The Beauty and Joy of Computing[1]

Lab Exercise 4: Starting a simple math tutor program – and more interaction

Objectives

By completing this lab exercise, you should learn to

- Create your own reporter and predicate blocks;
- Distinguish between three different scopes of BYOB variables: global, sprite local, and script local;
- Choose appropriate variable scope when writing BYOB programs;
- Describe and implement operations needed to add fractions; and
- Write programs that interact with the user through typed inputs.

Background (Pre-Lab Reading)

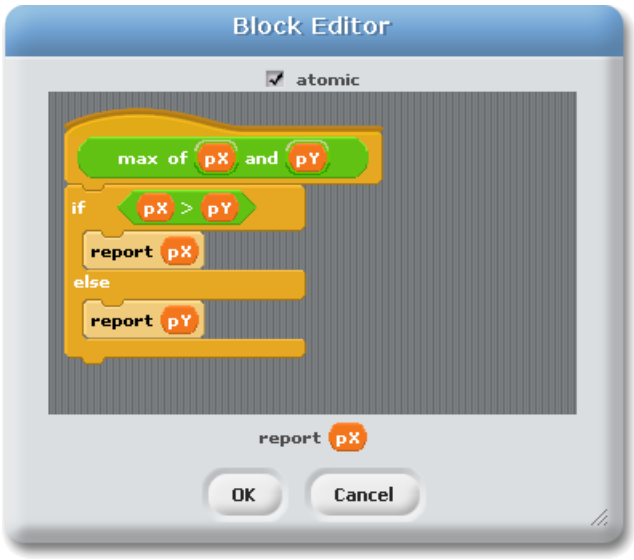
You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures that show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you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

The idea with this lab, and with the homework that follows up on this lab, it to create a math tutor program that teaches school kids about working with fractions: finding lowest common denominators and adding fractions. This lab is about doing the calculations, and your complete tutor program will combine these calculations with the "stamping" scripts from the previous lab so that a "teacher" can write problems on a "board" for students.

Defining Your Own Reporter and Predicate Blocks

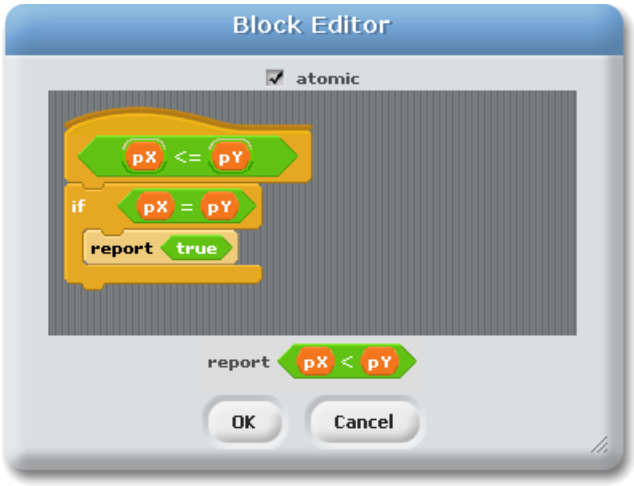
In the last lab you learned about different types of blocks (command, reporter, and hat), and created some of your own command blocks. The distinguishing characteristic of a reporter block is that it reports a value – since it produces a value, like a function in mathematics, many programming languages use the term "functions" rather than reporters. 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 we did with command blocks in the last lab, and add two parameters (using our naming convention!) along with the title text. Like the built-in operators of "+", "-", etc., we won't give the parameters default values. The definition

looks like this:



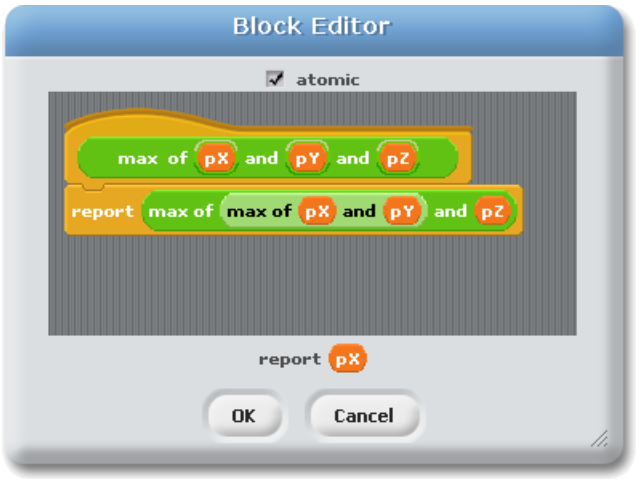
Notice the “report” blocks that are used in the script and the “report” statement at the bottom of the window. The internal ones directly report a value (if pX is larger it reports pX; otherwise, it reports pY). The “report pX” at the bottom is the default return value – if execution “falls off” the bottom of the script without a report block being found, this is the value that is reported. In this particular example definition 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”:



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.

