The Beauty and Joy of Computing[1]

Lab Exercise 2: Making things interesting with interaction, variables, and more

Objectives

By completing this lab exercise, you should learn to

- control sprite movement through a variety of mechanisms;
- write scripts that respond to user input and user actions;
- define and modify variables to track values in a program;
- use the sprite pen and stamp capability to draw pictures; and
- work with division and mod operators.

Background (Pre-Lab Reading)

In lab exercise 1, you learned how to make simple animations using BYOB, including how to get multiple sprites to coordinate and synchronize their actions. The end-product was an animation which could be viewed like a movie. What makes computers a powerful medium though, is the ability to interact with the user - for actions to change based on actions of the user. This two-way interaction is one of the important things that we look at in this lab exercise. As programs respond to users and change behavior based on user interaction, a program typically needs to keep track of certain values that control its actions, remembering user choices and responses to user actions. This is the fundamental concept of a program variable, which we introduce and explore in this lab exercise.

Since this is your second lab, the amount of things that you have to figure out for yourself is a little higher. The activity descriptions still lead you through things step-by-step, but there is more text and fewer pictures - you need to find the right pieces based on their descriptions, rather than just mimicking diagrams. Since you have to figure out more on your own, it's even more important that you read this "Pre-Lab Reading" section carefully so that you can come into the lab prepared and ready to take on the activities in the lab. This is the first step in "taking off the training wheels" - in future labs, there will be less of the step-by-step instructions, and you'll need to figure out even more on your own!

Directions in BYOB

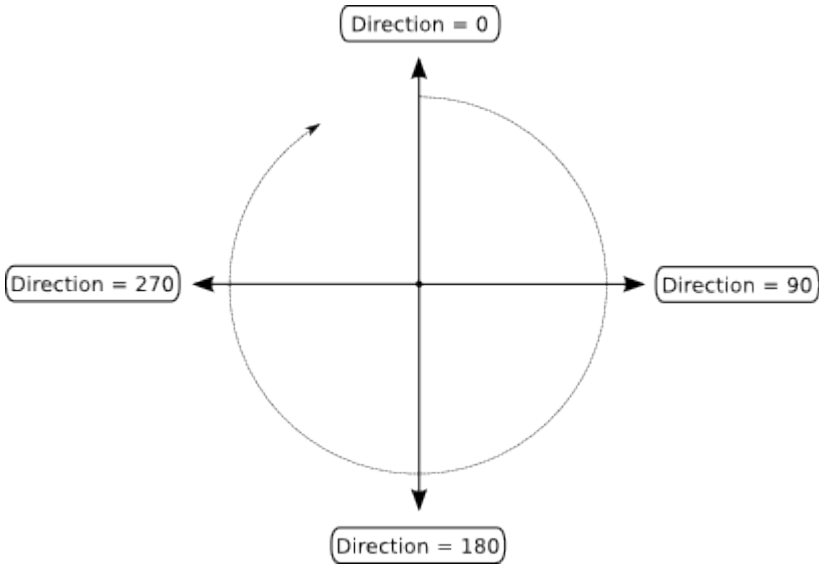
Recall that in the last lab, we talked about the "Move … steps" block - the block that makes a sprite move a certain number of steps in the direction that it is facing. We didn't really talk about what "the direction that it is facing" means, so now it's time to learn about directions. In particular, the last lab included an example in which we said that the coordinates/direction at the top of the sprites pane might say something like this:

x: 120 y: 2 direction: 90

---

[1] *Lab Exercises for "The Beauty and Joy of Computing"*

So what does the “direction” mean? It’s an angle, that is based on the following coordinate system:

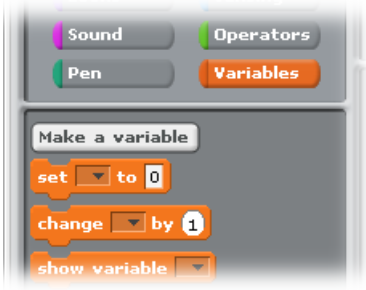


A direction set to 0 will result in the “move” block moving the sprite up, 90 will result in moving right, 180 will move down, etc. Therefore, if you experimented with the “move” block in the previous lab, when the direction was set to 90 the sprite should have moved to the right. Increasing the direction value will adjust the angle in a clockwise direction, like the dashed line in the figure above. Only the main directions are shown in the diagram, but any value in between these can be chosen as well: a direction of 45 will go up and to the right, and 225 will go down and to the left. This also “wraps around” at 360 (because 360 degrees is a full rotation), so direction 450 is to the right just like direction 90 (since 90+360=450), and direction -90 goes to the left just like direction 270 (since -90+360=270). [*Note: If you have seen polar coordinates in a math class, these directions might unfortunately cause some confusion. In standard polar coordinates, direction 0 is to the right, and increasing this angle moves counter-clockwise. It’s unfortunate that BYOB uses non-standard directions, which were inherited from the Scratch system that BYOB is based on; however, as unfortunate as it is, it’s just something you have to learn to work with.*]

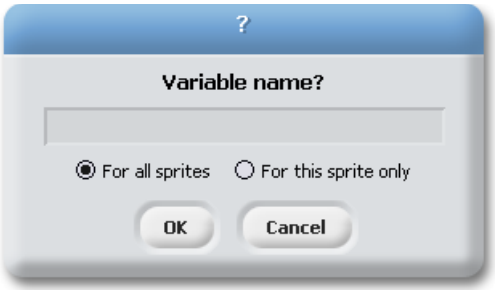
Storing and Monitoring Values using Variables

Each sprite always remembers its position (x and y coordinates) and its direction. Those are obvious values to remember, and are properties of all sprites, but what about other values you might want to keep track of? For example, a character in a game might have a health value or a strength. You might also need to keep track of a value that makes sense for your program as a whole but isn’t associated with any particular sprite, such as a current score or a timer. What we need is something we can use in our program that can represent these values, which can change as your program executes.

