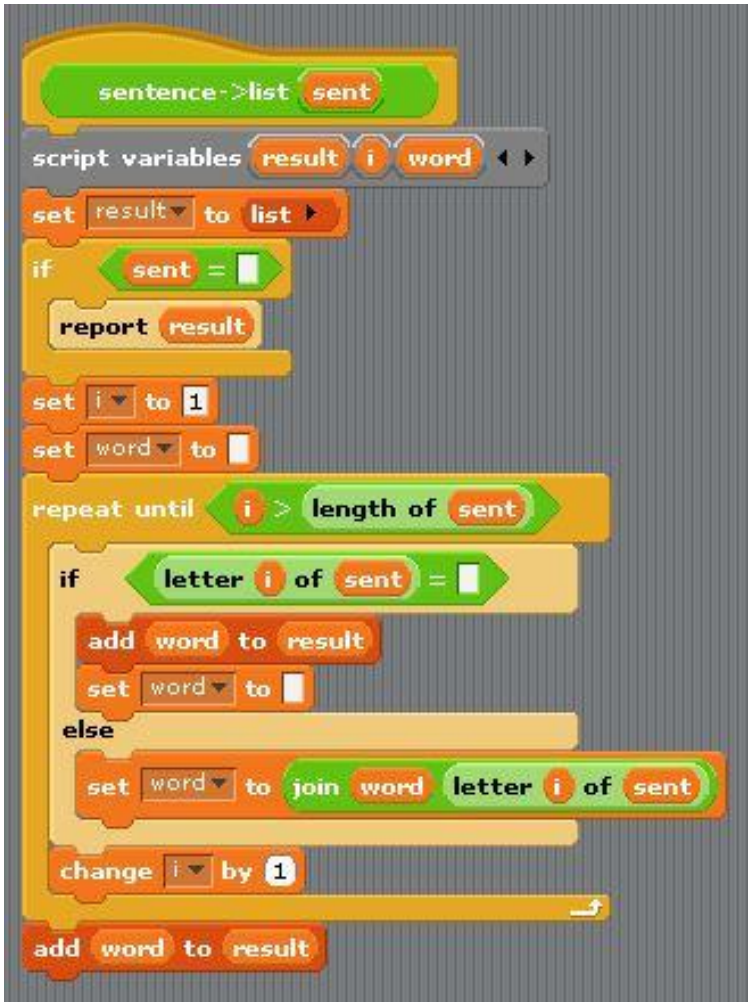
# Higher Order Functions

Computer Science Principles

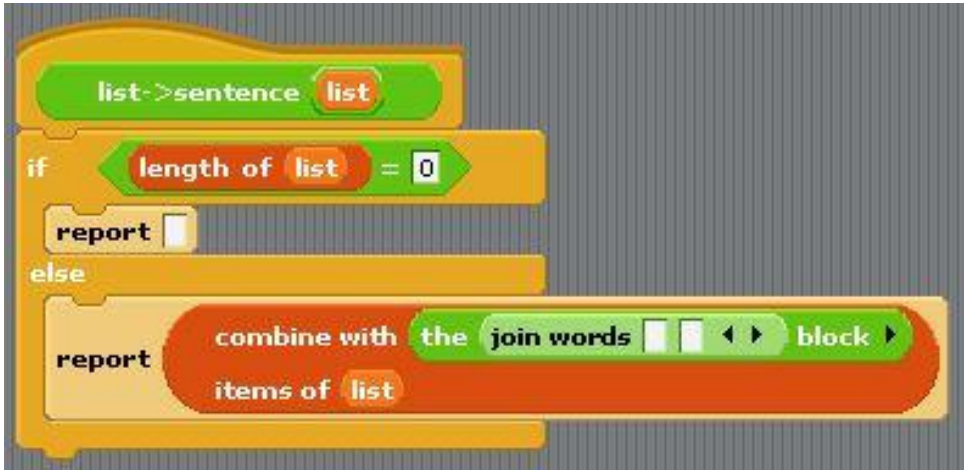
# Handy Blocks For This Section

- Sentence->List
  - Puts each word of a sentence into a list
- List->Sentence
  - Takes the words from a list and builds a sentence