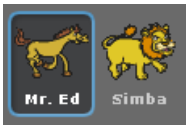
In the last lab you used one of your custom blocks (“stamp character”) to build up an even more complex block (“stamp string”). Using one of your blocks to build other blocks is common with reporter blocks as well. For example, you might define a block that reports the maximum of three numbers using 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:



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 more practice with this, making blocks that simplify your work in an upcoming homework assignment (creating a simple math tutor).

Scope of Variables

Another topic in this pre-lab reading covers some issues related to variables that we have just glossed over up until now. The idea of a variable is simple, and you have used them for the previous two labs, but there are some details and subtleties that you should understand. We start with a simple scenario so that we can introduce the concepts: We are making a simple game that has a horse and a lion (which often play nicely together, right?), so we start by importing these sprites – we’ll give these names by changing “Sprite1” and “Sprite2” at the top of their sprite pane, and we’ll name the horse “Mr. Ed” and the lion “Simba”. Once we create these, the sprites area shows both sprites with their names, like this:



Next we want to declare some variables: Our game will have a score, which we will keep track of in a variable, and the amount of food and gold that the two characters have. To make the score variable, we go to the “Variables” category and click the “Make a variable” button, which pops up the window that allows us to give the variable the name “score” – after doing that, this

is what the window looks like:

Items in BYOB, including scripts, variables, and block definitions, can exist either everywhere (in which case we say they are **global**) or just in the context in which they are defined (in which case we say they are **local** to some context). In the very first lab, we talked about scripts defined with a sprite being “local to the sprite”, which means that they only apply to the sprite in which they are defined – for example, in the “Angry Alonzos” script from Lab 2, Alonzo flew from one side of the screen to the other. But since that script was defined in Alonzo’s script area it was local to Alonzo. If we wanted the dragon to fly like that, we could not use the script defined for Alonzo. In that situation we could copy Alonzo’s script over to the dragon, but that just makes another local copy for the dragon – any changes we make to the dragon’s script would not change Alonzo’s script and vice versa. We refer to the context in which something is defined to be the item’s **scope**, which can either be global or local to some item like a sprite.

The choice in the variable definition between “For all sprites” and “For this sprite only” chooses the scope of the variable: Either the variable is global, so it exists for all sprites, or it is local to this sprite. Our game has only a single score, not a score for each character, and all characters might need to access or change the score – that means we need it to be global, and keep the “For all sprites” selection checked, and then click on “OK”.

Next we define a variable for how much food, or hay, the horse Mr. Ed has. This only applies to the horse, so in this case we first select Mr. Ed as our current sprite, and then click “Make a variable” to define “hay” and select “For this sprite only”. We do the same thing for Simba, to create a variable called “meat.” The effect of this is immediately apparent: when we select the Mr. Ed sprite and look in the Variables category of the blocks palette, we see this:

And when we select Simba, the top of the Variables category changes to look like this:

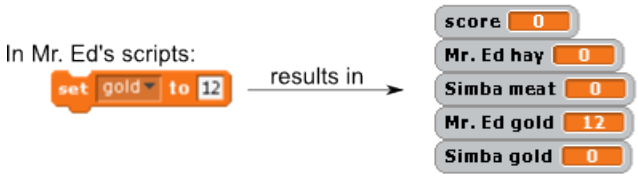
Notice that when Mr. Ed was selected we saw the global variable “score” and his food variable “hay” (separated by a line – global variables will always be above this line, and local variables

below), but we do not see Simba's variable "meat". When we select Simba, we can still see the global variable "score" but now the only local variable we see is Simba's variable "meat". Therefore, we have no access to Simba's "meat" variable when we are writing scripts for Mr. Ed, which makes sense.

Finally, let's define a variable for the amount of gold each character has (lions and horses love gold!). We define a local "gold" variable for Mr. Ed, and then do the same for Simba, where these are local to their sprite since each sprite can have a different amount of gold. Now when we flip between the two sprites, the variable "gold" looks like it stays there, but it's below the line in the variable list:

Below the line means it's a local variable, so despite the fact that you see the same name in both places there are actually two *different* "gold" variables: one for Mr. Ed and one for Simba. Since all of our variables are checked, they are all set as watch variables, and so are displayed on the stage:

Looking at that display, it's clear that there's only one "score" (no context is given), but there's both a "Mr. Ed gold" and a "Simba gold" – those are the two local "gold" variables, and the sprite name gives the context in which they are valid. While it's probably obvious if you think about it, you should be aware that when the "set …" block is used in a sprite's scripts, it can only access global variables or local variables for that particular sprite. For example:



When you create variables, you should follow what I'll call the **Principle of Least Scope**[2]:

> *Define variables with the smallest possible scope.*

This means that if a variable *can* be local, then it *should* be local. There are two reasons for this: First, if you start defining lots of global variables, they appear in the "Variables" pane of every sprite in your program, so you will see all those variables (most of which might be completely

[2] That's not a standard computer science phrase – I just made it up – but it's patterned after a common computer security concept known as the "Principle of Least Privilege".

irrelevant to your sprite) and have to find a variable that you want – that's confusing, and there's no good reason for it! The second reason is that global variables form connections between sprites: changing a global variable in one sprite may change the operation of a completely different sprite that uses the same global variable. If you can reduce the amount of connections between the behavior of your sprites, then when something goes wrong it is much easier to locate and fix the error.