In algebra we use letters to stand in for values that can change - for example, 2x+10, where x is a variable that can take on different values. We do the same thing in programming with **variables** - in fact, there's an entire "Variables" category in the blocks palette, and when it is selected you'll see the following at the top of the blocks palette:



Each value you want to keep track of is a different variable, and you create variables by clicking the "Make a variable" button shown above. This pops up the following dialog box:



The text field, where you can type, is for the name of the variable - while you might be used to names like "x" in algebra, you should give variables in programs meaningful names. For example, a variable representing the character's health could be named "health," so when you see it in your program you don't have to remember what a meanless name like "x" represents. In the last lab we talked about scripts being local to a sprite, meaning that they were only defined in the context of that one sprite (so Alonzo's scripts were different from the dragon's scripts). Unlike scripts, variables can be either local ("For this sprite only" - for things like a character's health or strength) or **global** ("For all sprites" - for things like an overall game score or timer that is not associated with a single sprite). You can select whether the variable is global or local by selecting the appropriate option underneath the sprite name. Once you enter the name and select the global or local option, clicking "OK" will create the variable for use in your program. We will explore differences between local, global, and other variables in a later lab.

Once you create variables they appear as blocks in the "Variables" category. For example, after creating a global variable for "score" and a local variable for a sprite's "health", the following appears in the Variables category:

Global variables will always appear above the line, and local variables below the line. Since local variables are associated with a specific sprite, they will only appear when that sprite is elected in the sprites pane.

When we make a variable represent a specific value (meaning a specific number), we say that we **set** the value of the variable or **store** a value in the variable. To set the value of a variable in your program, you use the block that looks like this:

The first argument is the name of the variable you want to set - if you click on it you will get a drop-down menu showing all the different variables that you can set - and the second argument is the value to store in the variable. In the picture above, the variable “health” is set to 42. After making a new variable, what is its value? The answer to this question is not obvious (there is an answer, it’s just not obvious!), and so it is good practice to use a “set … to …” block to set a new variable to a value that you select - we call this **initializing** the variable.

The one remaining thing to know about variables has to do with the check box next to the variable blocks, as shown above with checks in them for the “score” and “health” variables. When the box next to a variable is checked, the value of the variable is displayed on the stage at all times so that you can watch it change. For obvious reasons, we say that a variable that is checked has been set as a **watch variable** - a variable that the programmer wants to observe while the program is running. Normally, watch variables are only enabled during **debugging**, when the programmer is testing the program and wants to gain insight into what is happening or possibly going wrong in the program. If you explore the BYOB interface, you’ll see that even variables that are defined for you, such as coordinates and direction, can be set as watch variables.

Computing things in BYOB

Computing is what computers are best at: adding, subtracting, multiplying, dividing, etc. Now that we know how to store values in variables, it is time to look at how to perform computations using those variables. For example, you might want to add 5 to the score. These kinds of calculations are all in the “Operators” category of BYOB, and the first four blocks are in fact the four basic arithmetic operators:

To add 5 to a score, you would drag this block out, go to the “Variables” category and drag the “score” variable into the first position, and type a 5 in the second position, giving this block:

Notice that the arithmetic blocks are a different shape than others that we’ve seen before. They are rounded, so don’t snap into a sequence of operations in a script. This is because they only calculate values that can be used inside other blocks, and they don’t do anything else or change anything on their own. Blocks like these, which calculate values but don’t do any other action, are called **reporter** blocks because they report a value. If you want to see what a reporter block does, you can put in parameters, click on the block, and it will pop a small “speaking bubble” to tell you what value the block is reporting. For example, there’s a “mod” operator that you might

not have seen before, so you can drag it out, put in some values, and click on it to see something like this:

Do you know what that did? Can you guess? The “mod” block is actually very important for this lab, so we’ll describe what it does below.

Adding 5 to the score with a reporter block does not change the value stored in the variable score - it only calculates and reports what the score plus 5 is. If you want to actually change the score, you would put this inside a “set score to …” block. Adding to a variable and storing that as the new variable value is a very common operation, and so BYOB provides a shorter way to do this with the “Change … by …” block, so if we actually want to increase the value stored in the “score” variable by 5, we could use either of the two blocks - they do exactly the same thing:

The “change” shortcut only works when you want to add to a variable. If you wanted to double a variable, you’d need to use the longer “set …” form with the multiplication operator.

You can put formulas inside other formulas, leading to some fairly complex calculations. The order in which the individual calculations are performed depends on what is inside what, which is called the **nesting order** of the formula (the inside formula is said to be **nested** inside the outer one).[2] Consider the following two examples, where the one on the left has the formula “2*3” nested inside the formula “1+...”, whereas the one on the right has “1+2” nested inside “... * 3”:

The formula on the left computes 2*3 first (giving 6) and then adds 1 to that, for a final result of 7. The one on the right computes 1+2 first (giving 3), and then multiplies that by 3, for a final result of 9. These formulas look very similar, with the only difference being the order that formulas are inserted into other formulas, but this small difference changes the value that is computed!

Breaking up numbers into digits

When you store a two-digit number, like 47, in a variable, it is a single value. Even though you write down two digits, the computer doesn’t see it that way - it’s just a number. When it comes time for the computer to display this number, however, it needs to write down two digits just like you would, so the question is: How does the computer figure out that this number that is stored in a variable should be written as the digit 4 followed by the digit 7? In other words, how do we extract individual digits from a number?