# Sentence->List



# List->Sentence



# Adding Tools

- Need to add more tools so you will import the Tools sprite. How?
- Method 1:
  1. BYOB, go to "File" on top and click on "Import Project".
  2. In the "Import Project" window, locate your BYOB folder and find the "tools" file. Click on the file, then press OK.
  3. After checking that the new blocks are in their palettes, delete the tools sprite in bottom right

# Adding Tools

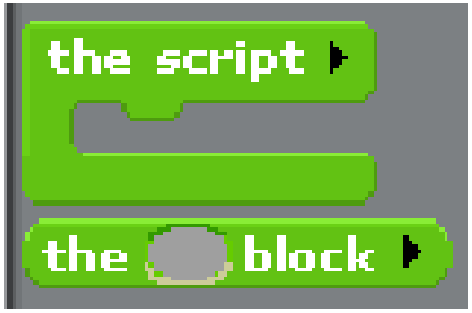
- Method 2:
  1. Open BYOB
  2. Locate your BYOB folder (if you didn't open BYOB from there) and find ToolSprite.ysp.
  3. Now simply click and drag ToolSprite.ysp anywhere in the opened BYOB window
  4. You will now see all the new blocks that were imported from the tools sprite in each of the palettes, and you can read the description of each of them if you like in the workspace.
     - After you are done, you can delete the sprite that was created.

# Higher Order Functions

- Blocks & scripts can be used a ***data***.
- How?
  - Put the block or script into the program used as an *action*
    - BYOB will perform that action
  - Need to ***encapsulate*** that block or script inside another special block called a **wrapper block**.
    - Encapsulate just means to "wrap around or wrap up"

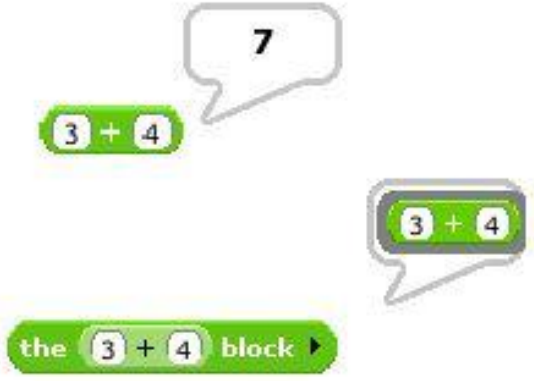
# Wrapper Blocks

- BYOB Wrapper Blocks
  - The block
  - The script
- We will focus on The Block right now.
- Find them in the Operator menu



# The Block

- **The block** takes a block as input, and reports the block itself.
- Compare these two interactions and you'll see what that means.