When you were defining your own blocks in the last lab, you were also given a choice between "For all sprites" and "For this sprite only". This has the exact same effect as the scope of variables: a block may be general-purpose and apply to all sprites, in which case it should be global, or it could be something specific to just a single sprite, in which case it should be local. For example, both Mr. Ed and Simba might run, so there could be a global block defined for "run", but then since only Mr. Ed eats hay there could be an "eat hay" block that would be for Mr. Ed only. If Mr. Ed is selected and you look at the blocks created this way, you will see this:

(note that there is no nice global/local dividing line like there is for variables). If you select Simba, the "eat hay" block that is local to Mr. Ed disappears, but instead there might be a block that reflects Simba's eating preferences:

Script Variables

There is one more kind of variable, that we haven't used yet, but will be very useful for this lab: a **script variable**. A script variable has a very small scope, as it is only valid within the script in which it is defined. Consider this problem: In our game, gold is being produced faster and faster as the game progresses. In the first minute the game produces one bar of gold; in the second minute it produces two additional bars of gold; in the third minute it produces three *more* bars of gold, etc. So the total produced in the first 3 minutes is 1+2+3=6 bars of gold. We want to make a block that can tell us how many blocks of gold were produced in some time interval – thinking abstractly about *what* we want, rather than *how* we are going to do it, we decide we want a reporter block that looks like this:

This will report the value of the numbers from the low number to the high number, inclusive. So the block above reports the value of 1+2+3+4, or 10. This is a general operation, not tied to any specific sprite, so we'd like to define this as a global block (i.e., "For all sprites"). Identifying the operation we want to perform and how we want to use it, before actually worrying about the details of how the block will work or implementing it, is called **top-down design**.

There's actually a formula for computing this sum, but let's pretend that we don't know that and that we need to actually add up the numbers in the range. To add up 1 to 4 we'll need something that counts and takes on the values 1, 2, 3, and 4 (that's one variable), and we'll need to keep track of the sum as we add each of these numbers in (that's another variable). In

fact, we'll use the "counter pattern" from the previous lab, but consider the variables (such as the counter) more carefully. Remember the "Principle of Least Scope"? We can't use a local sprite variable for this, since the block is to be used for any sprite, and we don't want to use a global variable for all the reasons that make the Principle of Least Scope a good idea. The answer to this dilemma is a special kind of variable called a script variable – script variables have smaller scope than even sprite local variables, because they are only valid in the script in which they are defined and used (remember that a script is just a connected sequence of blocks, so a sprite may have multiple scripts). Script variables are most commonly used in a block definition, but they are not limited to that: regular scripts defined for a sprite can also have script variables. To create a script variable, you use the "script variables" block that is in the "Variables" category of the block palette, and looks like this:

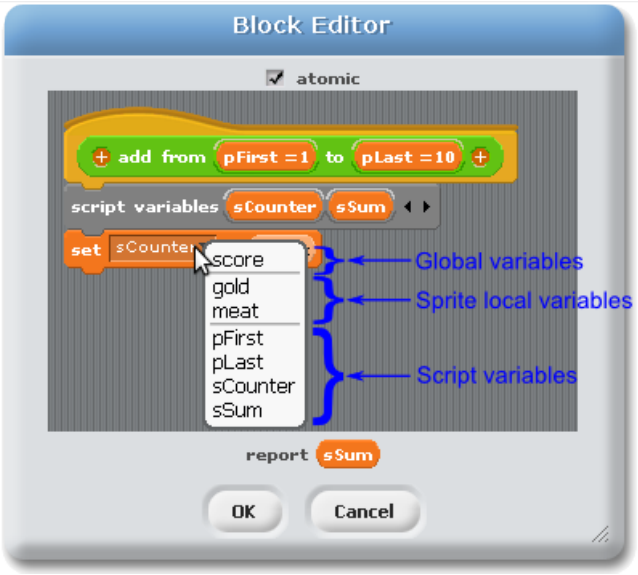
This block creates a script variable named "a". The block also has a new feature that we haven't seen before: it takes a variable number of arguments. If you click on the arrow at the far right hand side of the block, it will add an additional argument. Here's what it looks like after we do that to create a second script variable, which BYOB calls "b":

At this point you can create more script variables, or delete the ones that are there (using the left arrow). Script variables "come into existence" when the script executes this "script variables" block, and disappear when the script containing this block finishes.