An important mathematical operator that we will use to answer this question is the “**modulo**” or “**mod**” operator, which gives the remainder after dividing the first argument by the second argument. For example, the value of the formula “32 mod 5” is 2, since dividing 32 by 5 gives a

[2] If you have programmed in other languages, where formulas are typed, you needed to know what the “precedence rules” were for the language to know the order in which operations were performed. This isn’t an issue in BYOB, since the nesting order always indicates the order that operators are evaluated - no precedence rules needed!

quotient of 6 and a remainder of 2 (in other words, 32 = 6*5 + 2). As we saw above, there is a "... mod ..." block in BYOB, so creating this formula in BYOB would look like this:

Now consider taking any number mod 10 - what do you get? For example, what is 47 mod 10? How about 52 mod 10? 38 mod 10? If you did those operations correctly, you'd see that you always get the rightmost digit of the number (the answers to those mod examples are, in order, 7, 2, and 8). So now you know how to calculate the rightmost digit of a number: just take the number mod 10!

What about the leftmost digit or a two-digit number? If the number were a multiple of 10, you could just divide by 10. For example, if your number is 40, dividing this by 10 gives you 4, which is the leftmost digit. But what about the number 42? Dividing this by 10 gives you 4.2, so you end up with a fractional result[3], throwing everything off. If we could just get rid of the fractional part, we'd have the correct digit. Do you know what causes the fractional part? The remainder of the division by 10, and we know how to calculate that! Consider this: Take any two digit number, subtract the remainder when that number is divided by 10, and what do you get? You will always get a number that is evenly divisible by 10. Try it with a few examples:

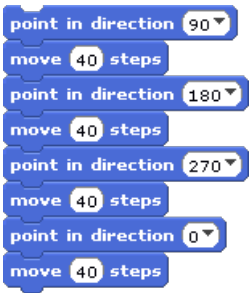
42 mod 10 = 2	and	42 - 2 = 40
56 mod 10 = 6	and	56 - 6 = 50
13 mod 10 = 3	and	13 - 3 = 10

In every one of those examples, after subtracting off the remainder you can divide by 10 to get the leftmost digit. Can you put this into a BYOB formula? The answer is at the end of this handout, but don't look right away - think this through and see if you can figure out the answer yourself, because it's a great test or your understanding!

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. If a sprite's direction is set to 135, and it starts in the center of the stage and keeps moving in this direction, where would it go off the stage?
2. What shape does the following BYOB script trace out?



3. Consider a game in which every character has a certain amount of gold. Would you use a local variable or a global variable to keep track of these amounts?
4. Consider a game that keeps track of how many enemies are left in the game. Would you use a local variable or a global variable to keep track of this?

[3] At least you do in BYOB - it's not the same in all programming languages!

5. What is the value of the following BYOB formula?

6. What is the value of the variable “mystery” after the following script runs?

Activities (In-Lab Work)

The following activities are to be done during the lab meeting time.

Activity 1: Experimenting with moving sprites

In this activity, you will experiment with sprite movement. Begin by creating a new project, so that Alonzo is the only sprite and is located in the center of the screen. Make sure the “Motion” category is selected in the blocks palette, and read through the different blocks to familiarize yourself with the possibilities. At the top of the sprites pane, you should see that Alonzo starts with a direction of 90 - using the directions picture from the pre-lab reading, what direction is Alonzo facing? Click the “move 10 steps” block several times - does Alonzo move in the direction that you were expecting?

Now let’s make Alonzo move up: Drag out the “point in direction...” block, and then the “move … steps” block, which you should snap below the “point in direction...” block. Now click on the argument in the “point in direction...” block, and change the direction to 0 (i.e., up). Your blocks should look like this:

Now if you click on the two-block set you just created, you should see Alonzo move up (click more than once if you didn’t see him move the first time). Did anything else change on the stage other than the direction Alonzo moves?

You can control some other aspects of how sprites are drawn when they move, using buttons at the top of the sprite info pane. Specifically, look at the buttons to the left of Alonzo’s picture, which we call the **sprite orientation control buttons**:

Like control buttons in almost any computer application, you can hover the mouse over one of these buttons and a hint will pop up giving information about what the button does - try this with all three of these buttons. Once you see what all three descriptions are, try clicking them (these are called "**radio buttons**", so only one can be enabled at a time). Try selecting different options, and clicking on your two-block move-up sequence for each one. You might not be able to tell the difference between the bottom two options now, but the difference will become clear in a few minutes.

If you want to take a little time to play around with the other movement blocks, such as "glide..." or "turn..." this would be a good time to do that. Just don't get so distracted that you don't have time to finish this lab! There is nothing to save or turn in for this activity.

Activity 2: Responding to the user

Making scripts respond to user actions is the key ingredient to making an animation more engaging than just watching a movie. Our first step along these lines is to make it so that sprite movements are controlled by the arrow keys. Starting from the two-block action defined in the previous activity, let's trigger that movement by a keypress - in computing terminology, something that can trigger an action is called an **event**, and the code or script that responds to an event is called an **event handler**. In this activity, we are responding to a **keypress event**. There was an event in the previous lab as well - do you remember what it was?

To define an event handler, you find the start block associated with that event. In this case it is in the "Control" blocks, and is the one that is labeled "when … key pressed" - drag that piece out and snap it on top of your two blocks from the previous activity. By default, the key that you are responding to is the space bar - change that to "up arrow" to respond to the up arrow key. Next, we'd like to do the same things, but in different directions, for the other three arrow keys. Since the sequence of actions is the same in all cases, but with different arguments for the key and the direction parameters, we'll speed up this task by duplicating the set of blocks you just made. If you right-click on the top block, you'll get a pop-up menu in which you can select "duplicate" - do that, and you'll see a new copy of that set of blocks that is ghosted out and moves with the mouse cursor so that you can place the new copy wherever you want. Do this three times so that you have a total of four event handlers, and set the keys and directions accordingly. After you do this, you should be able to press all four arrow keys and see Alonzo move around in the desired direction. This is an excellent time to experiment some more with the sprite orientation control buttons, and see how the sprite reacts in each of the three possible orientation control settings.