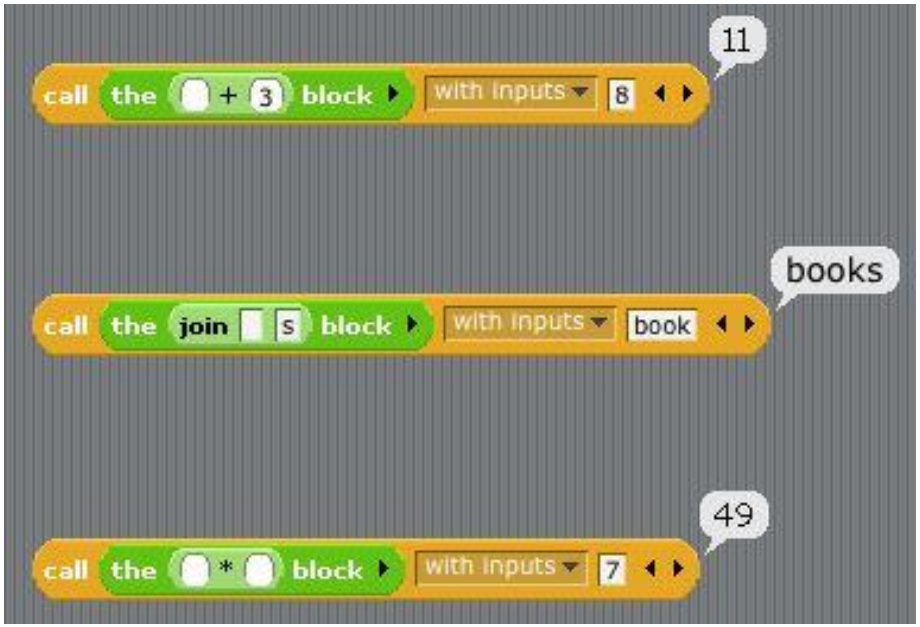


# The Call Block

- So why? We can  call the function represented by the block — to run, evaluate, or call the block.
- This is where the **call** block comes in.
- The **call** block (Control palette) takes an encapsulated block and evaluates it.
  - If we put the block of 1+1 into the **call block**, 1+1 should now be evaluated and return 2.

# The Call Block

- We can take a block with *unfilled* input slots, and **call** it with the necessary input values provided.
- To call a block with one or more input values, click the right-facing arrowhead at the end of the **call** block.
- Here are some examples



# Higher Order Functions

- **Higher Order Functions** (HOFs) are "higher order" because they are functions that take another function as an input.
- Here's a silly example just to show what we mean.
  - The function **call-with-5** takes a function as input, and reports the value you get by calling that function with the input value 5.
  - You can use any function as the input.

# Higher Order Functions

- It turns out that higher order functions are particularly useful along with lists.
- In this lesson we will mainly use three HOFs that take a function and a list as inputs: **Map**, **Keep**, and **Combine**.
- They are found in the "Variables" palette, with the other list blocks.
- Even though we emphasize them, keep in mind that they aren't the only higher order functions; in fact, you can write your own.

# Higher Order Functions

- All of the three main HOFs take a function and a list as inputs, and the function will somehow be applied to the list items.
- The result varies depending on the HOF.
- In BYOB, HOFs are all reporters

# Higher Order Functions

- If the reported value is a list, it's a newly constructed list; these blocks do ***not*** modify their input lists.
- This style of programming, in which existing values are not modified, is called ***functional programming***

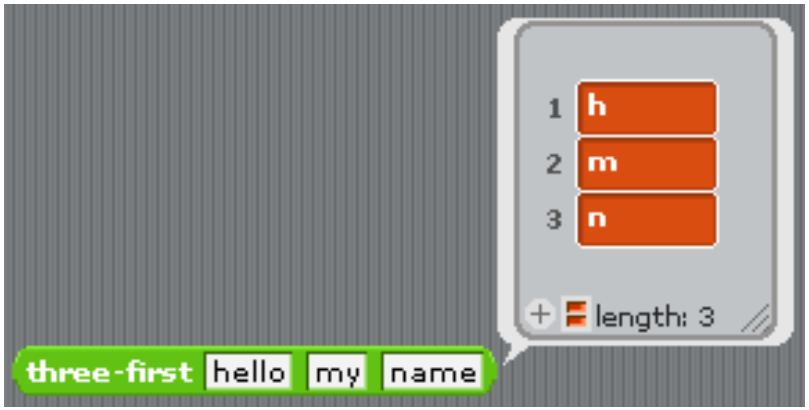
# THE MAP BLOCK

# The Map Block

- You decide that you want take the first letters of a couple words using the "letter _ of ___" block in a list.
- To start with a simple case, suppose you have a list of two words.
- You could apply the "first" block to each of them and make a list out of the results

# The Map Block

- Similarly, here's the version for three words in a list



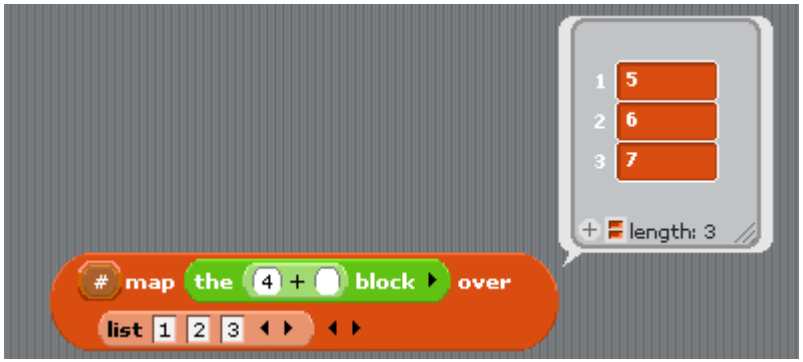

- This approach, though, would get very redundant if you had a list of five words - you'd have to make blocks specifically for the case of five words. You don't want to keep making blocks for a specific case.
- Of course, you can use recursion to solve this problem. But you can also use the Map block.

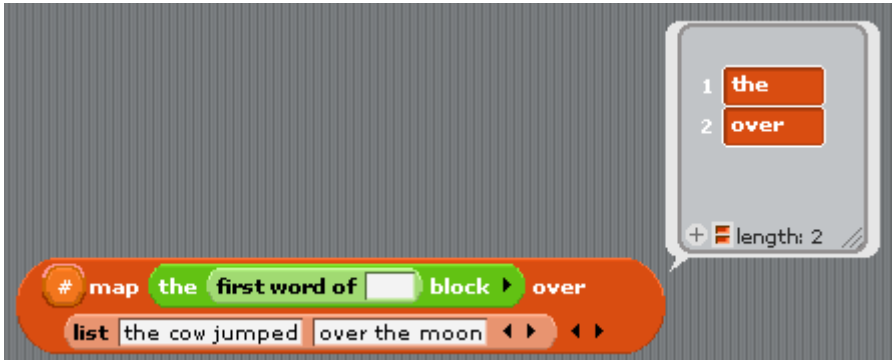
# The Map Block

- **Map** takes a reporter block and a list as inputs.
  - It computes a function of each item of the list and report a list of resulting values.
  - The first figure is an abstract representation of how map would apply to an arbitrary function **fn**; the others are examples of how to use **map**.
  - You'll notice that at the left end of the map block is an orange circle containing a number sign.
    - Ignore that for now; it's a rarely-used feature that we'll get back to later.

# The Map Block

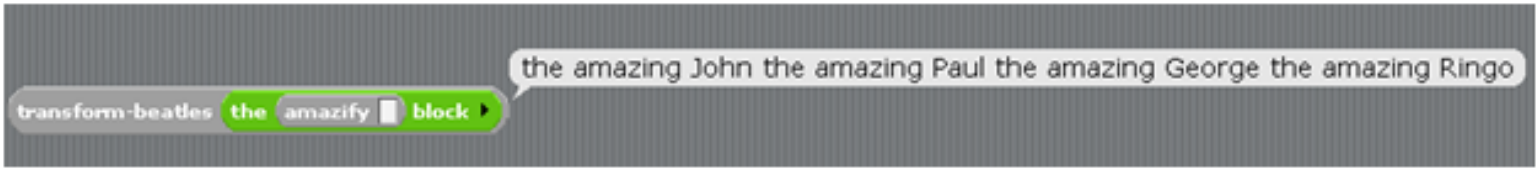




# Transforming the Beatles

- Make a "transform-beatles" block that takes a reporter as an argument, applies it to each of the Beatles (John, Paul, George, Ringo), and returns the result in a sentence.



# Transforming the Beatles

- Use the "`list->sentence`" block, and *make sure you use a list* in the script that defines this block.


  - Note that transform-beatles does *not* take a list or a sentence as an input.

- We can use this to "amazify" the Beatles!

# Transforming the Beatles

- Let's create the Amazify procedure first.