The default names for script variables are "a," "b," etc. These are pretty horrible names, so you should always change the names of the script variables. Just like we did for parameters, we will use a naming convention for script variables: they should start with a lowercase 's' – since a variable always represents some quantity, the rest of the name should reflect what it represents. For example, in our definition of the "add from … to …" block, we need the two variables described above, so we will name them "sCounter" (for the counter that goes 1, 2, 3, 4) and "sSum" (which keeps track of the sum). To change the name of a variable, just click on it in the "script variables" block, and a pop-up window will allow you to set a new name.

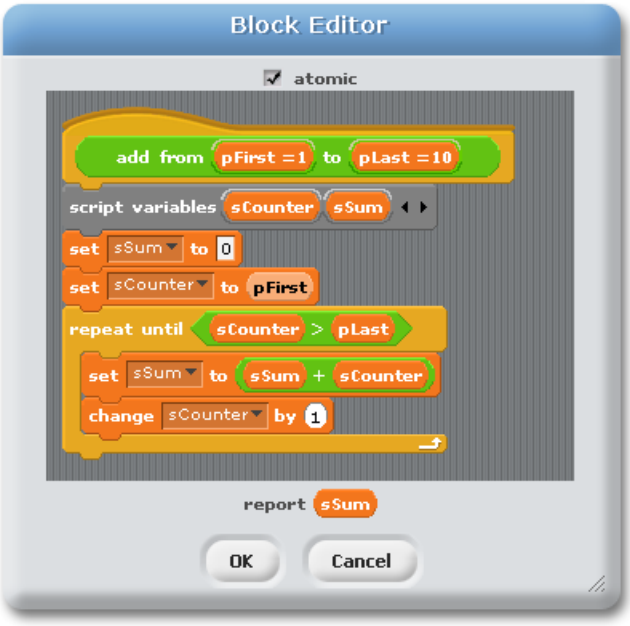
Let's get back to our problem of defining an "add from … to ..." block: We start defining the block as we've done many times, giving it two parameters (pFirst and pLast) with default values of 1 and 10, then we put in the "script variables" block and set the names of our script variables. The next thing we want to do is initialize our script variables, so we use a "set" block – dragging it out and clicking on the drop-down menu to select the variable name, we see variables grouped by scope, separated by horizontal lines. It looks like this, where the scopes of variables are labeled

in our picture (they aren't labeled in BYOB):



Notice that the parameters are actually script variables! That makes a lot of sense, because parameters are only meaningful in the script that defines the block to which they are parameters. An important point about script variables is that they "come into existence" when the script executes the "script variables" block, and don't exist outside of the script containing the "script variables" block. If you were to put a "set …" block above the "script variables" block, it would not give you those variables as options, since they don't exist at that point in the script. The script variables will also not be available in scripts that are not connected to this one – keep that in mind if you like to assemble multi-block components that you snap together later into a script, since anything separated from the "script variables" block will not have those variables available, even if you intend to combine it with the "script variables" block later!

Now let's finish our block definition. The idea is simple extension of the counter pattern, but you should study the solution below to get comfortable with how this works. First, we limit the iteration not by a fixed number of iterations, but by a final value for the counter ("pLast"). Second, we use the variable "sSum" to keep the running total as we count through the range from "pFirst" to "pLast." Finally, our block should report the sum. Putting these ideas together, we get the following solution (on the next page):



Adding a variable to a loop that keeps a running value is another very common computing pattern, and so this has a name: the **accumulator pattern** (you are accumulating the final answer – in this case the sum). Make sure you fully understand how this script works!

There's only one last thing to say about script variables: It is possible for a script variable to have the same name as either a global or a sprite variable. In other words, it's possible that we could have a global variable in our program named "sum" in addition to a script variable named "sum". Despite the fact that these have the same name, and they look exactly alike when used in a script, they are two *different* variables. You can tell the difference in the "set …" block because of the positioning within the drop-down menu, but in other parts of the code it's impossible to tell which variable you are using – *unless*, you reliably follow our naming conventions! With the naming conventions, you know that a variable starting with an 's' is a script variable, and can't be global or sprite local. Hopefully you're starting to see the value of the naming convention. *Tip: You can also get BYOB to help you see the scope by turning on the "Scope Contrast" option in BYOB's "Edit" menu.*

Why doesn't BYOB force script variables to have different names from other variables? There's actually a very good reason, and it has to do with abstraction: When we define a block, we should concentrate on the logic of that block, and not be concerned at all with what's going on outside that block definition – the block definition should be entirely self-contained. If we say we have to avoid using the same name as a global or sprite variable, then we are not self-contained since we're thinking about things outside the block definition. You can also look at this from the other perspective, that of someone using the block. A programmer using the "add from … to …" block should not have to worry about the fact that certain variable names are used inside the block definition. Good block definitions are self-contained abstractions, where the user of the block only needs to be concerned about *what* the block does, and should not have to think at all about *how* the block does its work. By using our naming convention, we get the same useful

benefit of BYOB forcing the names to be different, without BYOB trying to enforce its own view of how you should think about things.