Next, we will add a new sprite that we can move independently to create a little game of tag. Since we want another sprite that moves in a similar way to Alonzo, the easiest way is to duplicate Alonzo and then make the necessary changes. In the sprites pane, right click on the existing Alonzo sprite ("Sprite1" unless you changed the name - which would be a good idea!), and select "duplicate". The next thing to do is to change the looks (costume) of the new sprite, and then change which keys control it. To change the looks, select the new sprite in the sprites pane, select the "Costumes" tab in the sprite info area, and then click "Import" to select a new

costume - pick whichever one you like best! After the new costume is imported, scale it to an appropriate size, and delete the original costume (that one that looks like Alonzo). Finally, go back to the scripts pane and change the keys that trigger movement events so that 'w' moves up, 'd' moves right, 'a' moves left, and 's' moves down (these are the "WASD" controls that a lot of computer games use). Once that is done, try having two different people on the two sides of the keyboard, one using arrow keys to move Alonzo and one using w/a/s/d to move the new sprite. You've almost got a game! [*Warning: You have to "play nice" - because of the way BYOB interfaces with the keyboard, if two keys are held down at the same time it will only "see" the first key that was pressed, so one player can lock the other out simply by holding down a key. There are ways to overcome this problem in some other programming languages, but in BYOB you just need to have a player's agreement that no one is allowed to hold keys down: press and release!*]

The only thing left to do is to detect when the player who is "it" catches the other one. Let's say that Alonzo is always "it", to make things simple. To do this, we need to see another kind of event handler - one that iterates forever, but waits until a collision happens to do anything. Look in the "Control" category for a block that says "wait until …" - this will pause this script until some event occurs or condition is met, so we just need to find the condition. The one we want is under "Sensing" - it's the one that indicates when the current sprite is touching another sprite. Drag that out and put it in the wait block, and select the appropriate argument for the "touching …" block. Finally, right after the wait block, let's have Alonzo say "Got you!" for 1 second. Finish your game by placing this in a "forever" iteration block, capped off with a "when green flag clicked" which will act as a "Start Game" event.

Now you have a game of tag! I'm sure you'll want to play for hours and hours, but for now save your game as "Lab2-Tag" and move on to the next activity!

Activity 3: Working with variables

This activity does not build on the previous one, so once you are sure that your Activity 2 work has been saved, start a new BYOB program. In this activity, you are going to write a script that makes Alonzo move in a natural way. We'll start by simply having Alonzo move smoothly across the stage from left to right. For this activity, it's best if Alonzo were a little smaller, and to do that we need to edit Alonzo's "costume." We edited a sprite's costume in the last lab as well, when we flipped the dragon to face the other way - this time you should shrink the size of Alonzo, so hover over each of the buttons at the top of the sprite costume edit window to find the right one. Each time you click the "shrink" button the sprite drawing should get a little smaller. Three clicks should do it.

We're going to move Alonzo around, so go ahead and select the "Motion" category in the blocks palette. Once you've made Alonzo the right size, move him to the lower left corner of the stage. In the last lab talked about using a "go to x: … y: …" block as a "reset button," and now we'll show you a new trick for doing that. Once you have Alonzo in his starting position in the lower left corner, look at the "go to: …" block and then double-click Alonzo. What happened to the values of the arguments in this block? Compare them to Alonzo's position at the top of the

sprite info pane. Now you can drag this block out into the scripts area for Alonzo, and it can be your reset button. Move Alonzo around and click the block that you dragged out - Alonzo snaps back to his starting position!

Now it's time to make Alonzo move across the screen. We want to repeatedly move Alonzo to the right, 10 steps at a time, and we want to do this by working directly with the x/y coordinates rather than worrying about a direction for the "move … steps" block. In the "Motion" category, you should see blocks for "set x to …" and "change x by …" (and similar blocks for the y coordinate). These work exactly like the set and change blocks that were described in the pre-lab reading, where the x and y coordinates work like variables. Drag the "change x by …" block out to the scripts area, click it a few times to assure yourself that it does what you think it should, and then click on your position reset block to get Alonzo back to his starting position. To avoid clicking for each step, we need to repeat this action several times - or in terminology more commonly used in computing, we **iterate** this action (the general process of repeating actions is referred to as **iteration**). In the last lab we used the "forever" block to repeat operations, and in this lab we want to try a more refined approach. The blocks to control iteration are in the "Control" category of the blocks palette - as a first try, use the "repeat …" block with the number inside (it should have the default value of 10 if you haven't changed it). When you drag this out, position it so that it "eats" your "change x by ..." block (so the change block will snap inside the repeat block). Now click the script that you just made - what happens?

You probably figured out that this repeat block repeats whatever is inside it 10 times, or however many times the number is set to. Try 20. Then 40. Experiment to find the right number of repetitions to move Alonzo all the way to the right without going off the stage. To make Alonzo start in the same position every time, snap your reset position block onto the top of this block by either moving the block or duplicating it and snapping in the duplicate. Alonzo moves pretty fast, right? There are two ways to slow him down - first, you could decrease the distance he moves in each step by decreasing 10 to 5. You can also put in a slight delay after each movement by dragging out the "wait … secs" block from the "Control" category, and snapping it directly under the "change x by …" block (make sure it's inside the "repeat" block!) - change the wait time from 1 second to 0.1 second unless you want Alonzo to move really, really slowly. Think about these two methods of slowing Alonzo down. Which one provides smoother motion? (Optional exploration: See if you can figure out how to use the "glide" block to make a really smooth motion.)

