume is 80x80, then at 100% size it will be displayed as 80 pixels by 80 pixels. On the other hand, if you set the size to 20%, then the sprite will shrink and be drawn as 16 pixels by 16 pixels. Your goal is to make it so that you can do something like this in your main program, with the result shown next to the code:



Notice that we set the size to 20% and the direction to 45 before calling “stamp character” so the output is smaller and angled up at 45 degrees. To accomplish this, you need to edit the “stamp character” block definition, so right click on it and select “edit.” All you need to do is replace the “change x by …” block with a “move” block, since that moves in the current direction, and then calculate the distance to move based on both the costume width and the current size (notice that there is a reporter block in the Looks category that gives the current size). *Important: Don’t “cheat” by making assumptions about the distance to move. For some letter styles the formula can be very simple, but for others it cannot, so make it as general as possible. Your distance should use both the multiplication operation (for scaling) and the division operation (for turning the size percentage into a fractional amount) – if you haven’t used a multiplication block and a division block, then you haven’t done it correctly!*

Finally, notice that this part of the activity illustrates another advantage of abstraction. If you were using the original code, with three copies of everything, to allow for different sizes and directions you’d have to change your code in three different places. Since you put all of the code for stamping and moving in a single definition, you can make a single change, and it takes effect for all three of the “stamp character” calls.

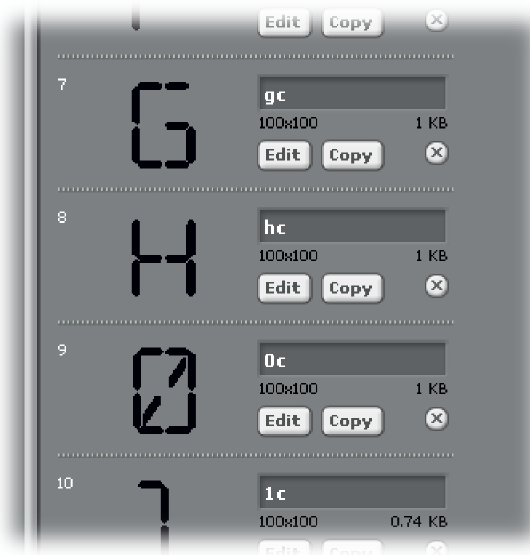
When you have completed this activity, make sure you save your work as “Lab3-Activity1”.

Activity 2: Stamping strings of characters.
Project name to use: Lab3-Activity2

In this activity you’ll add the ability to stamp numbers as well as letters, and then see how we can use iteration to simplify stamping out words or multi-digit numbers. Unfortunately, there is one potential difficulty: if you picked the “outline” style of characters for your work to this point, there are no digits in this style (I don’t know why not!). If you’re using the outline style, just use the digits from the “funky” style, since they are roughly the right size. Whether you need to make this style substitution or not, get started by going to the Costumes tab and importing the digit 1. What is the name of this costume after the import? Try changing the name to “1” so that the name is the same as the digit. What happens?

Unfortunately, there is some ambiguity in BYOB: Costumes can be referred to by name (such as "a") or by number (costume #1 is probably the "a" costume in your costume list, if you have followed the instructions in the last lab). To try to avoid this confusion, BYOB will not let you name a costume with a single digit number; otherwise, if you called "switch to costume 1" it wouldn't know if you meant costume #1 or the costume named "1". Unfortunately, since it *does* let you use two-digit costume names, if you have more than 10 costumes you will still have this problem with ambiguous meanings.

Here's a solution that avoids this problem: We will change the names so that every name ends in the letter "c" (for "character"). So the name of the costume for letter "a" should be "ac" and the name for digit "1" should be "1c". Since all the names now contain a letter, they won't be confused for costume numbers. You should go through all of your costumes and change the names as was just described. Then finish importing all of the digit values, and give them the appropriate names. When you are done, your Costumes tab should contain something like this:

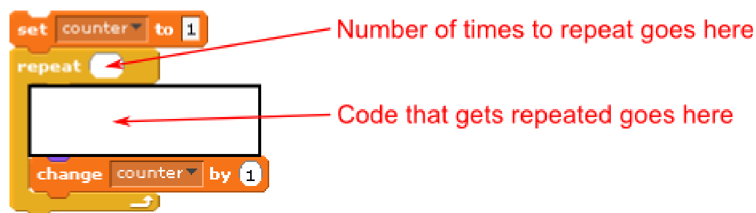


Now try running your 3-letter word stamping program again. Does it work? Do you know why not? Think about it for a little bit to see if you can figure it out before moving on.