Now you know almost everything there is to know about variables in BYOB!

Math Background for Lab 4

The goal of this lab and the following homework assignment is to create a simple math tutor program, that helps school kids learn about and practice adding fractions. While you are free to have your tutor program explain adding fractions in any way that is sensible, the underlying code of this assignment will need to add fractions as well, and should take the specific approach that we describe here. For example, let's say that we want to compute the sum 5/6 + 6/15. We need to convert these fractions so that they have a common denominator, so this sum is the same as 25/30 + 12/30, at which point we can add numerators and get the sum of 37/30. How do we describe this algorithmically, so we can do this computation in our program?

We can compute everything we need if we can compute the "greatest common divisor", or GCD, of the denominators. The greatest common divisor is the largest integer that divides evenly into the two values, so in our case we find that the GCD of 6 and 15 is 3, which we typically write as GCD(6,15)=3. You'll have to define a block for computing the GCD in Activity 1 below, but you should think a little about how you could do it now: think of starting at the largest possible value of the GCD and decreasing it until it divides evenly into both numbers. So we could start at 6 and ask "does 6 divide evenly into 6 and 15?" 6 doesn't divide evenly into 15, so we decrease it and try 5. 5 doesn't divide evenly into 6, so we step down to 4. That doesn't divide evenly into either number, so we go down to 3. That one works, and since we were counting down and we stopped at the first common divisor, we know we have the *greatest* possible divisor. This is tedious if you do it by hand, but tedious calculations are exactly the thing that computers are good at! Note that there are much more efficient ways to compute the GCD, but this simple technique is all we need for this lab exercise.

Now that you can compute GCD's, how does that help you find common denominators? That turns out to be easy! Let's say we're adding a/x + b/y, where a, b, x, and y are all integers. The two denominators are x and y, and we use g to denote the GCD of these values (so GCD(x,y)=g). Now we know that g divides evenly into both x and y, so $x * \frac{y}{g}$ is an integer, which happens to be the lowest common denominator! Why? We can compute y/g (remember, g divides y, so this is an integer), and multiply the fraction a/x by $\frac{y/g}{y/g}$. This gives $\frac{a*(y/g)}{x*(y/g)}$ for the first fraction. For the second fraction, we multiply by $\frac{x/g}{x/g}$ (note that x/g is an integer), so the second fraction becomes $\frac{b*(x/g)}{y*(x/g)}$, and both fractions have the common denominator $\frac{x\,y}{g}$. Solving one problem (finding lowest common denominators) by turning it

into a related problem (GCD) is called a **reduction** – we will say more about this in the Discussion section.

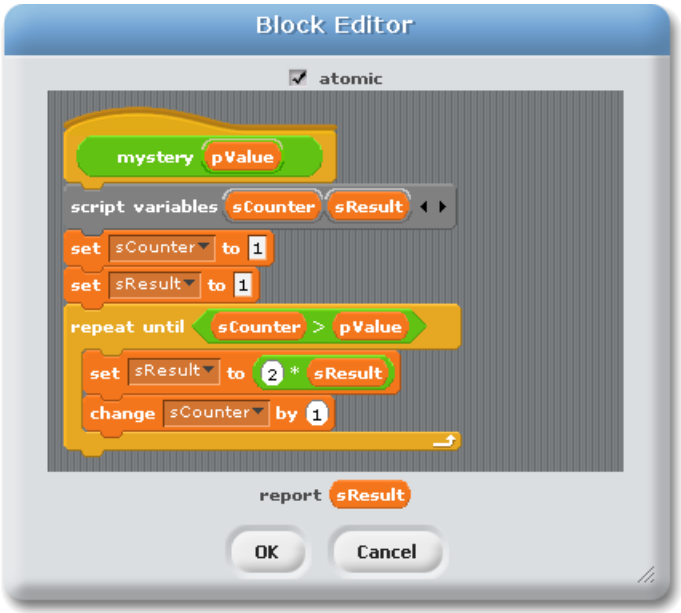
Putting our example in this context: we have 5/6 + 6/15, so a=5, b=6, x=6, and y=15. We figured out above that GCD(6,15)=3, so g=3. To adjust the first fraction, we compute y/g = 15/3 = 5, and multiply the first fraction by 5/5 to get 5*5 / 6*5 = 25/30. For the second fraction, we compute x/g = 6/3 = 2, and multiply the second fraction by 2/2 to get 6*2 / 15*2 = 12/30. Now that both are over the common denominator of 30, we can add numerators to get (25+12)/30 = 37/30.

You may not have done additions of fractions by computing GCDs in the past, but the nice thing about this approach is that once you know the GCD of the two denominators, computing everything else is easy! Hopefully all of this was clear to you – work through the practice problems below in “Self-Assessment Questions” to practice and make sure you see how things work out.