Our next step is to make Alonzo move vertically as well as horizontally, where gravity pulls him down toward the bottom of the stage - we could throw several sprites and call it "Angry Alonzos." Think a little bit about how gravity works: if you throw something straight up in the air, it steadily slows down until it reverses direction and comes back down, speeding up as it falls. One way we can slow things down is to reduce the distance it moves in each step - if we start moving 10 steps, then in the next move we might only go 9 steps, then 8, then 7, etc. We will change the y coordinate of Alonzo in exactly this manner, but to do this we can't just put a number in the "change y by …" block - the amount that y changes by varies from step to step, so we need to keep track of Alonzo's current speed - a perfect job for variables! Define variables for the x speed and y speed: Click "Make a variable" and when it prompts for a variable name, use "x speed" and check that it is "For this sprite only" (this allows you to define different speed

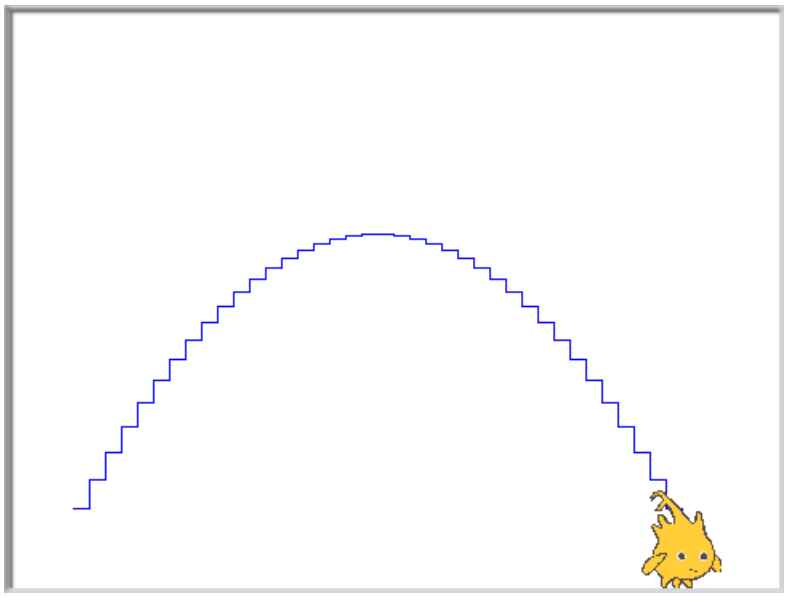
variables for each sprite in a complicated program). Do the same thing for y speed, and unless you just want to watch the values of these variables, uncheck them after they are created.

Since we defined some new variables, we need to make sure they are initialized. Going back to our code block for moving Alonzo, drag out a “set … to …” block, and put it at the top, either before your position reset block or after it. Click on the first argument and change it to “x speed” and make sure the second argument is 10, and then do the same thing to create a block that initializes the y speed variable. Now we will use those speed variables: You should still have a “change x by …” block, so drag out “x speed” from the Variables category, and snap it in place right over the number. Make a similar block to change y, and use the y speed. Finally, we want to change the y speed at each step, just like gravity does - drag out a “change … by …” block and put it inside the “repeat” loop but below the “change” blocks, and have it change the y speed by -1. Your blocks should now look something like this:

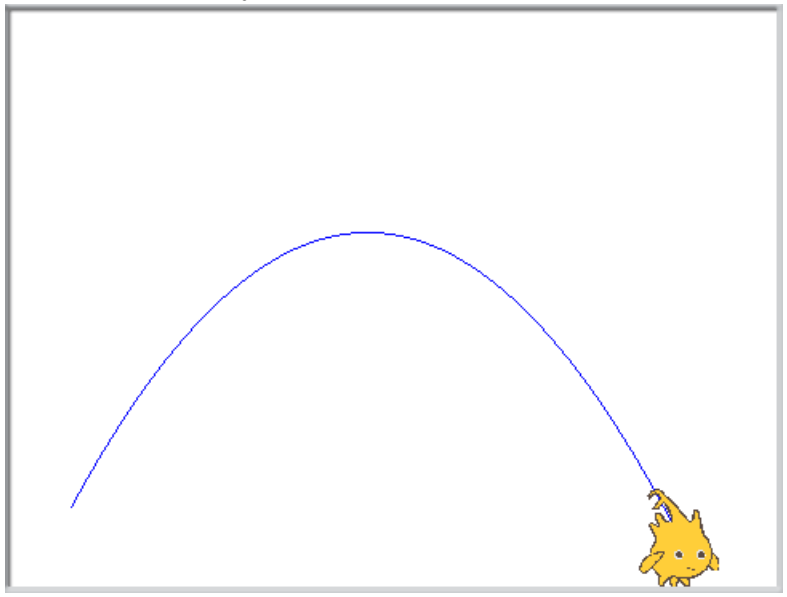
Try it! Does it look like a projectile flying through the air? Adjust the initial values for x speed and y speed to see how that changes things. Finally, the repetition count of 40 is just a value we put in by trial-and-error, trying to find something that worked out. In this situation, what’s a better way to do this? Could we stop as soon as we hit the ground? For this, we repeat until the y coordinate gets below some level that we call the ground - since my Alonzo started out at y coordinate of -130, I’m going to call y coordinate of -135 the ground, and repeat until y is less than -135. Our first step is to replace the “repeat 40” block by the “repeat until …” block, and then we construct the following condtion for this repeat block:

Now play around with initial x and y speeds so that Alonzo makes a nice long arc from one side of the stage to the other. Alonzo is flying!

Finally, consider how smooth Alonzo’s motion is - it looks a little jerky to me, but it would be good to see the exact path that Alonzo takes. Fortunately, BYOB makes this easy through the use of “pens.” A pen can be associated with any sprite, so that when the sprite moves - if the pen is “down” - it will draw out a path of where it has been. To see how pens work, click on the “Pen” category in the block palette and think about what we want to do. First, we want to raise the pen (“pen up”), then clear any drawing it has made (“clear”), and then after moving to the starting position we want to put the pen down (“pen down”). Place these three pen control blocks in appropriate places in your Alonzo movement block, and shoot him across the screen again. Now you should see something roughly like this on the stage:



See how bumpy Alonzo's motion is? Think about why this might be, and how you could correct it. The final thing you will do in this activity is to change the way Alonzo moves to make it more smooth - look through the Motion blocks and see if you can figure out how to make the movement smoother. There is a very simple solution that will result in the following picture:



See if you can figure out how to do this. [*Hint: the stair-step pattern of movement comes from changing x then changing y - can you set both x and y in a single step? In addition to a different motion block, you will need to use other "Operators" blocks, such as + for addition.*]

Save your best solution as "Lab2-Flying".

Activity 4: Displaying numbers

This activity does not build on the previous one, so once you are sure that your Activity 3 work has been saved, start a new BYOB program. In this activity, we will explore some additional

ways to draw, similar to the pen commands in the previous activity. Here's the problem we want to solve: We want to display a number (for now, a two-digit number) using drawing commands - in other words, without using "say" or "think" or setting a watch variable. A warning ahead of time: you would think that, since computers are good with numbers, this would be easy to do - it's not! At the end of this activity you'll end up with the longest script that you've created so far, and it will be a bit ugly. To make you feel a little better about this, you should know that there are ways to clean this up and make it nicer and more concise - and in fact, we'll continue with this example next week to introduce some of the concepts you need to make a more elegant solution.

Our first step is to figure out how to draw numbers on the screen. Bring up a new, blank project (with Alonzo sitting in the middle of the screen), and select the "Pen" category in the blocks palette. Click on the "stamp" block, and then move Alonzo around on the stage. What happened? Do another "stamp" in a new position for Alonzo, and keep moving around. So now we can draw more than just lines: we can stamp out images of our sprite costumes onto the stage! You click on the "clear" block in the blocks palette to clear off your "stamps" once you've got the idea. You can either delete the Alonzo sprite now, or just move him out of the way - down to the bottom of the stage, for instance.

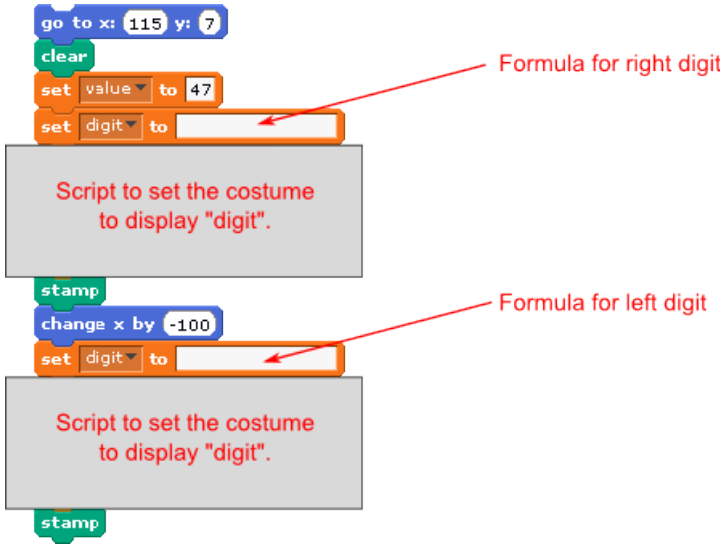
Given that we can stamp out pictures based on sprites, what if the sprite that we were using looked like a digit? Go to the sprites pane, and create a new sprite using a provided image - do you remember how to do this? You want the button where the "hint" that pops up says "Choose new sprite from file." Look in the provided costumes under the "Letters" category (I know, I know, … we want numbers, not letters, but trust me on the category), and browse through the different letter styles to find one you like. For this lab, you can use any style you want - what looks good to you? Once you have decided on a style, click on the "0" number in that style to create the sprite. Now the boring part: we need to add "costumes" to this sprite for every other digit - select your new sprite (which should look like a zero!), go to the "Costumes" tab in the sprite info pane, and click "Import" to load in a second costume, the one that looks like a 1 (one). Repeat this for all the other digits - it's important that you do them in order, so that in the end you have 10 costumes, looking like the digits 0 through 9, in that order. Play around a little bit with your new sprite - use "stamp" to leave an image, and use "switch to costume" to change which digit is displayed.

Our first task is to stamp out a single digit: a value between 0 and 9. Remember the discussion of "problem decomposition" in the previous lab? That's what we're doing here: breaking down the problem we really want to solve (stamping out two digits) into a smaller problem of stamping out a single digit. Once we have that figured out, we'll attack the larger problem. To get started, go to the "Variables" category in the block palette, and make a variable named "digit". Should this be "for all sprites" or "for this sprite only"? You may not have the experience to make a fully informed decision on this, but think through the possibilities, and make a choice. We'll discuss this further in class.

Now that you have a digit variable, drag out a "set" block that will initialize it to some value that is 4 or greater, and our goal will be to "stamp" out this digit. The problem we have now is this: how can we select the right "costume" for our digit, based on the value of the digit? Take a look

at costume changing blocks under the “Looks” category. Unless you’re particularly brilliant, the solution to this problem isn’t immediately clear - however, this is one of the things about computing that can be both very frustrating, since what you want to do isn’t immediately available, and also very very cool, because you get to “think outside the box” and come up with creative solutions which aren’t obvious. You can be a problem solving ninja! You have seen all the pieces you need to solve this problem in this lab: you have a block that can set the sprite to use a particular costume (like the first one), and you have a block that will advance to the “next costume”, and you also have a block to repeat an action a certain number of times. See if you can put those ideas together so that you have a script that advances the costume to the correct digit - then try changing the value that “digit” is set to and see if it still works. (Hint: Try dragging out the oval representing the “digit” variable and putting it into various places, such as the parameter for the “repeat …” block - what do you think this does?) Test your script for several digits and make sure you consistently get the right answer - you could try all digit values, but at the very least make sure you test the extremes: Does it work for zero? Does it work for 9? **Software testing** is an important skill for developing high quality software, and entire courses can be taught on how to do software testing of complex systems - although for small scripts like we use in this class, common sense testing (as long as your “common sense” includes testing with both common and rare or extreme values) is good enough.