The problem is that when we changed our costume names, the "switch to costume" block inside the "stamp character" definition no longer works. If we sent it the character "a" then "switch to costume a" won't work because there is no costume named "a" any more! The solution to this problem is to stick a "c" on the end of every character value before using it as the argument to "switch to costume," and there is a "join" reporter block in the Operators category that does exactly this: it joins together two values so one is after another. So in particular, if the character I want to stamp is passed in to "stamp character" as a parameter named "pChar", then consider the following modified version of the "switch to costume" call to see how this works:

With that change made, now try stamping numbers out instead of letters (so from your main script, send digits instead of the letters ‘a’, ‘c’, and ‘e’ that we’ve been using in our examples). Hopefully both letters and numbers should stamp out correctly now.

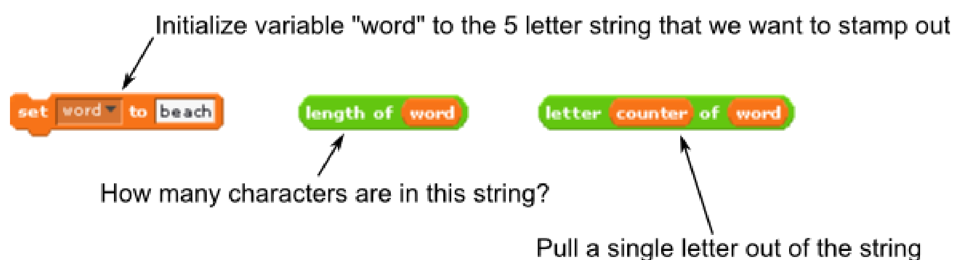
So far in this activity there has been a lot of explaining, but not much to do. The remainder of the activity will introduce a new pattern of code that is very commonly used in programming. A **pattern** is like a “fill-in-the-blanks” piece of code that is useful in a lot of different situations, and so it is in fact just another form of abstraction. In the first pattern that we’ll see in this class, we simply want to count inside a loop in order to keep track of how many iterations of the loop have been performed. We will have a counter variable that has the value 1 the first time through the loop, then has the value 2 the second time, and then 3 the third time, and so on. We call this the **counter pattern**, and here’s what it looks like in general:



If we filled in “10” in the repeat block and had a “say counter for 1 second” block inside the loop, then the sprite would say “1” for 1 second, then “2” for 1 second, then “3” for 1 second, and so on, until it says “10” at the end.

In computing, a sequence of characters is called a **string**. Putting a 5-character word into a string is much easier than creating 5 separate calls with one letter each, and the final goal for this activity is to use the counter pattern to go through the string to stamp out one character at a time. The first time through the loop we want to stamp out the first letter from the string; the second time we want to stamp out the second letter; the third time letter number 3; … Hopefully you can see how the structure of what we want to do matches exactly with the counter pattern we described above.

Continuing with this activity, you should create a new variable named “word”, and use a “set” block to initialize it to some word like “beach”, and then loop through one letter at a time to stamp each one out. If you do this right, you’ll have a script that is structured like the pattern above, in which your “stamp character” block appears exactly once. I’m not going to give the solution to you, but here are all the pieces you’ll need, along with hints on how each one should be used:



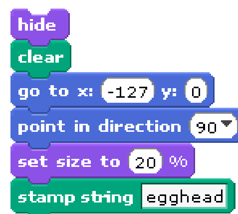
Should one of these be used in the “repeat” count in the pattern? Should one of them be used for the argument to “stamp character”? Should that be inside or outside the loop?

Once you have completed this, you should be able to stamp out an entire word that is stored in the variable “word”. Try it with different words that use the letters A through H. Try “beach” and “ahead” and “decade”. What’s the longest word you can come up with that uses just these letters? Try it with numbers too (set word to “512643”)!

Once you are confident that your script is working correctly, make sure to save it as “Lab3-Activity2”.

Activity 3: Using a block definition to define another custom block.
Project name to use: Lab3-Activity3

This activity is very simple to describe: take the loop you created in the last activity, and use it to create a new custom block named “stamp string.” The goal is hide the details of the loop so that your entire main script can look just like this:



This really gets at the heart of what you want to do: specify the position, direction, and size for the word you want to stamp out, and then let the “stamp string” block take care of the rest. If all you want to do is write out a word, why should you have to worry about costumes, iteration, counter variables, and all of that?

While the main thing to do in this activity really involves just pulling your loop inside a block definition, make sure you take care of the parameter correctly. The correct way to pass the word that you want to stamp into the block script is through the parameter – the script should *not* be accessing any variables that exist outside the block. In particular, if you are referencing the variable “word” from inside the block definition, you’re doing something wrong!