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. What scope variable would you use to keep track of the time remaining in a game?
2. What scope variable would you use to keep track of the amount of silver that a character has?
3. If characters can have gold, copper, and silver, and you created a reporter block to report their “net worth” – adding up the value of all their different kinds of treasure – what scope variable would you use for adding up this value?
4. Consider the following BYOB block definition – what does this report when called with argument 5 (as shown on the right)?



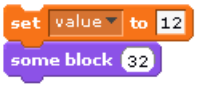
What does this report?

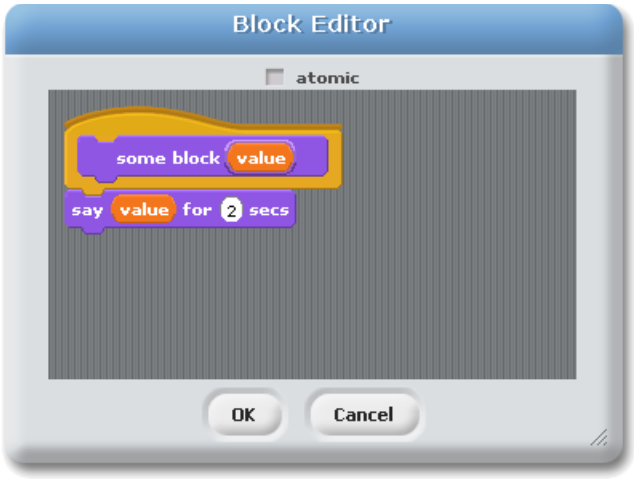
Can you describe at a high level what this block computes (remember to focus on what it computes – how it does so is irrelevant)? Let’s say that the programmer wanted to see

what the value of sCounter was after the loop ran, and so somehow managed to create this code:

What's wrong with this?

5. In the following picture, the script on the left is defined in Alonzo's scripts area, and the part on the right shows how the block "some block" is defined:





What does Alonzo say when this is run? Before you spend too much time trying to figure that out, think of non-traditional answers – this is actually something of a trick question. If you figure that hint out, can you describe what the problem is here? What *should* have been done to avoid this problem?

6. Walk through the steps of adding the fractions 2/3 + 3/4 using the technique described above. List out each step: What is g? Show how you use g to calculate what each fraction is multiplied by, and then show the final sum.
7. Walk through the steps of adding the fractions 5/6 + 3/10 using the technique described above. List out each step: What is g? Show how you use g to calculate what each fraction is multiplied by, and then show the final sum.

Activities (In-Lab Work)

Activity 1: For this activity, you should make a reporter block that computes the greatest common divisor of two integers. Your reporter block should look like the following (shown displaying the GCD of 6 and 15):

Hint: The "Math Background for Lab 4" section in the Pre-Lab Reading described a simple algorithm, or technique, for computing the GCD of two integers. This involves testing if one number divides evenly into another, which you know how to do if you think about it: asking whether x divides evenly into y is the same thing as asking whether "y mod x" is 0 (i.e., the remainder of the division is 0). The GCD solution involves counting down in a loop, very, very similar to the way we counted up in the "sum of …" block in the first part of the Pre-Lab reading.

Figuring out when this loop should stop is a little more difficult, but consider how easy it would be if you had a block of the following form:

If you define this block first so that you can use it in your GCD definition, it will simplify this activity greatly! There are several ways you could define this block. You can actually do it with just a couple of "if" blocks, or alternatively you could consider using the "and" block, which is 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" in the name of the block, so consider a test that looks like this (for appropriate definitions of x, y, and z):

You'll need to fill in the parts that correspond to dividing evenly (remember the "mod" block!) if you want to use this alternative, but if you figure that out then the definition is pretty simple!

Save your complete solution as Lab4-Activity1.

Activity 2: Use the technique described in the "Math Background" section to create a reporter block that finds that lowest common denominator for two integers. It should look like the following:

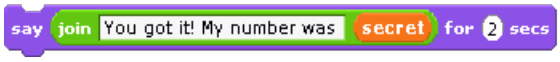
Save your solution to this activity as Lab4-Activity2.

Activity 3: In this activity, you will create a reporter block that takes 4 parameters, defining two fractions, and computes the numerator of the sum of these (so, taken together with the block from the previous activity, you should be able to compute the sum, both numerator and denominator), of any two fractions. You do not need to worry about putting fractions in lowest terms. Here's an example of what this block should look like:

Save your solution to this activity as Lab4-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 the last activity in this lab, you will make a number guessing game so that you can practice using two new blocks: the “ask” block and the “join” block. Here are the basics of what you will do: create a game in which a character (you can use Alonzo or whatever character you want) picks a random number between 1 and 10, and asks the player to guess the number. The “ask...” block is in the “Sensing” category of the blocks palette – when executed, it draws a dialog talking bubble for the character (like the “say” block) and provides an input box for the user to type an answer. You can test what the user typed by using the “answer” variable that is right below the “ask” block in the blocks palette, and your character should say either “Too low”, “Too high”, or “You got it! My number was 4” (where 4 is replaced with the actual number). For that last message, you need to combine a fixed text string (“You got it! My number was “) with the value of a variable (the secret number), and display all of that together using a “say” block. Remember the “join” block that you used in the last lab? This is a great place to use it again, doing something like this:



Finally, you should use a repeat loop to allow the user to keep guessing until they get the answer correct.