# Transforming the Beatles

- Now, let's drop the Amazify procedure into "the Block" so we can call it when we need it.

# Transforming the Beatles

- Create a reporter block called transform-beatles with an argument called block.
  - Why call it block? Because our Amazify block is going to be the argument.

# Transforming the Beatles

- Now let's create the transform-beatles script.
  - Let's think about what we want to happen… We want to add or map "the amazing" to each name in our list. So let's grab the `map` block.

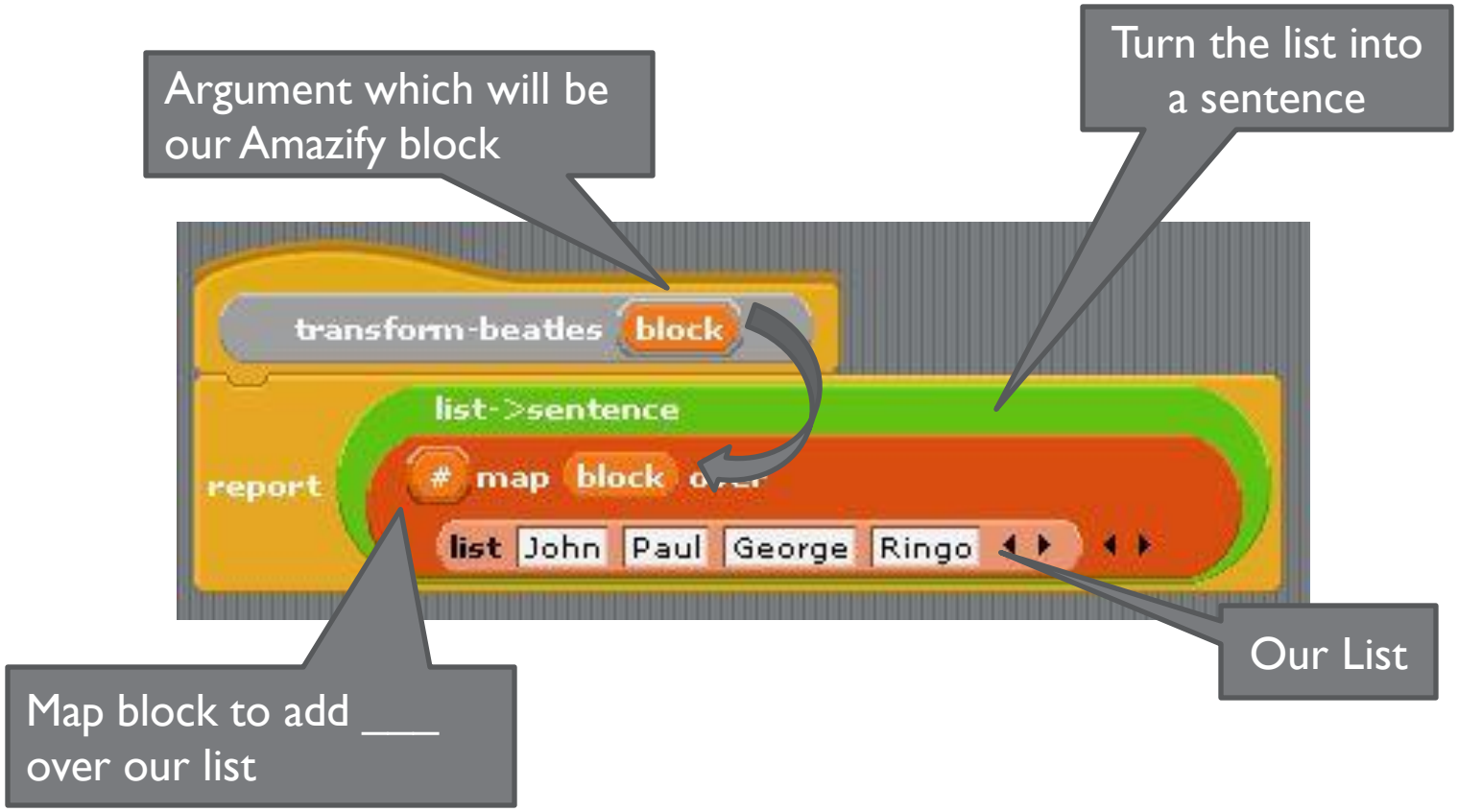

  - Add `block` (our argument) and a `list` with all of the names.