Now you should have a script running that stamps out whatever single digit value you want. To move to stamping a two-digit number, you need to be able to do a few things: Given a two-digit “value,” calculate the individual digits and stamp in the appropriate positions. We described how to calculate individual digits of a two-digit number in the pre-lab reading - hopefully you understood that, because if so you are ready to go! The whole process can be described like this: calculate the rightmost digit, stamp it out, move to the left, calculate the leftmost digit, and stamp that out. I’ll give you the skeleton for this, without giving away the single digit stamping part (you should have figured that out already!):



You should already have the “Script to set the costume to display digit” part written and tested, so you can use the “duplicate” capability to make a second copy to save you some time. Once this script is written, you should be able to change the value of the two digit value at the top to

any other value (not just 47) and it should work. Test it on a few different values until you are confident that it works.

Hopefully you have the basic **functionality** (how your script behaves) working correctly now, and you should spend a some of the time remaining in the lab session to refine your script so it works better. For example, do you really want to see the digit values scrolling by with each "next costume"? Maybe you could have them just appear with the right values - experiment with the "show" and "hide" blocks to see if you can make this happen. What about the size of the digits? They're pretty large right now, so you'll need a way to scale this down if you want a more compact number being drawn - experiment with "set size to … %" and see how you can adjust the size.

Once you are happy with your script, save it as "Lab2-Digits".

Discussion (Post-Lab Follow-up)

Animation smoothness: In creating "Angry Alonzo," there were several different ways to move the Alonzo sprite around the screen, and some were smoother than others. Since we simulate motion by cycling through a series of still images (or "frames"), there are two main things that control the smoothness of the motion: how consistent the motion is between frames, and how far things move between frames. Changing just the x coordinate between two frames and then just the y coordinate between the next two frames leads to jerky horizontal-then-vertical movement, giving a stairstep path (like the first path drawn by Alonzo in the activity you just did). Even with consistent motion, however, if a sprite moves too far between frames the motion will look jumpy, and to smooth this out you need to move a shorter distance between frames. For example, you could half the distance that the sprite moves so that it doesn't "jump" so far between frames, but to do this while keeping the overall movement the same you will need to double the number of frames you display every second. This is called the "frame rate" and people who play video games can be a little fanatical about frame rate - they will spend lots of money on faster video cards, so that they will draw the frames at a higher frame rate, and hence produce a smoother game. When people benchmark (i.e., test the speed of) video cards, they will often do it by testing how high a frame rate can be achieved for certain games. For reference, movies in a theater typically run at 24 frames per second, while TV signals are around 30 frames per second in the United States. People who make movies also think about frame rate - in the recently released movie *The Hobbit*, Peter Jackson filmed everything at 48 frames per second, and certain theaters (who buy new projection equipment) played the movie at that frame rate - did you see this movie? Do you think you would notice the difference?

Modeling Physics: "Angry Alonzo" flew across the screen in what was (hopefully) a fairly realistic looking arc. There are a lot of situations where we want computations performed by a program to reflect actions as they would happen in the real, physical world. The mathematics that gets turned into code is called a "model," and is a way to simulate an action in the real world. What we did in the activity in this lab is to create and use a very crude model of gravity: When an object is subject to gravity (at least an object that is not on a small, atomic scale), the vertical velocity changes at a steady rate. Specifically, the downward velocity increases by 9.8 meters/second (or m/s) every second. Therefore, if an object is dropped from a great height, it

is released from a standstill (velocity 0 m/s) - one second later it is traveling down at 9.8 m/s, and two seconds after release it is traveling down at 19.6 m/s (or 2*9.8 m/s). Lots of other factors affect the velocity of an object, including air resistance, but this is a reasonable approximation for now. Since we change the velocity steadily in our "Angry Alonzo" activity, we are roughly approximating the effect of gravity on Alonzo's vertical velocity. Of course, even if we used real gravity values, this wouldn't be a great approximation: we "jump" the velocity from 9.8 to 19.6 rather than smoothly varying it, and we have Alonzo travel at 9.8 m/s for an entire "time step" until increasing the velocity. Similar to our discussion of animation smoothness, we can increase the accuracy of our physics model by making the time step smaller: change the velocity every half second, so Alonzo travels at 9.8 m/s for half a second, 14.7 m/s for the next half second, and then finally increases to 19.6 m/s. If we keep decreasing the timestep, we approach a situation in which the velocity is continuously changing - this is what Calculus is all about, and using Calculus we could make even more accurate models! Physics models can get very complicated, and most people who write games use a software library that provides a ready-made physics model. For example, the Angry Birds game (and many others) uses a physics library called "Box 2D" - using a library that someone else has written, so you can use it in your program without having to worry about the details, is an example of "abstraction." Abstraction is one of the "big ideas of computing" that we will discuss much more over the next few days in this class, and is probably the single most important idea that enables the construction of complex and useful software systems.

Elegant Solutions: Programmers often talk about a solution or program being "elegant." The solution to displaying two digits in this lab exercise was definitely not elegant, which motivates the need for more powerful techniques that enable a more elegant solution. There is definitely a sense of aesthetics when it comes to programs, and people who are fluent in the basic language of computing recognize this and talk about "beautiful" or "elegant" code. Code that is wasteful, or repeats things unnecessarily is not elegant. Code that jumps all over the place so that it is hard to follow (sometimes called "spaghetti code") is not elegant. Code that uses an inefficient algorithm so that it runs slowly is not elegant. It's harder to explain what *is* elegant, but it's a lot like art - you know it when you see it, at least once you speak the basic language. Elegance in code comes from being concise, efficient, clear in purpose, and organized on the page or screen in a way that reflects good design and aesthetics. Elegance is such a central notion in computing that David Galernter, a computer science professor at Yale University, wrote a book on the subject entitled *Machine Beauty - Elegance and the Heart of Technology*, in which he wrote "Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity."