The only requirements for this activity are a number guessing game that repeatedly says “too low” or “too high” until the number is guessed, with at least one use of the “join” block in an argument to “say”. You are encouraged to be creative in the time you have for this activity: use different characters, change the dialog messages (as long as they fit the minimum requirements of the game), have a celebratory animation when the number is guessed correctly, or anything else you can think of. When you complete this activity, save your final program as Lab4-Game.

Submission

In this lab, you should have saved the following files: Lab4-Activity1, Lab4-Activity2, Lab4-Activity3, and Lab4-Game. In this lab, activities 2 and 3 do not destroy work in the previous activities, so if Lab4-Activity3 contains all of your work from the previous activities, you do not need to turn in Lab4-Activity1 or Lab4-Activity2. Turn these in using whatever submission mechanism your school has set up for you.

Discussion (Post-Lab Follow-up)

Reduction: In doing the math for adding fractions, the problem of finding the lowest common denominator was solved by computing a greatest common divisor and then transforming the result using some simple formulas. In other words, all of the real computational effort was done in the GCD function, and then we just did a few minor calculations to turn the answer into what we originally wanted (the lowest common denominator). This is called a **computational reduction** (or just “reduction” when it’s clear that you’re talking about a computational reduction), and we say that we have “**reduced** the problem of computing lowest common denominators to the problem of computing greatest common divisors.” Unfortunately, the word “reduce” has different meanings in different contexts: At the end of the “Math Background” section we noted that you can reduce a fraction to lowest terms using the GCD function, which is an entirely different use of the word “reduce” that has absolutely nothing to do with

computational reductions. Yes, this is confusing, and I'm about to make it even more confusing. Since we turned the problem of reducing a fraction into the problem of computing a GCD (plus a couple of additional divisions), we have reduced the problem of reducing a fraction to the problem of computing a GCD. Did you get that? It's unfortunate terminology in this case, but it does make sense if you think it through, and it's worth your time to figure out what that sentence means! As a final comment, notice that we reduced two different problems on fractions to a single core problem, that of computing a GCD. When you can do this in a program it is fantastic news! You put the complicated computation into one function, which you can write and test thoroughly, and then you can use that one well-tested solution to solve your other problems.

Top-down design: When we introduced the "add from … to …" block above, that block was driven by a higher-level need in the system. Starting from a system-level perspective and figuring out what building blocks or components you need for the system, and then figuring out the building blocks you need to build those components, and so on, is what we refer to as "top-down design". This is pretty much identical to the notion of "problem decomposition" that we described in Lab 1, but people tend to use the phrase "problem decomposition" for small-scale problems, and "top-down design" when designing complex systems. The opposite of top-down design is **bottom-up design**, which means starting at the lowest building blocks, and assembling things until you have something interesting – you start with the simple pieces and build more and more complex objects. While "top-down design" and "bottom-up design" are the common terminology in computer science, in other areas of design top-down design is referred to as "decomposition" and bottom-up design is referred to as "synthesis."

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

- *accumulator pattern*: A programming pattern that loops through values and updates a “running value,” like a running sum, running maximum, etc.
- *Boolean operator*: An operator that works on true/false values, such as “and”, “or”, and “not”.
- *bottom-up design*: Constructing components from basic operations, and building more complex components from those, and so on until a complete system is built from the bottom up.
- *computatoinal reduction (or reduction)*: Using the solution to one computational problem (“Problem A”) to solve another one (“Problem B”), typically with a small amount of computation to convert the inputs for Problem B into inputs for Problem A, and/or a small amount of computation to convert the output from Problem A into the output for Problem B.
- *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).
- *global*: A computational item – like a variable or block – that is available from all parts of a program.
- *local*: A computational item – like a variable or a block – that is available only in a particular context, where the context can be a particular sprite or a particular script.
- *principle of least scope*: The program design principle that says that you should always use the smallest possible scope for a variable – so, in particular, global variables should only be used as a last resort.

- *reduce*: In the description of "computational reduction" above, we say that we reduce Problem B (the problem we want to solve) to Problem A (the problem we have a solution to).
- *scope*: The parts of a program in which an item (like a variable or block) is defined and available.
- *script variable*: A variable with very small scope: it is only available from the point that it is defined (using a "script variable" block) until the end of that script.
- *top-down design*: Designing a program by starting at the top, system level, deciding what components are needed to make the system work, and further breaking those components down into smaller pieces.
- *truth table*: A table that shows the value of a Boolean expression for all possible values of inputs.