# Transforming the Beatles

- Let's drop the `map` (block) `over` block into a `list to sentence` block so that we get the output as a sentence.


- All that is left is dropping our block into a report block and adding it to the command.

# Transforming the Beatles



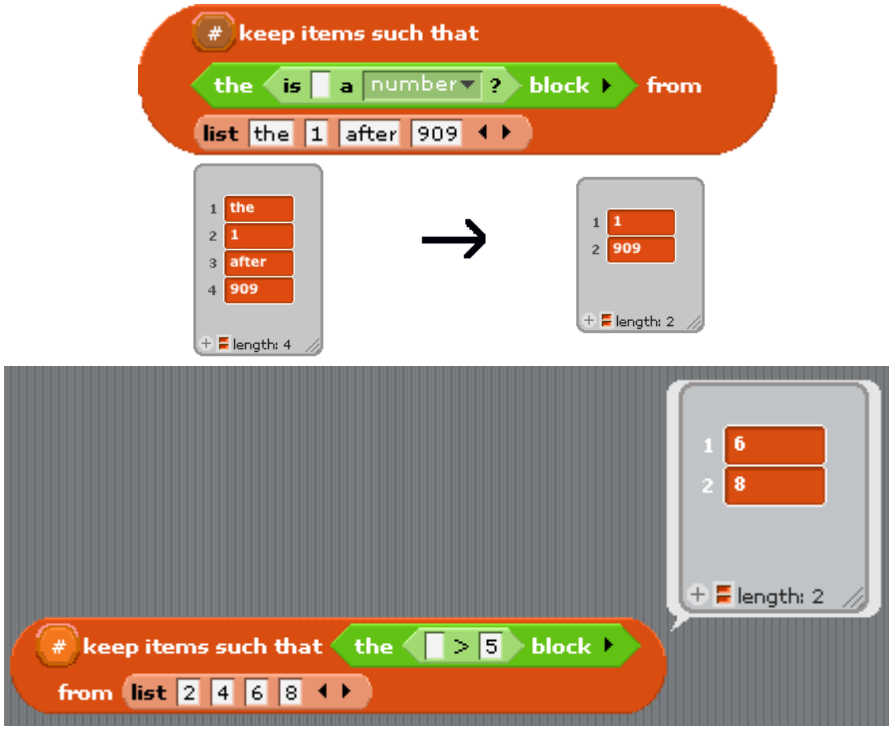
# Map Exercises

- Make the "first-letters" block that was introduced using map.
- Given any list of numbers as an input, use map to add each of those numbers in the list by ten.
- Given any list of words as an input, use map to add a letter "s" to the end of those words in the list.

# THE KEEP BLOCK

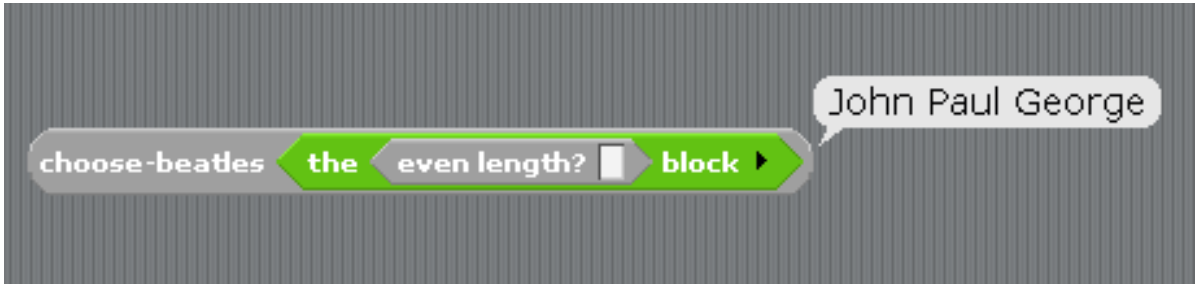
# The Keep Block

- **Keep** takes a predicate block and a list as inputs, and reports a new list containing the subset of that list for which the predicate reports TRUE.



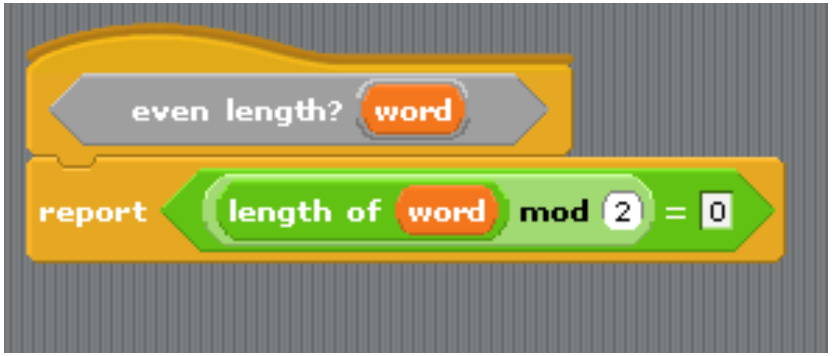
# Choose Beatles

- Make a "choose-beatles" block that takes a predicate block and returns a sentence of just those Beatles (John, Paul, George, and Ringo) that satisfy the predicate.
  - Note 1: Your task is to write CHOOSE-BEATLES, not EVEN LENGTH?.
  - Note 2: CHOOSE-BEATLES doesn't take a list of Beatles as input; its only input is the predicate block that specifies which of them to keep in the result.



# Choose Beatles

- Let's first create the Even Length? Procedure
  - It should report true or false (predicate)
  - Remember mod is remainder division; so if the length of the work mod 2 = 0 then it has an even number of letters.



# Choose Beatles

- Now let's create the Choose-Beatles reporter.
  - Add an argument called predicate. This is where we will drop the Even Length? Block.

# Choose Beatles

- Grab the `keep` block and drag the argument "predicate" into the empty predicate spot.
- Drag a `list` block into the empty list spot and add the names.

# Choose Beatles

- Since we want our answer as a sentence, grab a `list->sentence` block from the operators area.
- Drag the keep block into empty list spot

# Choose Beatles

- Grab a `Report` block (Command) and drag the `list->sentence` into the empty report space then click it into the header block.

# Choose Beatles


Argument

Turns the list into a sentence

The keep block will test each item in the list given the predicate block that will be the argument

The List

# COMBINE

# Combine

- **Combine** also takes a two-input reporter block and a list as inputs.
- Combine uses the two-input reporter block to combine items from the list one by one into a single value.
  - For example, the block shown below starts with 4, then computes 4+5=9, then computes 9+6=15.

# Combine

- Unlike MAP and KEEP, COMBINE really makes sense with a small number of reporter blocks as input, because the operation has to be associative (giving the same answer no matter how the operands are grouped).
- The blocks you'll use often with COMBINE are +, * (but not — or /), MAX, MIN, JOIN, and JOIN WORDS (which is in the tools package).