Terminology

- *debugging*: The process of observing code as it is running, typically in order to track down some part of the code that is not working correctly.
- *event*: Something that happens in a computer system that could potentially have a "response action" defined. Examples of events include things like the user pressing a key, an internal timer running out, a message being broadcast, etc.
- *event handler*: The script, or code, that is designed to run when a particular event occurs.

- *functionality*: How a program operates. In computing we can evaluate a program by several criteria, including functionality, robustness (how it responds to unexpected or erroneous inputs or events), and elegance.
- *global*: A variable that is not associate with a specific sprite (so isn't local) and is common across an entire program, like a score or a timer.
- *initialize*: The act of giving a new variable a starting (initial) value.
- *iterate*: To repeat some action.
- *iteration*: The general idea of repeating an action.
- *keypress event*: An event that is generated when the user presses a key.
- *modulo operation (mod)*: An operation which gives the remainder after a division. For example, dividing 14 by 6 results in a remainder of 2, so "14 mod 6" is 2.
- *nesting* or *nesting order*: Using one block inside as an argument to another block is referred to as nesting the blocks, and the order that they are inserted (showing which block is inside what other block) is called the nesting order.
- *radio buttons*: Input elements in a program that typically look like buttons, where only one of the buttons can be selected or active any any point in time.
- *reporter*: A type of BYOB block that calculates a value, like from a formula, and reports the value so that it can be used as an argument to another block.
- *set*: Storing a value in a particular variable.
- *software testing*: The process of trying code with different inputs and combinations of events and inputs to see if it responds as it should.
- *sprite orientation control buttons*: In BYOB, these buttons - specific to each sprite and located in the sprite info pane whtn that sprite is selected - determine how a sprite responds to changing its direction. Does the picture of the sprite rotate to point in the direction that it is moving?
- *store*: A synonym for "set" when it comes to variables.
- *variable*: A symbolic name that can take on different values as a program is running, and can be changed by the program.
- *watch variable*: A variable that has been designated as a "variable of interest," typically when a programmer is debugging a program. Watch variables have their current value displayed while the program is running, so that the programmer can watch the variable as it changes.

Submission

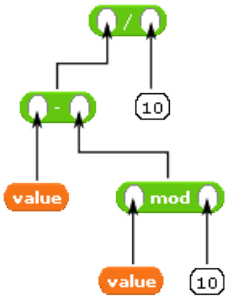
In this lab, you should have saved the following files: Lab2-Tag, Lab2-Flying, and Lab2-Digits. Turn these in using whatever submission mechanism your school has set up for you.

Solution to Computing Leftmost Digit

Hopefully you were able to figure this out for yourself, but since we haven't done a lot of nested operator blocks inside, the solution might be a little difficult at this point. Here it is:

Proper nesting of the blocks is critical to getting this right, so the diagram below breaks this out to make the nesting more clear. This kind of diagram is called a "parse tree", and it shows

specifically what the formula is doing and which blocks should be inside other blocks. You should study this carefully, particularly if you don't understand how to build the formula up from the concepts described in the lab.

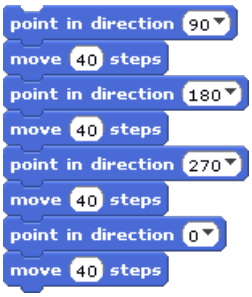


The actual computation starts at the bottom of this diagram: the (two-digit) value is first taken mod 10, to get the remainder. This result is then subtracted from the value, giving a two-digit number that is divisible by 10. Finally, we do the division by 10, giving the leftmost digit.

Answers to Pre-Lab Self-Assessment Questions

1. If a sprite's direction is set to 135, and it starts in the center of the stage and keeps moving in this direction, where would it go off the stage?
   Answer: Looking at the directions, this would cause the sprite to head down and to the right, so it's tempting to say that it will go off the lower right corner of the stage. However, you also need to keep in mind that the stage is wider than it is high, so in this direction it will hit the bottom edge before it makes it over to the right side of the stage. Therefore, the correct answer is "it will go off the bottom edge of the stage, close to but not quite at the lower right corner."

2. What shape does the following BYOB script trace out?



   Answer: This draws a square, with the starting position being the upper-left corner of the square. In particular, this moves right, then down, then left, then back up to the starting position (to close the square).

3. Consider a game in which every character has a certain amount of gold. Would you use a local variable or a global variable to keep track of these amounts?
   Answer: A local variable would be most appropriate here, since each character (or sprite) would have its own amount of gold - it's specific to each sprite.

4. Consider a game that keeps track of how many enemies are left in the game. Would you use a local variable or a global variable to keep track of this?
   Answer: A global variable would be most appropriate: There is only one "count" of how many enemies there are, and enemies can appear and disappear (die) during the game, so we certainly wouldn't want a local variable tied to an enemy that can disappear!

5. What is the value of the following BYOB formula?

   Answer: 12. You have to be careful to look at the nesting of this formula. The first calculation that is performed is 11-2, giving 9. Next it multiplies 4 times 9, giving 36. Finally, it divides this by 3, giving 12.

6. What is the value of the variable "mystery" after the following script runs?

   *Answer: At the end of this script, mystery has the value 14. Here's how it executes: First, mystery is initalized to 0. The second block adds 1 to mystery, making the value 1. The next block, the "set" block, calculates 2*mystery (which is 2), and stores that as the new value of mystery. So after the first three blocks execute, mystery has value 2. The next block adds 1 again (making mystery 3), followed by another doubling of the value (making mystery 6). Finally, the last "change" block increases mystery by 1 (making it 7), and then the very last block doubles this again, giving 14. If you take this a step at a time, it's really not difficult - just make sure you keep track of the "current value" of mystery after every step.*