Answers to Pre-Lab Self-Assessment Questions

1. What scope variable would you use to keep track of the time remaining in a game?

   Answer: Global. A value associated with the game as a whole, and not with a specific sprite, needs to be global.

2. What scope variable would you use to keep track of the amount of silver that a character has?

   Answer: Sprite local. This can't be local to a script or a block definition, since it's a long-term value (existing beyond the time that a specific script is running). It can be sprite local, and so it should be (as opposed to global) according the principle of least scope.

3. If characters can have gold, copper, and silver, and you created a reporter block to report their "net worth" – adding up the value of all their different kinds of treasure – what scope variable would you use for adding up this value?

   Answer: Script local. The variable used to add up the value is only needed temporarily, while the script is running. Therefore, it should be a script variable (scope script local).

4. Consider the following BYOB block definition [*see questions for the picture*] – what does this report when called with argument 5 (as shown on the right)?

   *Answer: It reports 32. You can figure this out by tracing through the code. When the block is called, pValue becomes 5. Before we reach the loop, variable sCounter is set to 1 and sResult is set to 1. Next, we hit the repeat loop – since sCounter (with value 1) is not greater than pValue (with value 1), we execute the blocks inside the loop which change sResult to 2*1=2, and changes sCounter to 2 (changed by 1 from previous value). Then we check the condition and see that sCounter (now 2) is still not greater than pValue (still 5), so we need to execute the blocks inside the loop again. We continue this way until the loop exits, and we summarize this tracing as follows:*

   *After next loop iteration: sResult = 2*2 = 4 ; sCounter = 2+1 = 3*
   *sCounter is not > 5, so loop again giving: sResult = 2*4 = 8 ; sCounter = 3+1 = 4*
   *sCounter is not > 5, so loop again giving: sResult = 2*8 = 16 ; sCounter = 4+1 = 5*
   *sCounter is not > 5, so loop again giving: sResult = 2*16 = 32 ; sCounter = 5+1 = 6*

And now, sCounter is > sResult, so we stop the loop iteration, reporting sResult (current value is 32).

Can you describe at a high level what this block computes (remember to focus on what it computes – how it does so is irrelevant)?

Answer: It computes 2^{pValue} (two to the power of the argument).

Let's say that the programmer wanted to see what the value of sCounter was after the loop ran, and so somehow managed to create this code:

What's wrong with this?

Answer: The variable sCounter doesn't exist outside the script in the definition of "mystery" – that means that this variable doesn't exist in this context! If it were possible to make this code, it must mean that there is a different variable named sCounter (not the script variable we're interested in) that this is accessing.

5. In the following picture, the script on the left is defined in Alonzo's scripts area, and the part on the right shows how the block "some block" is defined [*see questions for picture*]. What does Alonzo say when this is run? Before you spend too much time trying to figure that out, think of non-traditional answers – this is actually something of a trick question. If you figure that hint out, can you describe what the problem is here? What *should* have been done to avoid this problem?

   Answer: It's actually impossible to tell what it would say from these pictures. Is the "value" argument to the say block the global variable "value" or the parameter "value"? These are two different variables, even though they look the same. If this is the global variable, Alonzo would say 12. If this is the parameter, then Alonzo would say 32. If we had followed our naming convention we would never have gotten into this situation, since the parameter would be named something like "pValue" and it would be clear whether the "say" block was using the global variable or the parameter.

6. Walk through the steps of adding the fractions 2/3 + 3/4 using the technique described above. List out each step: What is g? Show how you use g to calculate what each fraction is multiplied by, and then show the final sum.

   *Answer: We first compute g, which is the GCD of denominators 3 and 4 – for these values, g=1. Next, we multiply both numerator and denominator of 2/3 by 4/g=4, so we get (2*4)/(3*4) = 8/12. We adjust the second fraction similarly, multiplying numerator and denominator of 3/4 by 3/g=3, so we get (3*3)/(4*3) = 9/12. Finally we can directly add 8/12 + 9/12 to get 17/12, which is the final answer.*

7. Walk through the steps of adding the fractions 5/6 + 3/10 using the technique described above. List out each step: What is g? Show how you use g to calculate what each fraction is multiplied by, and then show the final sum.

*Answer: We first compute g, which is the GCD of denominators 6 and 10 – for these values, g=2. Next, we multiply both numerator and denominator of 5/6 by 10/g=5, so we get (5*5)/(6*5) = 25/30. We adjust the second fraction similarly, multiplying numerator and denominator of 3/10 by 6/g=3, so we get (3*3)/(10*3) = 9/30. Finally we can directly add 25/30 + 9/30 to get 34/30, which is the final answer. Note that we could reduce this to lowest terms by computing the GCD of 34 and 30 (which is 2), and dividing each number by 2 resulting in 17/15. See how useful being able to compute GCDs is?*