# Letter Count

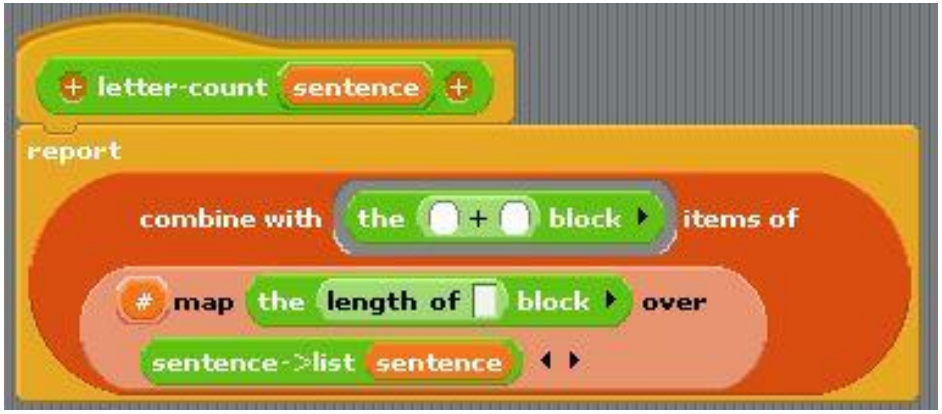
- Make a reporter block "letter-count" that takes a sentence and reports the total number of letters in the words of the sentence (not counting spaces between words).

# Letter Count

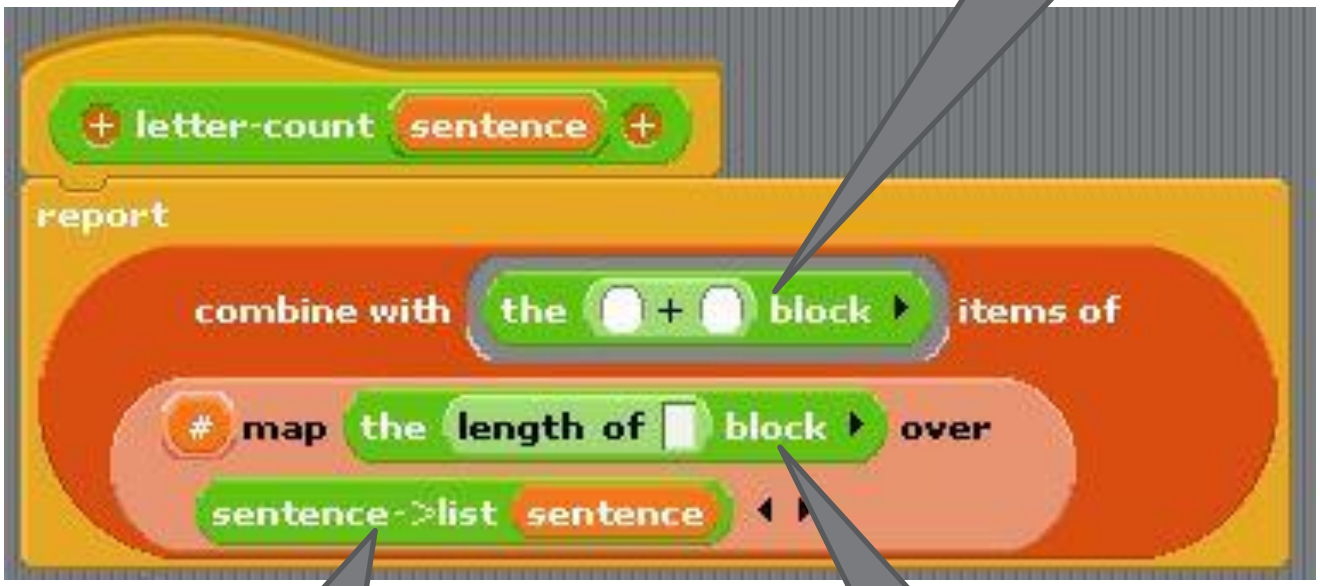
- You will need to add the length of each word over our string.
- First grab a `map () over ()` block.
- You will need to get the `length of []` block (to get the length of the word) then drop it into a "`the()block`" block.
- Get a `sentence->list` block with the sentence argument to drop into the empty list spot.

# Letter Count

- Grab a `combine with` block
  - Grab a `( ) + ( )` block and drop it into "the block" block
  - Drop your `map` block into the empty "items of ()" space.
- Grab a reporter and drop in your block.



# Letter Count




Calculates the running total

Converts the sentence into a list of words

Gets the length of the word