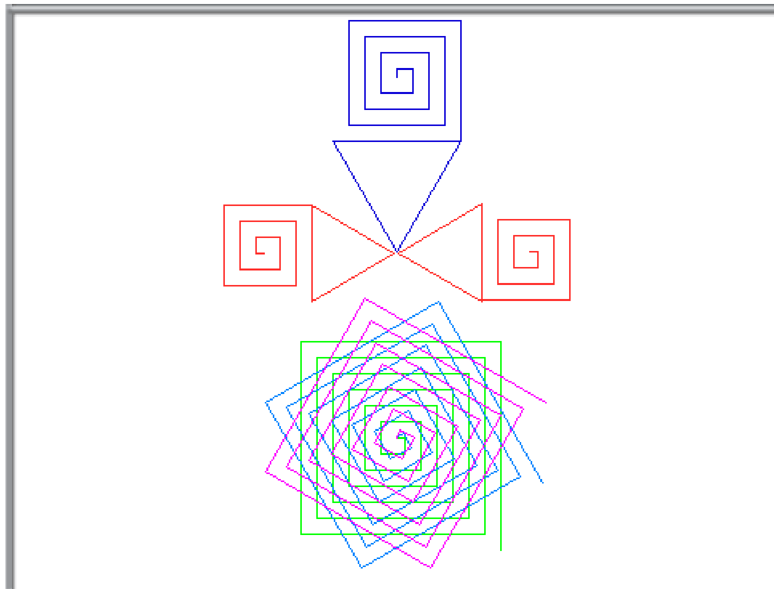
As a final note, take a little time to think about what you have done here: You started off abstracting a multi-block sequence of operations into a block definition (“stamp character”). Then you used that block definition to define a block that had slightly more complex block definition (“stamp string”). There is something deep and important going on here: more and more complex behavior is being built by combining other abstracted operations together. That’s the way complex software is built!

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

Activity 4: Using abstraction to create regular geometric artwork.
Project name to use: Lab3-GeometricArt

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".

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.

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 “draw square (pLength)” block defined earlier, pLength is a parameter, but when it is used you have something like “draw square (20)”, where 20 is an 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.

Terminology

- *block editor*: A window in BYOB that provides space to define a script that executes whenever a block is called.
- *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.
- *counter pattern:* A pattern of programming statements that includes a loop and a counter variable that keeps track of how many times the loop has been executed.
- *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.
- *naming convention*: A set of rules for naming variables and parameters that provide information to someone reading the program about what kind of variable this. Naming conventions are used to make programming easier and less error-prone.
- *pattern:* A structure of programming statements that solves a general and commonly occurring programming problem.

- *predicate*: A special type of reporter block that always reports either true or false.
- *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.
- *subroutine*: Synonym for "procedure" – a function that does not return a value.

Answers to Pre-Lab Self-Assessment Questions

1. Why is there no tab at the bottom of the "forever" block?

   Answer: The tab at the bottom of command blocks is there to fit in blocks that follow it in the execution sequence of that script – since the "forever" block never exits, there can be no "next block" that executes (what comes "after forever"?).

2. Name the three basic types of blocks in BYOB.

   Answer: Command blocks, hat blocks, and reporter blocks. These are the three basic blocks, distinguished by whether they are used internally in a sequence of operations (command blocks), whether they specify a condition for starting a script (hat blocks), or whether they report a value for further use. It is tempting to say that the three types of blocks are command blocks, reporter blocks, and predicates, since those are the three options that come up when you define your own block – however, a predicate is just a special kind of reporter block (that always reports true or false). The hat block is missing from that dialog, since there is no way in BYOB to define your own hat block.

3. If you wanted to create a block that played an animation, like a character doing a "You win!" dance, what type of block would you use?

   Answer: A command block would be best, because this is an operation that could be embedded in a longer sequence of operations (a script), and there is no result value to report.

4. If you wanted to define a block for a character that indicated whether or not it was an enemy in a game, what kind of block would you use?

   Answer: This is a reporter block, or more specifically a predicate block. Any time a block calculates a yes/no or true/false answer, you should define it as a predicate block.

5. Below is the definition of a "mystery" block – what does it do (a high-level description, not a step-by-step description)? [See questions above for the picture]

   *Answer: This block spins the current sprite around a certain number of times, where the number of times is passed as a parameter. Notice that this is a loop inside a loop – the inner loop turns the sprite 24 times, 15 degrees each time, for a total of 24*15=360 degrees. So the inner loop does a complete 360 degree rotation. Now this complete*

rotation is repeated pParam times, so if you wanted a sprite to spin around 3 times you would call it with "mystery 3".

6. The block and parameter names in the previous question were purposely vague, which is not what you want to do when you are really programming. How could you change the definition so that it makes more sense to both a user of the block (the name of the block is most important for this) and to programmers understanding how the block is implemented (names of parameters and variables are important for this)?

   Answer: For an abstraction to make sense, how it appears should describe the high-level description of what it does. The high-level description, which is what you should have figured out in question 5, is that it spins the sprite a certain number of times; therefore, a good name for the block would be "spin … times" (where … is the parameter). From a programmer's standpoint, the parameter name "pParam" is horrible since it gives no clue what the parameter actually represents. A better name would be "pRotations" since the parameter represents the number of rotations you are supposed to make. Here's a good final definition: