3.1

BYOB Reference Manual

Brian Harvey

Jens Mönig

Contents

BYOB Reference Manual

Version 3.1 DRAFT

BYOB is an extension to Scratch (http://scratch.mit.edu) that allows you to Build Your Own Blocks. It also features first class lists, first class procedures, and first class sprites (explained below). This document is a very terse explanation of the main features, probably not a good tutorial. It assumes that you are already familiar with Scratch.

I. Building a Block

A. Simple blocks

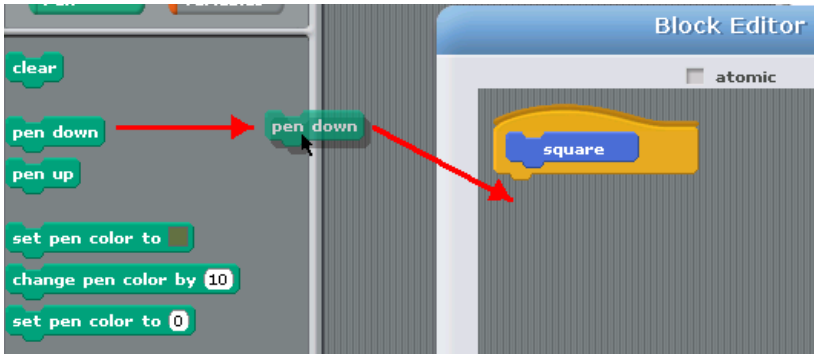
In the Variables palette, at the bottom, is a button labeled "Make a block." (A *palette* is one of the eight menus of blocks you can select in the leftmost column of the BYOB window.)

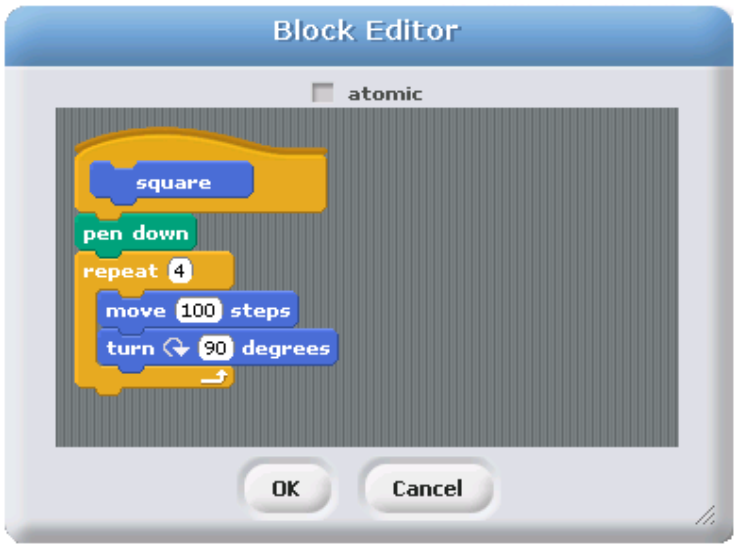
Clicking this button will display a dialog window in which you choose the block's name, shape, and palette/color. You also decide whether the block will be available to all sprites, or only to the current sprite and its children. Note: You can also enter the "Make a block" dialog by right-click/control-click on the script area background and then choose "Make a block" from the menu that appears.

For the most part, there is one color per palette, e.g., all Motion blocks are blue. But the Variables palette includes the orange variable-related blocks and the red list-related blocks. Both colors are available, along with an "Other" option that makes grey blocks in the Variables palette for blocks that don't fit any category.

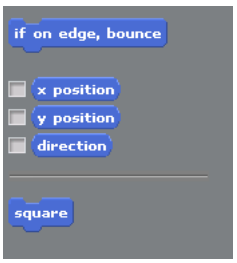
There are three block shapes, following a convention that should be familiar to Scratch users: The jigsaw-puzzle-piece shaped blocks are Commands, and don't report a value. The oval blocks are Reporters, and the hexagonal blocks are Predicates, which is the technical term for reporters that report Boolean (true or false) values.

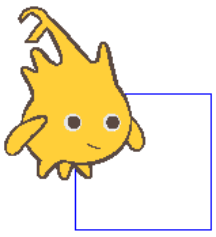
Suppose you want to make a block named "square" that draws a square. You would choose Motion, Command, and type the word "square" into the name field. When you click OK, you enter the Block Editor. This works just like making a script in the sprite's scripting area, except that the "hat" block at the top, instead of saying something like "when Sprite1 clicked," has a picture of the block you're building. This hat block is called the *prototype* of your custom block.[1] You drag blocks under the hat to program your custom block:





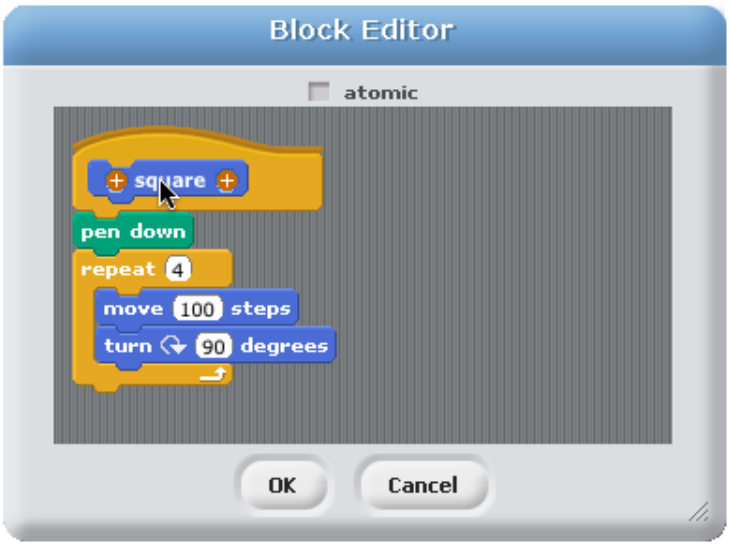
Your block appears at the bottom of the Motion palette. Here's the block and the result of using it:



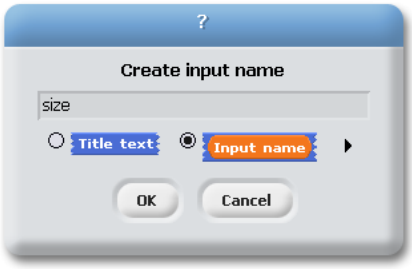


[1] This use of the word "prototype" is unrelated to the *prototyping object oriented programming* discussed later.

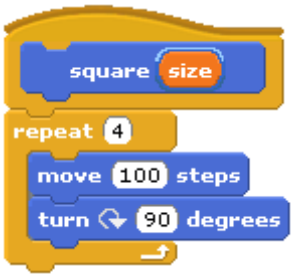
But suppose you want to be able to draw squares of different sizes. Control-click or right-click on the block, choose "edit," and the Block Editor will open. If you hover the mouse over the word "square" in the prototype in the hat block, you'll see two circle-plus signs appear next to it.



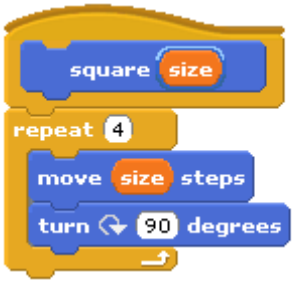
Click on the circle-plus on the right. You will then see the "input name" dialog:



Type in the name "size" and click OK. There are other options in this dialog; you can choose "title text" if you want to add words to the block name, so it can have text after an input slot, like the "**move ( ) steps**" block. Or you can select a more extensive dialog with a lot of options about your input name. But we'll leave that for later. When you click OK, the new input appears in the block prototype:



You can now drag the orange variable down into the script, then click okay:



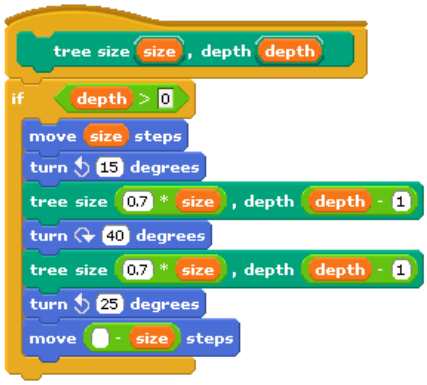
Your block now appears in the Motion palette with an input box:

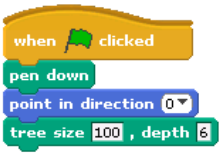
You can draw any size square by entering the length of its side in the box and running the block as usual, by double-clicking it or by putting it in a script.

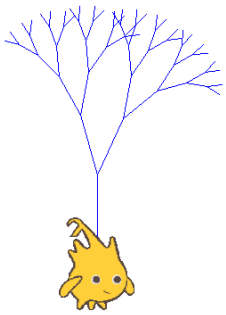
At the top of the Block Editor window is a check box labeled **atomic**. When this box is checked, the block's entire script is carried out in a single BYOB execution cycle. (As in Scratch, the way multiple scripts are able to operate more or less in parallel is that each script gets to carry out one step (one loop iteration, generally) and then each script gets another turn.) The advantage of atomicity is that the script runs faster. The disadvantage is that if the script draws a picture, or does a dance, the user won't see the individual steps happen but will instead see a longish pause followed by the appearance of the complete picture at once, or the last step of the dance. (You can make atomic custom blocks run non-atomically temporarily by holding down the ESC key.) The default is that the box is checked for reporter blocks, but not for command blocks.

B. Recursion

Since the new custom block appears in its palette as soon as you *start* editing it, you can write recursive blocks (blocks that call themselves) by dragging the block into its own definition:



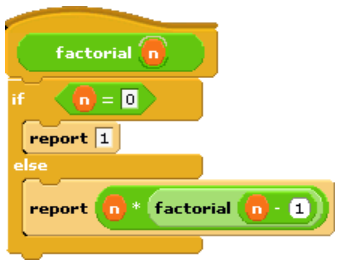




If recursion is new to you, here are a few brief hints: It's crucial that the recursion have a *base case,* that is, some small(est) case that the block can handle without using recursion. In this case, it's the case depth=0, for which the block does nothing at all, because of the enclosing **if**. Without a base case, the recursion would run forever, calling itself over and over.

Don't try to trace the exact sequence of steps that the computer follows in a recursive program. Instead, imagine that inside the computer there are many small people, and if Theresa is drawing a tree of size 100, depth 6, she hires Tom to make a tree of size 70, depth 5, and later hires Theo to make a tree of size 60, depth 5. Tom in turn hires Tammy and Tallulah, and so on. Each little person has his or her own local variables **size** and **depth**, each with different values.

You can also write recursive reporters, like this block to compute the factorial function:



Note the use of the **report** block. When a reporter block uses this block, the reporter finishes its work and reports the value given; any further blocks in the script are not evaluated. Thus, the **if else** block in the script above could have been just an **if**, with the second **report** block below it instead of inside it, and the result would be the same, because when the first **report** is seen in the base case, that finishes the block invocation, and the second **report** is ignored. There is also a **stop block** block that has a similar purpose, ending the block invocation early, for command blocks. (By contrast, the **stop script** block stops not only the current block invocation, but the entire toplevel script that called it.)

For more on recursion, see *Thinking Recursively* by Eric Roberts.

II. First Class Lists

A data type is "first class" in a programming language if data of that type can be

- the value of a variable
- an input to a procedure
- the value returned by a procedure
- a member of a data aggregate
- anonymous (not named)

In Scratch 1.4, numbers and text strings are first class. You can put a number in a variable, use one as the input to a block, write a reporter that reports a number, or put a number into a list.

But Scratch's lists are not first class. You create one using the "Make a list" button, which requires that you give the list a name. You can't put the list into a variable, into an input slot of a block, or into a list item—you can't have lists of lists. None of the Scratch reporters reports a list value. (You can use a reduction of the list into a text string as input to other blocks, but this loses the list structure; the input is just a text string, not a data aggregate.)

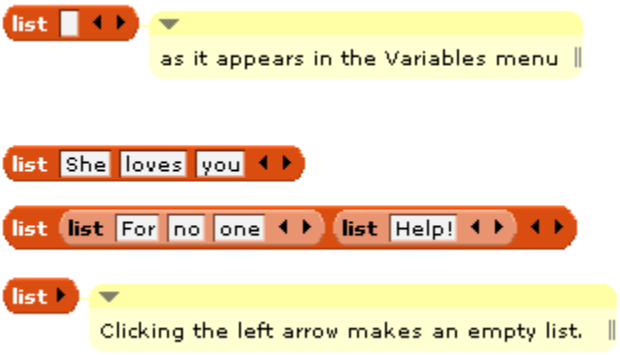
A fundamental design principle in BYOB is that ***all data should be first class***. If it's in the language, then we should be able to use it fully and freely. We believe that this principle avoids the need for many special-case tools, which can instead be written by BYOB users themselves.

Note that it's a data *type* that's first class, not an individual value. Don't think, for example, that lists made with the **list** block described below are first class, while lists made with the "Make a list" button aren't. In BYOB, lists are first class, period.

Everything first-class

A. The list block

At the heart of providing first-class lists is the ability to make an "anonymous" list—to make a list without simultaneously giving it a name. The **list** reporter block does that.



At the right end of the block are two left-and-right arrowheads. Clicking on these changes the number of inputs to **list**, i.e., the number of elements in the list you are building.

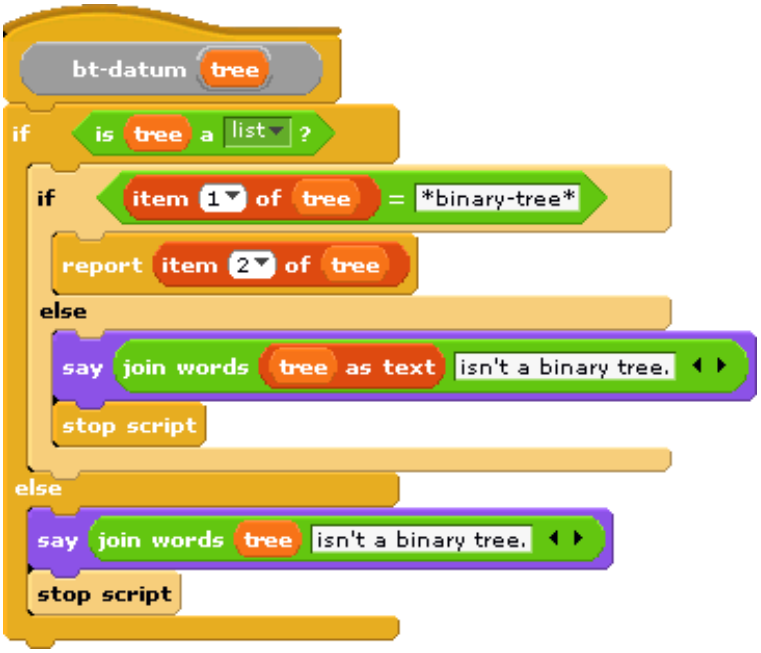
You can use this block as input to many other blocks:

Note that the Scratch list blocks use a dropdown menu with the names of all the "Make a list"-type lists. BYOB retains this notation for compatibility, but you can drag any expression whose value is a list over the dropdown menu:

B. Lists of lists

Lists can be inserted as elements in larger lists. We can easily create ad hoc structures as needed:

We can also build any classic computer science data structure out of lists of lists, by defining constructors, selectors, and mutators as needed. Here we create binary trees with type-checking selectors; only one selector is shown but the ones for left and right children are analogous.



III. Typed inputs

A. Scratch's type notation

Scratch block inputs come in two types: Text-or-number type and Number type. The former is indicated by a rectangular box, the latter by a rounded box:

A third Scratch type, Boolean (true/false), can be used in certain Control blocks with hexagonal slots.

The BYOB type system is an expanded version including Procedure, List, and Object types.

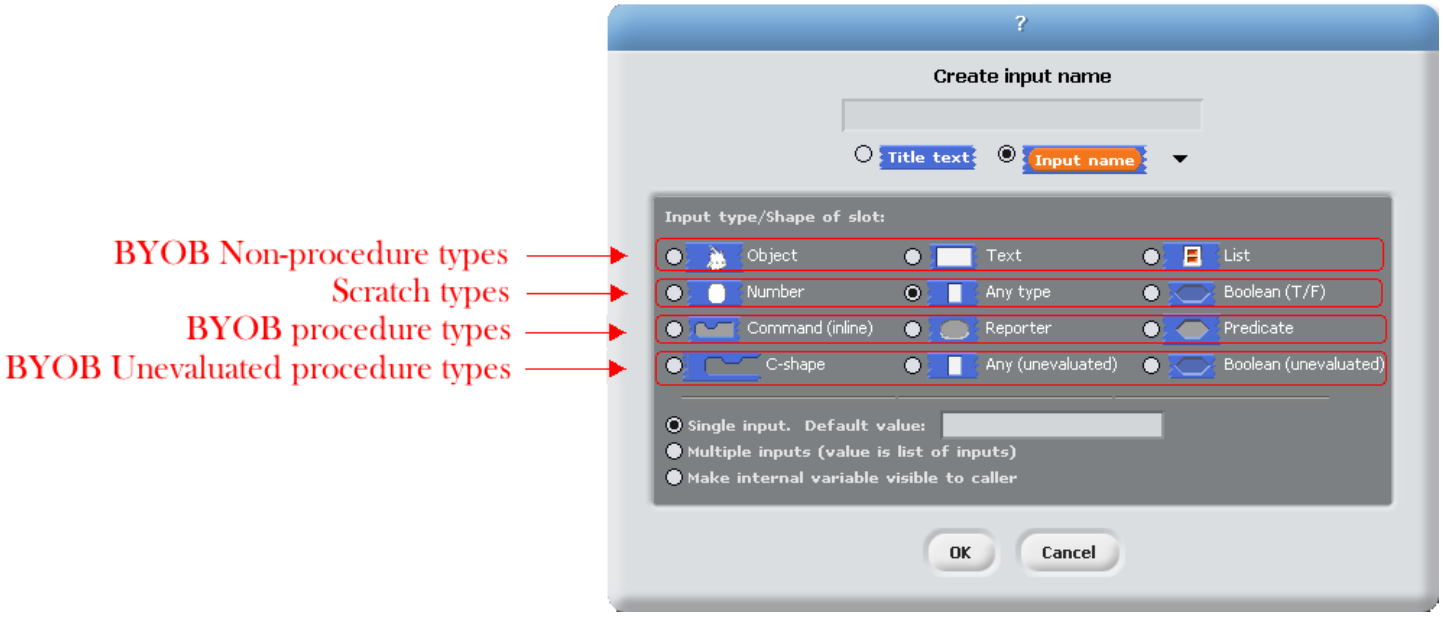
B. The BYOB Input Type dialog

In the input name dialog, there is a right-facing arrowhead after the "Input name" option:

Clicking that arrowhead opens the "long" input name dialog:

There are twelve input-type shapes, plus three mutually exclusive categories, listed in addition to the basic choice between title text and an input name. The default type, the one you get if you don't choose anything else, is "Any," meaning that this input slot is meant to accept any value of any type.

The arrangement of the input types is systematic. As the pictures on the next page show, each row of types is a category, and parts of each column form a category. Understanding the arrangement will make it a little easier to find the type you want.

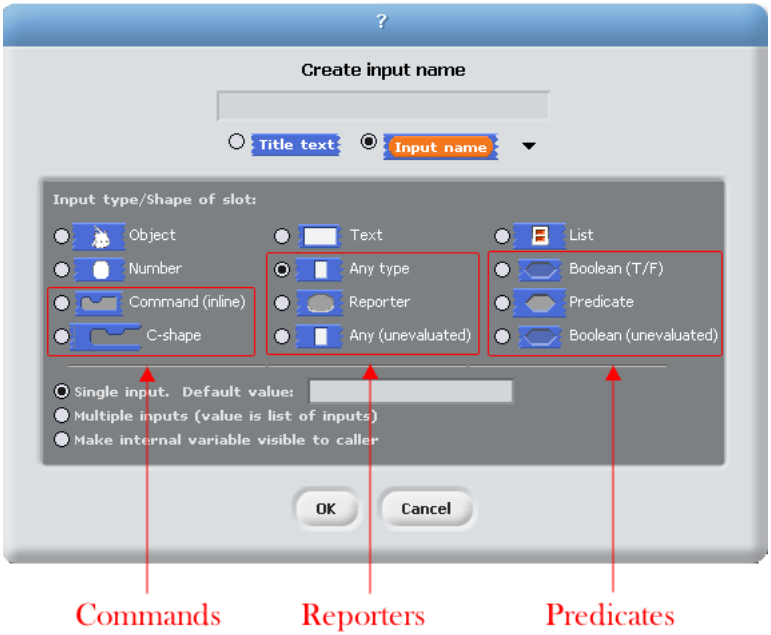


The second row of input types contains the ones found in Scratch: Number, Any, and Boolean. (The reason these are in the second row rather than the first will become clear when we look at the column arrangement.) The first row contains the new BYOB types other than procedures: Object, Text, and List. The last two rows are the types related to procedures, discussed more fully in section IV below.

The List type is used for first class lists, discussed in section II above. The red rectangles inside the input slot are meant to resemble the appearance of lists as Scratch displays them on the stage: each element in a red rectangle.

The Object type is for first class sprites, discussed in section V below.

The Text type is really just a variant form of the Any type, using a shape that suggests a text input. In Scratch, every block that takes a Text-type input has a default value that makes the rectangles for text wider than tall. The blocks that aren't specifically about text either are of Number type or have no default value, so those rectangles are taller than wide. At first we thought that Text was a separate type that always had a wide input slot; it turns out that this isn't true in Scratch (delete the default text and the rectangle narrows), but we thought it a good idea anyway, so we allow Text-shaped boxes even for empty input slots. (This is why Text comes above Any in the input type selection box.)



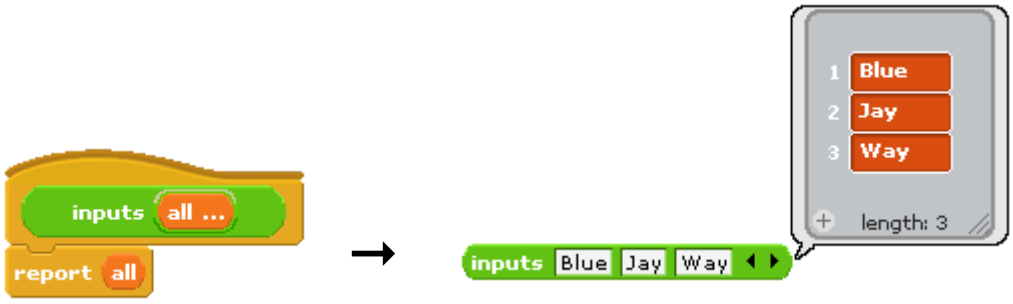
Although the procedure types are discussed more fully later, they are the key to understanding the column arrangement in the input types. Scratch has three block shapes: jigsaw-piece for command blocks, oval for reporters, and hexagonal for predicates. (A *predicate* is a reporter that always reports True or False.) In BYOB these blocks are first class data; an input to a block can be of Command type, Reporter type, or Predicate type. Each of these types is directly below the type of value that that kind of block reports, except for Commands, which don't report a value at all. Thus, oval Reporters are related to the Any type, while hexagonal Predicates are related to the Boolean (true or false) type.

The unevaluated procedure types in the fourth row are explained in section IV.D below. In one handwavy sentence, they combine the *meaning* of the procedure types with the *appearance* of the reported value types two rows higher. (Of course this isn't quite right for the C-shaped command input type, since commands don't report values. But you'll see later that it's true in spirit.)

We now turn to the three mutually exclusive options that come below the type array.

The "Single input" option: In Scratch all inputs are in this category. There is one input slot in the block as it appears in its palette. If a single input is of type Any, Number, or Text, then you can specify a default value that will be shown in that slot in the palette, like the "10" in the **move (10) steps** block. In the prototype block at the top of the script in the Block editor, an input with name "size" and default value 10 looks like this:

The "Multiple inputs" option: The **list** block introduced earlier accepts any number of inputs to specify the items of the new list. To allow this, BYOB introduces the arrowhead notation (◂ and ▸) that expands and contracts the block, adding and removing input slots. Custom blocks made by the BYOB user have that capability, too. If you choose the "Multiple inputs" button, then arrowheads will appear after the input slot in the block. More or fewer slots (as few as zero) may be used. When the block runs, all of the values in all of the slots for this input name are collected into a list, and the value of the input as seen inside the script is that list of values:



The ellipsis (...) in the orange input slot name box in the prototype indicates a Multiple input.

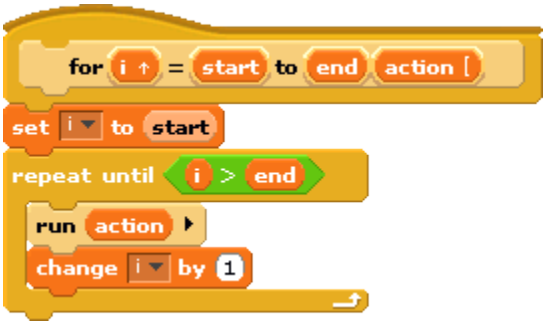
The third category, "Make internal variable visible," isn't really an input at all, but rather a sort of output from the block to its user. An upward-pointing arrow indicates this kind of input name in the prototype:

The variable **i** can be dragged from the **for** block into the blocks used in its C-shaped command slot. Also, by clicking on the orange **i**, the user can change the name of the variable as seen in the script (although the name hasn't changed inside the block's definition). This kind of variable is called an "upvar" for short, because it is passed *upward* from the custom block to the script that uses it.

IV. Procedures as Data

A. Procedure input types

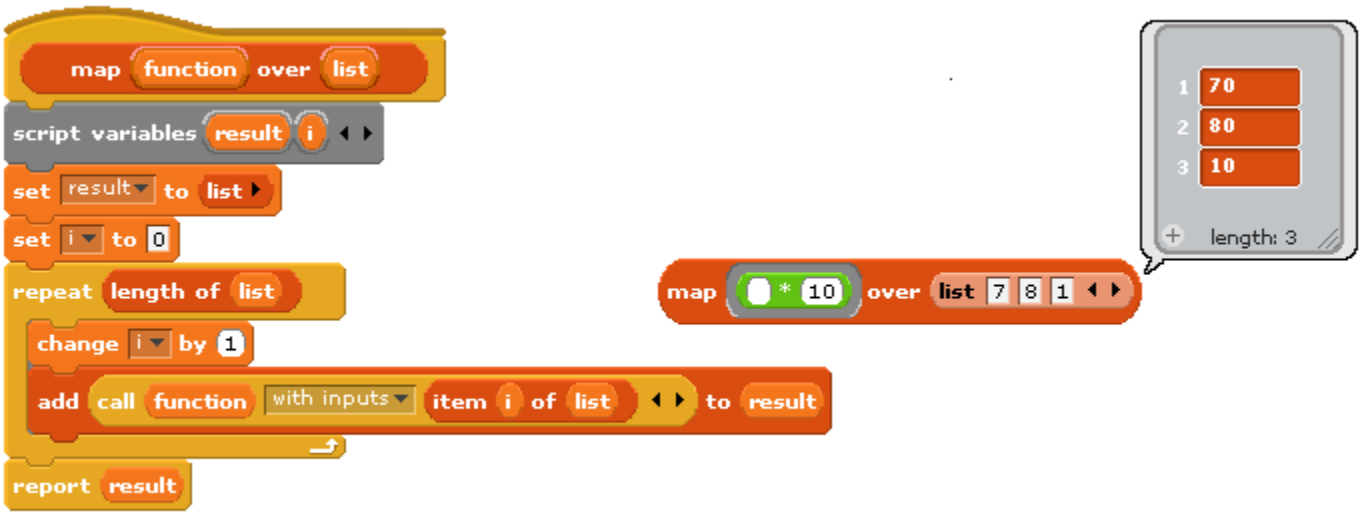
In the **for** block example above, the input named **action** has been declared as type "Command (C-shaped)"; that's why the finished block is C-shaped. But how does the block actually tell BYOB to carry out the commands inside the C-slot? Here is a simple version of the block script:



This is simplified because it assumes, without checking, that the ending value is greater than the starting value; if not, the block should (depending on the designer's purposes) either not run at all, or change the variable by −1 instead of by 1.

The important part of this script is the **run** block near the end. This is a BYOB built-in block that takes a Command-type value (a script) as its input, and carries out its instructions. There is a similar **call** block for invoking a Reporter or Predicate block. The **call** and **run** blocks are at the heart of BYOB's first class procedure feature; they allow scripts and blocks to be used as data — in this example, as an input to a block — and eventually carried out under control of the user's program.

Here's another example, this time using a Reporter-type input:

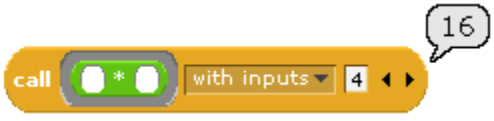


Here we are calling the Reporter "multiply by 10" three times, once with each item of the given list as its input, and collecting the results as a list. (The reported list will always be the same length as the input list.) Note that the multiplication block has two inputs, but here we have specified a particular value for one of them (10), so the **call** block knows to use the input value given to it just to fill the other (empty) input slot in the multiplication block.

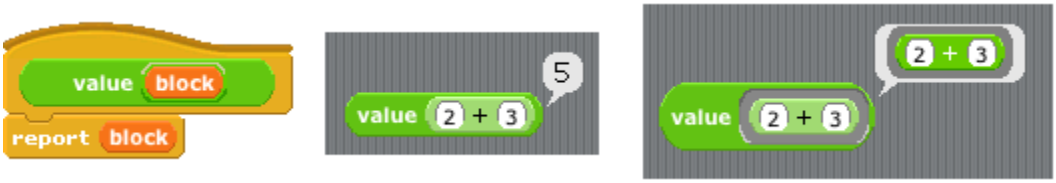
The **call** block (and also the **run** block) has a right arrowhead at the end; clicking on it adds the phrase "with inputs" and then a slot into which an input can be inserted:

If the left arrowhead is used to remove the last input slot, the "with inputs" disappears also. The right arrowhead can be clicked as many times as needed for the number of inputs required by the reporter block being called. (You'll have noticed that "with inputs" is presented in a way that suggests there's an alternative, but we'll come back to that later.)

If the number of inputs given to **call** (not counting the Reporter-type input that comes first) is the same as the number of empty input slots, then the empty slots are filled from left to right with the given input values. If **call** is given exactly one input, then *every* empty input slot of the called block is filled with the same value:

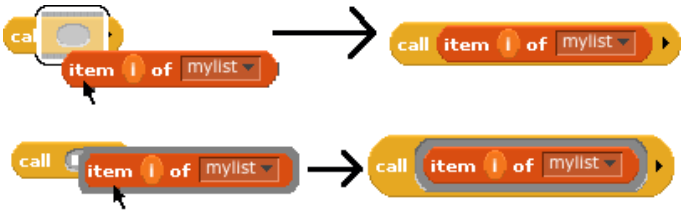


An even more important thing to notice about these examples is the thick grey border around the blocks that we've put in the Reporter-type input slots to **call** and **map** above. This notation, not seen in Scratch, indicates that *the block itself,* not the number or other value that the block would report when called, is the input.



By contrast, inside the **for** and **map** scripts, we actually want to evaluate a block: the orange one that reports the procedure's input value:

How do we distinguish these two cases as we're building scripts? There's a long way and a short way. The long way, to be described below, uses the **the block** and **the script** blocks. The short way uses the *distance* from the input slot to control whether a Procedure-type input slot's actual input block is evaluated, or is taken as the input without evaluation. As you drag a block toward a Procedure-type input slot, you see first the usual Scratch white halo around the input slot, then as you get closer, a white horizontal line inside the input slot and a grey halo around the dragged block.

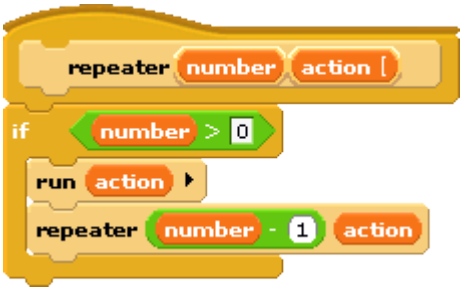


B. Writing Higher Order Procedures

A *higher order procedure* is one that takes another procedure as an input, or that reports a procedure. In this document, the word "procedure" encompasses scripts, individual blocks, and nested reporters. (Unless specified otherwise, "reporter" includes predicates. When the word is capitalized inside a sentence, it means specifically oval-shaped blocks. So, "nested reporters" includes predicates, but "a Reporter-type input" doesn't.)

Although an Any-type input slot (what you get if you use the small input-name dialog box) will accept a procedure input, it won't do the automatic grey-border input technique described above. So the declaration of Procedure-type inputs makes the use of your custom block much more convenient.

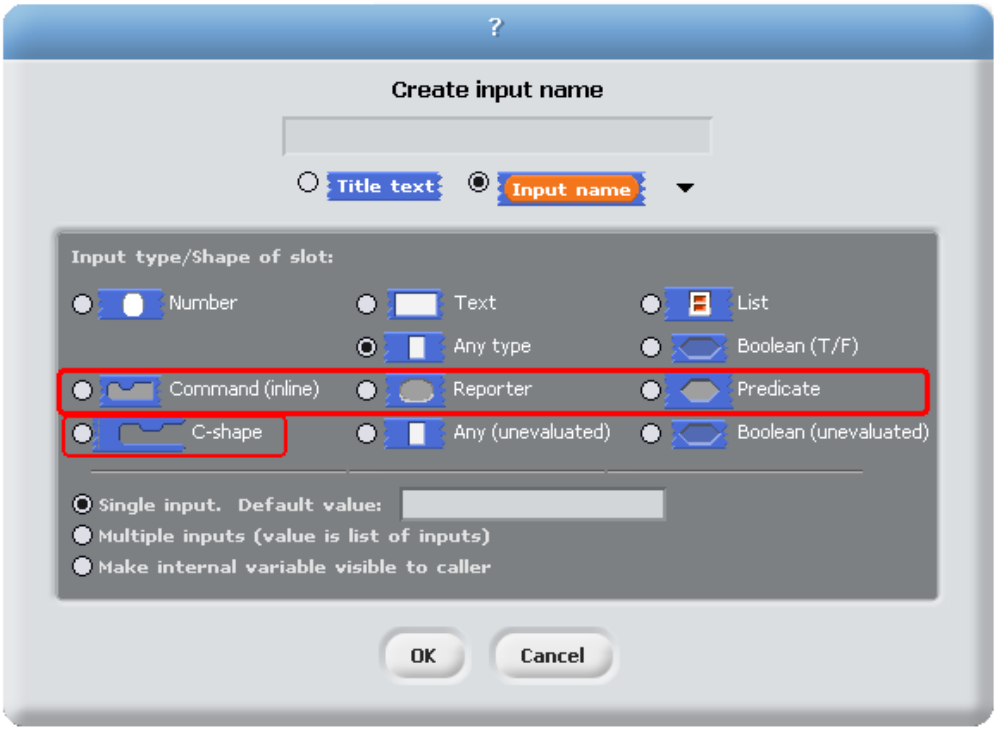
Why would you want a block to take a procedure as input? This is actually not an obscure thing to do; the Scratch conditional and looping blocks (the C-shaped ones in the Control palette) are taking a script as input. Scratch users just don't usually think about it in those terms! We could write the **repeat** block as a custom block this way, if Scratch didn't already have one:



The bracket **[** next to **action** in the prototype indicates that this is a C-shaped block, and that the script enclosed by the C is the input named **action** in the body of the script. The only way to make sense of the variable **action** is to understand that its value is a script.

To declare an input to be Procedure-type, open the input name dialog as usual, click on the arrow:

Then, in the long dialog, choose the appropriate Procedure type. The third row of input types has a grey slot in the shape of each block type (jigsaw for Commands, oval for Reporters, and hexagonal for Predicates). In practice, though, in the case of Commands it's more common to choose the C-shaped slot on the fourth row, because this "container" for command scripts is familiar to Scratch users. Technically the C-shaped slot is an *unevaluated* procedure type, something discussed in section D below. The two Command-related input types are connected by the fact that if a variable, an **item (#) of [list]** block, or a custom Reporter block is dropped onto a C-shaped slot, it turns into an inline slot, as in the **repeater** block's recursive call above. (Other built-in Reporters can't report scripts, so they aren't accepted in a C-shaped slot.)



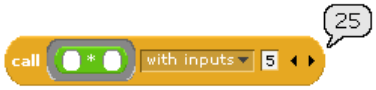
Why would you ever choose an inline Command slot rather than a C shape? Other than the **run** block discussed below, the only case I can think of is something like the C/C++/Java **for** loop, which actually has three command script inputs, only one of which is the "featured" loop body:

Okay, now that we have procedures as inputs to our blocks, how do we use them? We use the blocks **run** (for commands) and **call** (for reporters). The **run** block's script input is inline, not C-shaped, because we anticipate that it will be rare to use a specific, literal script as the input. Instead, the input will generally be a variable whose *value* is a script.

The **run** and **call** blocks have arrowheads at the end that can be used to open slots for inputs to the called procedures. How does BYOB know where to use those inputs? If the called procedure (block or script) has empty input slots, BYOB "does the right thing." This has several possible meanings:

1. If the number of empty slots is exactly equal to the number of inputs provided, then BYOB fills the empty slots from left to right:

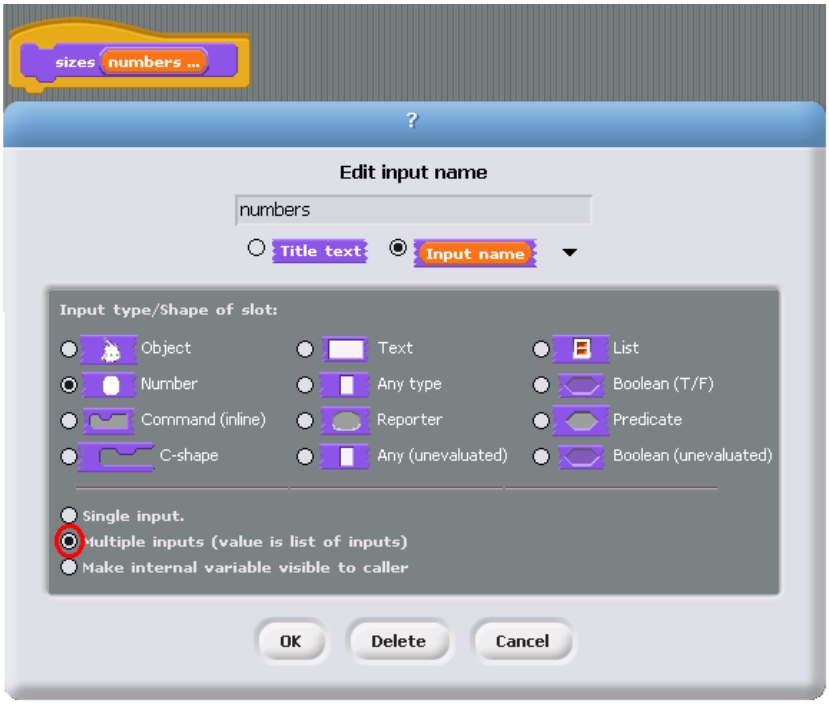
2. If exactly one input is provided, BYOB will fill any number of empty slots with it:



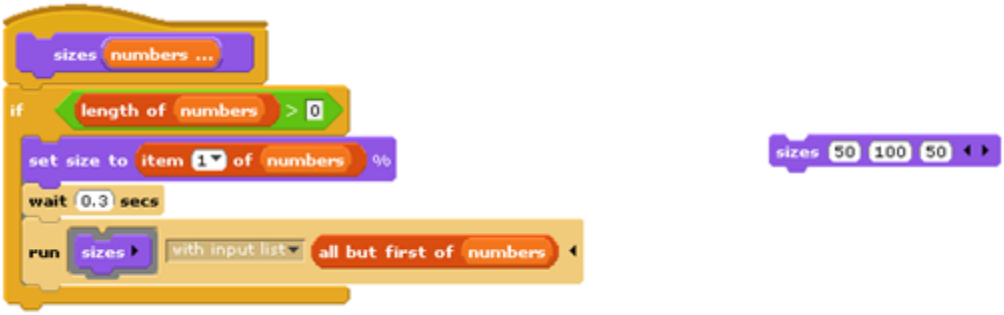
3. Otherwise, BYOB won't fill any slots, because the user's intention is unclear.

If the user wants to override these rules, the solution is to use **the block** or **the script** with explicit input names that can be put into the given block or script to indicate how inputs are to be used. This will be discussed more fully below.

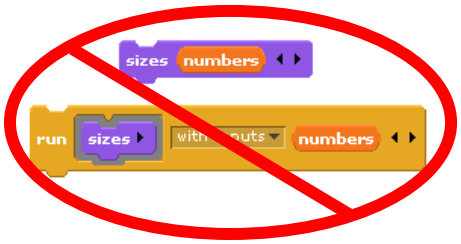
The text **with inputs** is in a dropdown menu that has one other choice: **with input list**. This variant is used mainly when making a recursive call to a block that takes a variable number of inputs:



This block will take any number of numbers as inputs, and will make the sprite grow and shrink accordingly:



The user of this block calls it with any number of *individual numbers* as inputs. But inside the definition of the block, all of those numbers form *a list* that has a single input name, **numbers**. This recursive definition first checks to make sure there are any inputs at all. If so, it processes the first input (item 1 of the list), then it wants to make a recursive call with all but the first number. (**All but first of** isn't built into BYOB, but is in the tools project distributed with BYOB.) But **sizes** doesn't take a list as input; it takes numbers as inputs! So these would be wrong:

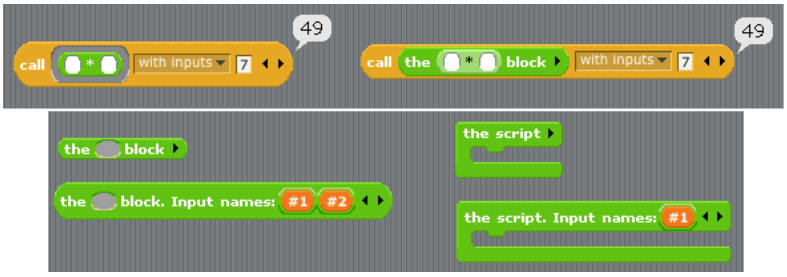


The **with input list** option replaces the multiple Any-type input slots in the **run** block with one List-type slot:

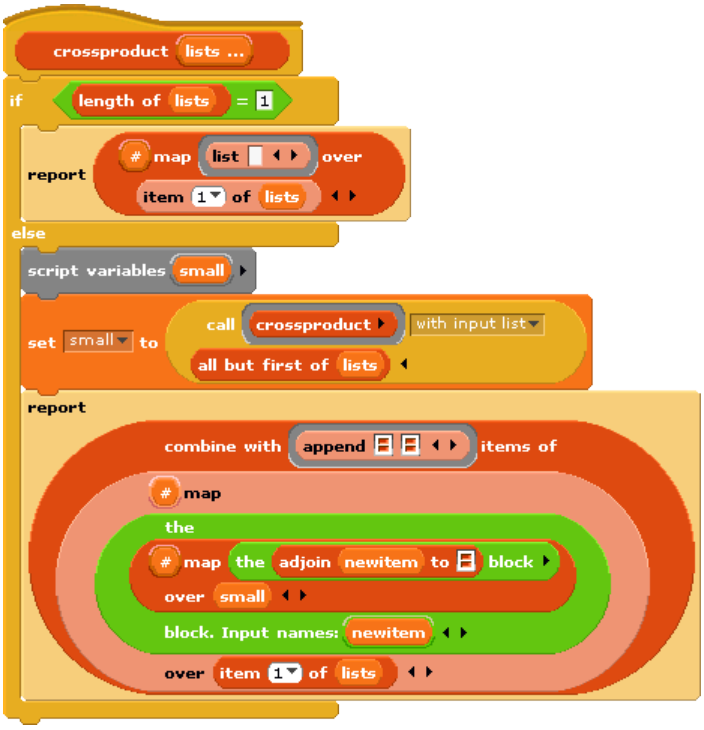
The items in the list are taken *individually* as inputs to the script. Since **numbers** is a list of numbers, each individual item is a number, just what **sizes** wants.

C. Procedures as Data

The grey borders we've been using around Procedure-type inputs up to now are actually an abbreviation for the blocks **the block** and **the script** found in the Operators palette. The arrowheads in these blocks can be used to give the inputs to a block or script explicit names, instead of using empty input slots as we've done until now.



There are two reasons why you might need **the block** or **the script**: Either you need more control over the use of inputs to an *encapsulated* procedure (one being used as data), or you want to use a procedure as the value for an input slot that is not declared to be of Procedure type (because other inputs are also meaningful in that context). Those both sound abstract and confusing, so here are some examples. First, the need for finer control over the use of input data:



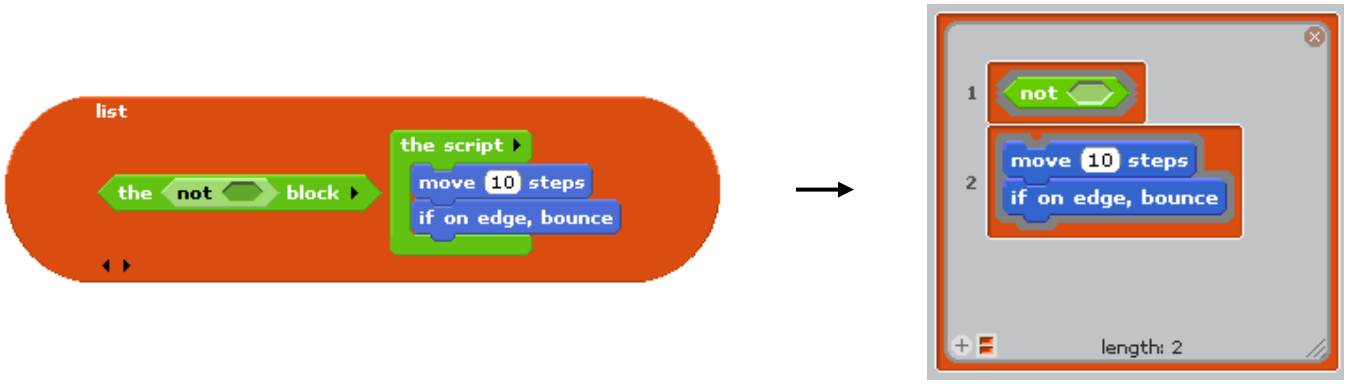
This is the definition of a block that takes any number of lists, and reports the list of all possible combinations of one item from each list. The important part for this discussion is that near the bottom there are two nested calls to **map**, a higher order function that applies an input function to each item of an input list. In the inner block, the function being mapped is **adjoin**, and that block takes two inputs. The second, the empty List-type slot, will get its value in each call from an item of the inner **map**'s list input. But there is no way for the outer **map** to communicate values to empty slots of the **adjoin** block. We must give an explicit name, **newitem**, to the value that the outer **map** is giving to the inner one, then drag that variable into the **adjoin** block.

If that example is too confusing, here is a simpler but more contrived one:

Here we just want to put one of the inputs into two different slots. If we left all three slots empty, BYOB would not fill any of them, because the number of inputs provided (2) would not match the number of empty slots.

By the way, once the called block provides names for its inputs, BYOB will not automatically fill empty slots, on the theory that the user has taken control. In fact, that's another reason you might want to name the inputs explicitly: to stop BYOB from filling a slot that should really remain empty.

Here's an example of the other situation in which a procedure must be explicitly marked as data:



Here, we are making a list of procedures. But the **list** block accepts inputs of any type, so its input slots are not marked as Procedure type, so there is no option of dropping the block or script with an automatic grey border. We must say explicitly that we want the block itself as the input, rather than whatever value would result from *evaluating* the block.

Notice, by the way, that BYOB *displays* blocks and scripts in grey borders even if you used **the block** or **the script** to create them.

The block and **the script** mark their input as data; the former takes (reporter) blocks and the latter takes scripts. Both have right arrows that can be used to supply explicit names for expected inputs. Clicking the arrow exposes an orange oval containing the default name **#1** for the first such input, **#2** for the second, and so on. These default names can be changed to something more meaningful by clicking on the orange oval. (Don't drag by accident.) These variables can be dragged into the block or script being encapsulated. The names of the input variables are called the *formal parameters* of the encapsulated procedure.

Besides the **list** block in the example above, other blocks into which you may want to put procedures are **set** (the value of a variable to a procedure), **say** and **think** (to display a procedure to the user), and **report** (for a reporter that reports a procedure):

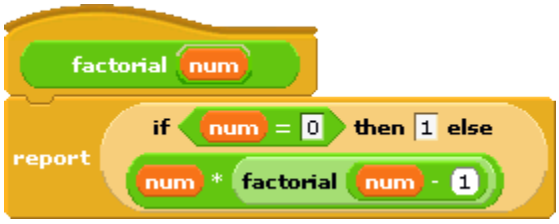
When you are using **the block** to control the encapsulated block's use of its inputs with variable names, be careful not to put a grey border around **the block** itself, which would encapsulate **the block** instead of whatever block you used as input to it:

Although unusual, it is possible that you'd want to make a list of blocks that includes **the block**. That's why the automatic grey border isn't ruled out.

D. Special Forms

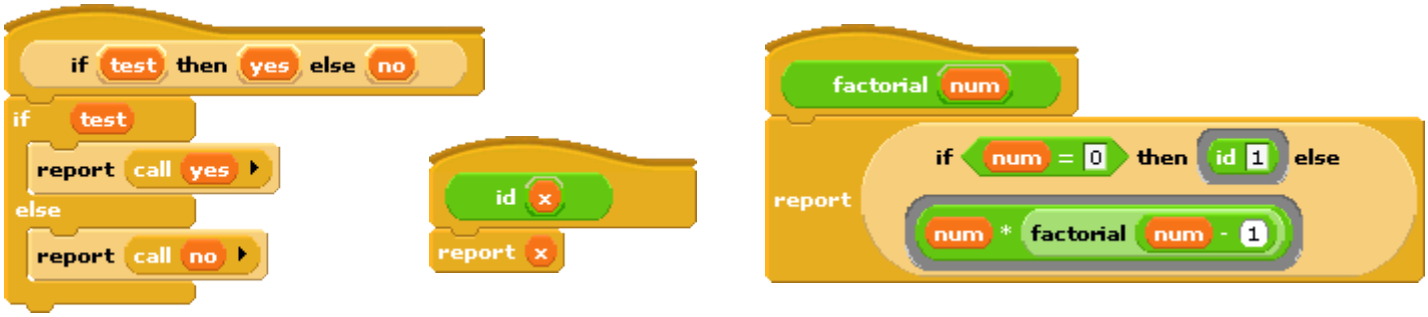
Scratch has an **if else** block that has two C-shaped command slots and chooses one or the other depending on a Boolean test. Because Scratch doesn't emphasize functional programming, it lacks a corresponding reporter block to choose between two expressions. We could write one:

Our block works for these simple examples, but if we try to use it in writing a recursive operator, it'll fail:



The problem is that when any Scratch block is called, all of its inputs are computed (evaluated) before the block itself runs. The block itself only knows the values of its inputs, not what expressions were used to compute them. In particular, all of the inputs to our **if then else** block are evaluated first thing. That means that even in the base case, **factorial** will try to call itself recursively, causing an infinite loop. We need our **if then else** block to be able to select only one of the two alternatives to be evaluated.

We have a mechanism to allow that: declare the **then** and **else** inputs to be of type Reporter rather than type Any. Then, when calling the block, enclose those inputs in **the block** (or use the equivalent grey border notation) so that the expressions themselves, rather than their values, become the inputs:



In this version, the program works, with no infinite loop. But we've paid a heavy price: this reporter-**if** is no longer as intuitively obvious as the Scratch command-**if**. You have to know about procedures as data, about grey borders, and about constant functions. (The **id** block implements the identity function, which reports its input. We need it because **the block** only takes reporters as input, not numbers.) What we'd like is a reporter-**if** that *behaves* like this one, delaying the evaluation of its inputs, but *looks* like our first version, which was easy to use except that it didn't work.

Such blocks are indeed possible. A block that seems to take a simple expression as input, but delays the evaluation of that input by wrapping a hidden **the block** around it (and, if necessary, an **id**-like transformation of constant data into constant functions) is called a *special form*. To turn our **if** block into a special form, we edit the block's prototype, declaring the inputs **yes** and **no** to be of type "Any (unevaluated)" instead of type Reporter. The script for the block is still that of the second version, including the use of **call** to evaluate either **yes** or **no** but not both. But the slots appear as white Any-type rectangles, not Reporter-type ovals, and the **factorial** block will look like our first attempt.

In a special form's prototype, the unevaluated input slot(s) are indicated by an asterisk next to the input name:

Special forms trade off implementor sophistication for user sophistication. That is, you have to understand all about procedures as data to make sense of the special form implementation of **if then else**. But any Scratch programmer can *use* **if then else** without thinking at all about how it works internally.

Special forms are actually not a new invention in BYOB. Many of Scratch's conditional and looping blocks are really special forms. The hexagonal input slot in the **if** block is a straightforward Boolean value, because the value can be computed once, before the **if** block makes its decision about whether or not to run its action input. But the **forever if**, **repeat until**, and **wait until** blocks' inputs can't be Booleans; they have to be of type "Boolean (unevaluated)," so that Scratch can evaluate them over and over again. Since Scratch doesn't have custom blocks, it can afford to handwave away the distinction between evaluated and unevaluated Booleans, but BYOB can't. The pedagogic value of special forms is proven by the fact that no Scratcher ever notices that there's anything strange about the way in which the hexagonal inputs in the Control blocks are evaluated.

Also, the C-shaped slot familiar to Scratch users is an unevaluated procedure type; you don't have to use **the script** to keep the commands in the C-slot from being run before the C-shaped block is run. Those commands themselves, not the result of running them, are the input to the C-shaped Control block. (This is taken for granted by Scratch users, especially because Commands don't report values, so it wouldn't make sense to think of putting commands in the C-shaped slot as a composition of functions.) This is why it makes sense that "C-shaped" is on the fourth row of types in the long form input dialog.

V. Object Oriented Programming with Sprites

Object oriented programming is a style based around the abstraction *object:* a collection of data and *methods* (procedures, which from our point of view are just more data) that you interact with by sending it a *message* (just a name, maybe in the form of a text string, and perhaps additional inputs) to which it responds by carrying out a method, which may or may not report a value back to the asker. Reasons for using OOP vary; some people emphasize the *data hiding* aspect (because each object has local variables that other objects can access only by sending messages to the owning object) while others emphasize the *simulation* aspect (in which each object abstractly represents something in the world, and the interactions of objects in the program model real interactions of real people or things). Data hiding is important for large multi-programmer industrial projects, but for BYOB users it's the simulation aspect that's important. Our approach is therefore less restrictive than that of some other OOP languages.

Technically, object oriented programming rests on three legs: (1) *Message passing:* There is a notation by which any object can send a message to another object. (2) *Local state:* Each object can remember the important past history of the computation it has performed. ("Important" means that it need not remember every message it has handled, but only the lasting effects of those messages that will affect later computation.) (3) *Inheritance:* It would be impractical if each individual object had to contain methods, many of them redundant, for all of the messages it can accept. Instead, we need a way to say that this new object is just like that old object except for a few differences, so that only those differences need be programmed explicitly.

A. First Class Sprites

Scratch comes with things that are natural objects: its sprites. Each sprite can own local variables; each sprite has its own scripts (methods). A Scratch animation is plainly a simulation of the interaction of characters in a play. There are three ways in which Scratch sprites are less versatile than the objects of an OOP language. First, message passing is primitive in three respects: Messages can only be **broadcast**, not addressed to an individual sprite; messages can't take inputs; and methods can't return values to their caller. Second, there is no inheritance mechanism for sprites in Scratch. (You can *duplicate* a sprite, but the new sprite doesn't follow any change in the methods or state of the original.) Third, and most basic, in the OOP paradigm objects are *data;* they can be the value of a variable, an element of a list, and so on.

BYOB sprites are first class data. They can be created and deleted by a script, stored in a variable or list, sent messages individually, and inherit properties from another sprite. A sprite can be *cloned,* creating a new sprite that shares methods, variables, and lists with its parent. As we'll see in detail later, each property of the child can individually be shared with, or separated from, the corresponding property of the parent.

The fundamental means by which programs get access to sprites is the **object** reporter block. It has a dropdown-menu input slot that, when clicked, lists all the sprites, plus the stage (also an object, and the main reason why the block isn't named **sprite**), and special keywords "myself" and "all sprites"; the second of these reports a list of sprites rather than a single object. An object reported by **object** can be used as input to any block that accepts any input type, such as **set**. If you **say** an object, the resulting speech balloon will contain a smaller image of the object's costume or (for the stage) background.

B. Sending Messages to Sprites

The messages that a sprite accepts are the blocks in its palettes, including both all-sprites and this-sprite-only blocks. (For custom blocks, the corresponding methods are the scripts as seen in the Block Editor.

Scratch provides a way for one sprite to get at certain attributes of another: the **<var> of <sprite>** block in the Sensing palette. BYOB extends this block to allow access to *any* property of a sprite. The two input slots in that block are dropdown menus for Scratch compatibility, but any object-valued expression can be dragged onto the right one, and any block enclosed in **the block** or **the script** can be dragged onto the left one. The result can be used with **call** or **run** to use the corresponding method.

Notice that the **move** block is ordinarily global; when used in a script, it moves whichever sprite runs the script. But when combined with **of**, as in the picture above, the result is effectively a sprite-local block that always moves Sprite2 regardless of which sprite **run**s it

As a convenience, BYOB includes a **launch** block that's identical to **run** except that it calls the method as a separate script, so the calling script can meanwhile do something else. **Launch** can be thought of as an improved **broadcast**, while **run** of another sprite's method is more analogous to **broadcast and wait**.

Although the picture above shows a block with its input slot filled, messages with empty input slots can also be used, and can be given input values by **call** or **run** as usual.

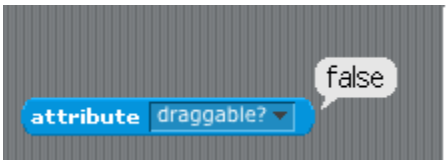
C. Local State in Sprites: Variables and Attributes

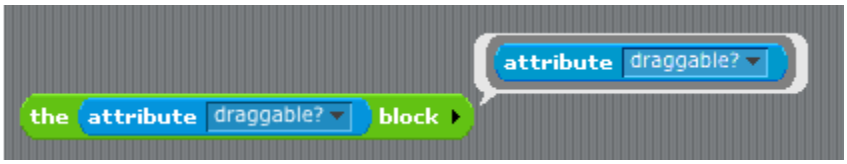
A sprite's memory of its own past history takes two main forms. It has *variables,* created explicitly by the user with the "Make a variable" button; it also has *attributes,* the qualities every sprite has automatically, such as position, direction, and pen color. Each variable can be examined using its own orange oval block; there is one **set** block to modify all variables. Attributes have a less uniform programming interface in Scratch. A sprite's direction can be examined with the **direction** block, and modified with the **point in direction <dir>** block. It can also be modified less directly using the blocks **turn**, **point towards**, and **if on edge, bounce**. There is no way for a Scratch script to examine a sprite's pen color, but there are blocks **set pen color to <color>**, **set pen color to <number>**, and **change pen color by <number>** to modify it. A sprite's name can be neither examined nor modified by scripts; it can be modified by typing a new name directly into the box that displays the name, above the scripting area. The block, if any, that examines a variable or attribute is called its *getter;* a block (there may be more than one, as in the examples above) that modifies a variable or attribute is called a *setter.*

In certain contexts, as explained shortly, it is useful to distinguish between the *value* of a variable or attribute and the variable or attribute as a thing in its own right. An example would be the distinction between "this sprite is currently pointing toward the right" and "a sprite always has a direction, which is an angle measured clockwise from north," respectively. ***In BYOB, a variable or attribute is represented by its getter block.*** As a result, we already have the means to express the distinction between the value and the thing-in-itself: for the latter, we put the

getter block inside a **the block** block. For example, the value reported by **direction** is a number, the sprite's current direction; but the value reported by **the (direction) block** is a block, the direction attribute itself.

Using **the block** to represent an attribute works only if there *is* a getter block for that attribute. We want to be able to represent every component of the state of a sprite. Instead of adding dozens of getter blocks to the palettes, BYOB introduces a single **attribute** block in the Sensing palette. Its dropdown-menu input has an entry for every sprite attribute. The value reported by **attribute (direction)** is the same number reported by the **direction** block in the Motion palette. The value reported by **the (attribute (direction)) block** is the direction attribute itself. Similarly, **attribute (draggable?)** reports true if the sprite can be dragged in presentation mode (that is, if the padlock icon near the top of the scripting area is unlocked); **the (attribute (draggable?)) block** reports the sprite's draggability attribute.





One benefit of BYOB's first class blocks is that we don't need an equivalent to **attribute** for setters. Instead, we extend the **set <variable> to <value>** block, allowing an attribute (with **the block**) to be dragged onto the dropdown list of variables: **set (the (attribute (draggable?)) block) to <true>**. Note that this works only for variable and attribute getters, not for reporters in general! You can't say **set (the (sqrt of (x)) block) to 5** to set the variable *x* to 25.

D. Prototyping: Parents and Children

Most current OOP languages use a *class/instance* approach to creating objects. A class is a particular kind of object, and an instance is an actual object of that type. For example, there might be a Dog class, and several instances Fido, Spot, and Runt. The class typically specifies the methods shared by all dogs (RollOver, SitUpAndBeg, Fetch, and so on), and the instances contain data such as species, color, and friendliness. BYOB uses a different approach called *prototyping,* in which there is no distinction between classes and instances. Prototyping is better suited to an experimental, tinkering style of work: You make a single dog sprite, with both methods (blocks) and data (variables and lists), and when you like it, you use it as the prototype from which to clone other dogs. If you later discover a bug in the behavior of dogs, you can edit a method in the parent, and all of the children will automatically share the new version of the method block. Experienced class/instance programmers may find prototyping strange at first, but it is actually a more expressive system, because you can easily simulate a class by hiding the prototype sprite! Prototyping is also a better fit with the Scratch design principle that everything in a project should be concrete and visible on the stage; in class/instance OOP the programming process begins with an abstract, invisible entity, the class, that must be designed before any concrete objects can be made.

There are three ways to make a child sprite. First, if you control-click or right-click on a sprite in the "sprite corral" at the bottom right corner of the window, you get a menu that includes "clone" as one of the choices. Second, there is a **clone** block in the Operators palette that creates and reports a child sprite. Third, sprites have a "parent" attribute that can be set, like any attribute, thereby *changing* the parent of an existing sprite.

In projects that create clones, the total number of sprites may be very large, and several of them may look identical, making it hard to discover which of them is the prototype for the others. Control-clicking or right-clicking on the stage icon in the sprite corral displays a menu containing icons and names of only those sprites without parents. Clicking on a sprite selects it and, if it has children, displays a menu of those children. (Some of those may themselves have children, which can be displayed by the same technique.)

E. Inheritance by Delegation

A clone *inherits* properties of its parent. "Properties" include scripts, custom blocks, variables, named lists, system attributes, costumes, and sounds. Each individual property can be shared between parent and child, or not shared (with a separate one in the child). The getter block for a shared property, in the child's palette, is displayed in a lighter color; separate properties of the child are displayed in the traditional colors.

When a new clone is created, by default it shares all its properties with its parent *except for* system attributes. If the value of a shared property is changed in the parent, then the children see the new value. If the value of a shared property is changed in the *child*, then the sharing link is broken, and a new private version is created in that child. (This is the mechanism by which a child chooses not to share a property with its parent.) "Changed" in this context means using the **set** or **change** block for a variable, editing a block in the Block Editor, editing a costume or sound, or inserting, deleting, or reordering costumes or sounds. To change a property from unshared to shared, the child deletes its private version, using the **delete** command block. This block has an input slot that will accept a sprite or any property of a sprite (as represented by its getter block). Its input slot is of type Reporter, so you can use the grey-border notation for a block input. This can be used instead of the "Delete a variable" button in Scratch. Note however that sprites themselves aren't properties, so don't grey-border a sprite.

When a sprite gets a message for which it doesn't have a corresponding block, the message is *delegated* to that sprite's parent. When a sprite does have the corresponding block, then the message is not delegated. If the script that implements a delegated message refers to **object (myself)**, it means the child to which the message was originally sent, not the parent to which the message was delegated.

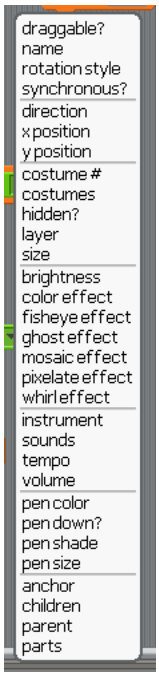
F. Nesting Sprites: Anchors and Parts

Sometimes it's desirable to make a sort of "super-sprite" composed of pieces that can move together but can also be separately articulated. The classic example is a person's body made up of a torso, limbs, and a head. BYOB allows one sprite to be designated as the *anchor* of the combined shape, with other sprites as its *parts.* This is another form of connection among sprites, separate from the parent/child connection. Sprite nesting can be set up both interactively and from a script. To do it interactively, drag the sprite corral icon of a *part* sprite onto the stage display (not the sprite corral icon!) of the desired *anchor* sprite. In a script, a sprite's "anchor" attribute can be set like any other attribute.

Sprite nesting is shown in the sprite corral icons of both anchors and parts:

In this illustration, it is desired to animate Alonzo's arm. (The arm has been colored green in this picture to make the relationship of the two sprites clearer, but in a real project they'd be the same color, probably.) Sprite1, representing Alonzo's body, is the anchor; Sprite2 is the arm. The icon for the anchor shows small images of up to three attached parts at the bottom. The icon for each part shows a small image of the anchor in its top left corner, and a *synchronous rotation flag* in the top right corner. In its initial setting, as shown above, it means that the when the anchor sprite rotates, the part sprite also rotates as well as revolving around the anchor. When clicked, it changes from a circle to a dot, and indicates that when the anchor sprite rotates, the part sprite revolves around it, but does not rotate, keeping its original orientation. (The part can also be rotated separately, using its **turn** blocks.) Any change in the position or size of the anchor is always extended to its parts.

G. List of attributes



At the right is a picture of the dtopdown menu of attributes in the **attribute** block. Four of these are not real attributes, but rather *lists* of things related to attributes:

- **costumes**: a list of the *names* of the sprite's costumes
- **sounds**: a list of the names of the sprite's sounds
- **children**: a list of sprites whose **parent** attribute is this sprite
- **parts**: a list of sprites whose **anchor** attribute is this sprite

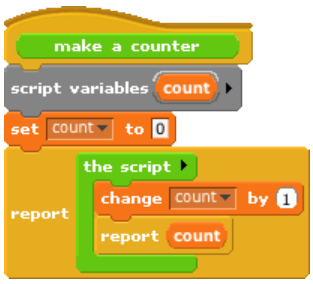
Most of the others should be clear because their names match Sprite getter or setter blocks, except for a few that Scratch controls only through the GUI: **draggable?** is true if the padlock is open; **rotation style** is a number from 1 to 3 corresponding to the column of three small buttons near the padlock; **synchronous?** is true for an anchored sprite if it rotates when it revolves.

VI. Building Objects Explicitly

The idea of object oriented programming is often taught in a way that makes it seem as if a special object oriented programming language is necessary. In fact, any language with first class procedures allows objects to be implemented explicitly; this is a useful exercise to help demystify objects.

The central idea of this implementation is that an object is represented as a *dispatch procedure* that takes a message as input and reports the corresponding method. In this section we start with a stripped-down example to show how local state can work, and build up to full implementations of both class/instance and prototyping OOP.

A. Local State with Script Variables



This script implements an object class, a type of object, namely the **counter** class. In this first simplified version there is only one method, so no explicit message-passing is necessary. When the **make a counter** block is called, it reports a procedure, created by the **the script** block inside its body. That procedure implements a specific counter object, an instance of the **counter** class. When invoked, a counter instance increases and reports its **count** variable. Each counter has its own local count:

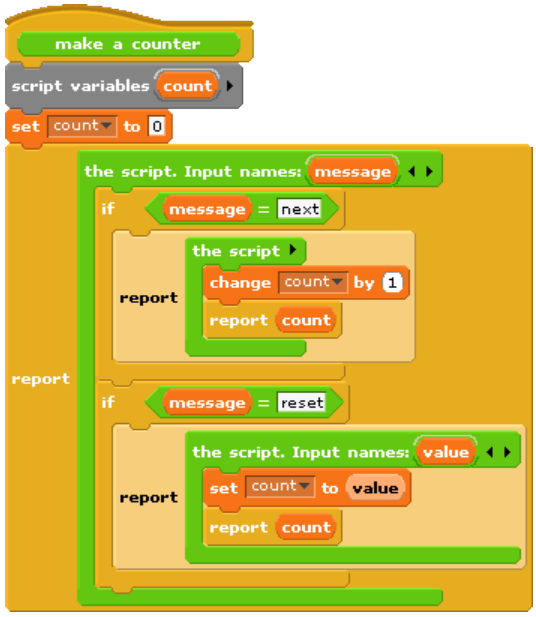
This example will repay careful study, because it isn't obvious why each instance has a separate count. From the point of view of the **make a counter** procedure, each invocation causes a new, block-associated **count** variable to be created. Usually such "block variables" are temporary, going out of existence when the procedure ends. But this one is special, because the procedure returns another procedure that makes reference to the **count** variable, so it remains active. (The **script variables** block makes variables local to a script. It can be used in a sprite's script area or in the Block Editor. Script variables can't be dragged outside the script in which they are created, but they can be "exported" indirectly by being used in a reported procedure, as here.)

In this approach to OOP we are representing both classes and instances as procedures. The **make a counter** block represents the class, while each instance is represented by a nameless script created each time **make a**

counter is called. The script variables created inside the **make a counter** block but outside the **the script** block are instance variables, belonging to a particular counter.

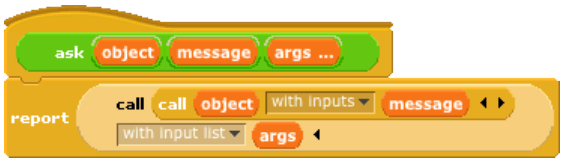
B. Messages and Dispatch Procedures

In the simplified class above, there is only one method, and so there are no messages; you just call the instance to carry out its one method. Here is a more detailed version that uses message passing:



Again, the **make a counter** block represents the **counter** class, and again the script creates a local variable **count** each time it is invoked. The large outer **the script** block represents an instance. It is a *dispatch procedure:* it takes a message (just a text word) as input, and it reports a method. The two smaller **the script** blocks are the methods. The top one is the **next** method; the bottom one is the **reset** method. The latter requires an input, named **value**.

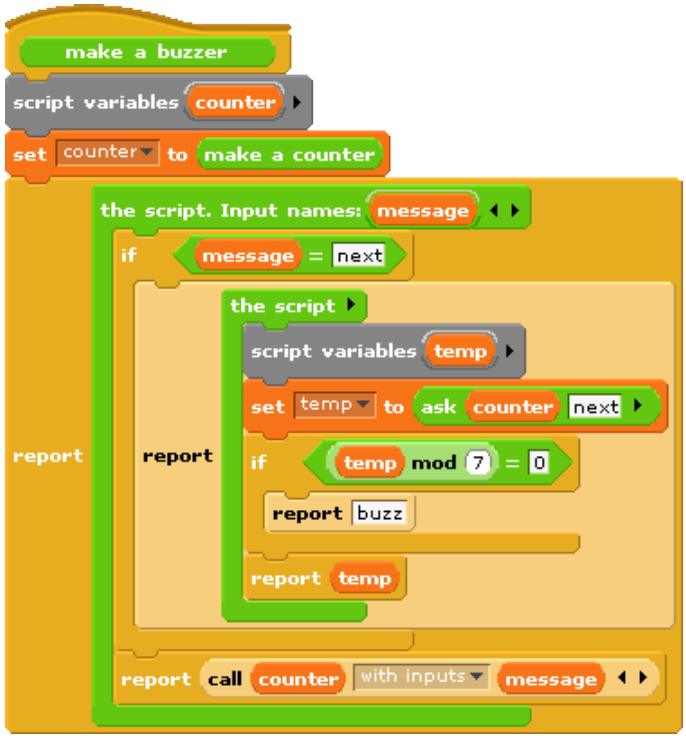
In the earlier version, calling the instance did the entire job. In this version, calling the instance gives access to a method, which must then be called to finish the job. We can provide a block to do both procedure calls in one:



The **ask** block has two required inputs: an object and a message. It also accepts optional additional inputs, which BYOB puts in a list; that list is named **args** inside the block. The block has two nested **call** blocks. The inner one calls the object, i.e., the dispatch procedure. The dispatch procedure always takes exactly one input, namely the message. It reports a method, which may take any number of inputs; note that this is the situation in which we need the "with input list" option of **call**.

C. Inheritance via Delegation

So, our objects now have local state variables and message passing. What about inheritance? We can provide that capability using the technique of delegation. Each instance of the child class contains an instance of the parent class, and simply passes on the messages it doesn't want to specialize:

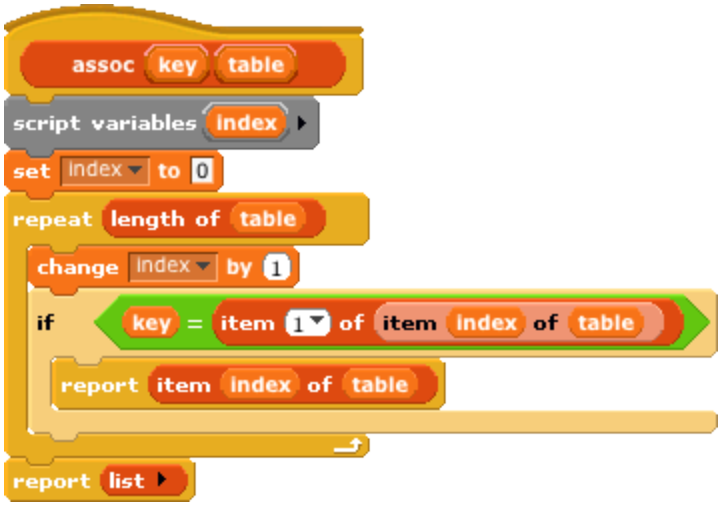


This script implements the **buzzer** class, which is a child of **counter**. Instead of having a count (a number) as a local state variable, each buzzer has a **counter** (an object) as a local state variable. The class specializes the **next** method, reporting what the counter reports unless that result is divisible by 7, in which case it reports "buzz." If the message is anything other than **next**, though, then the buzzer simply invokes its **counter**'s dispatch procedure. So the counter handles any message that the buzzer doesn't handle explicitly.

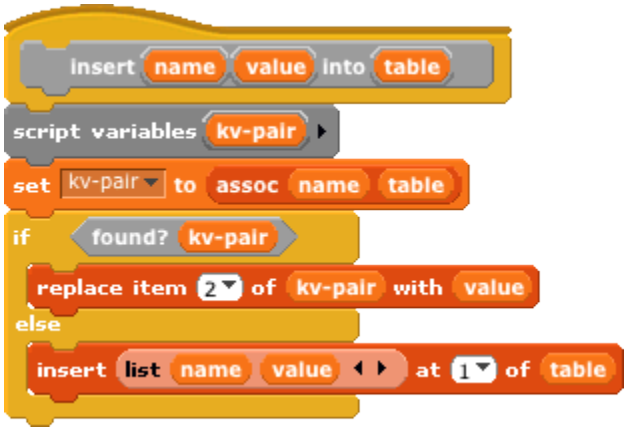
D. An Implementation of Prototyping OOP

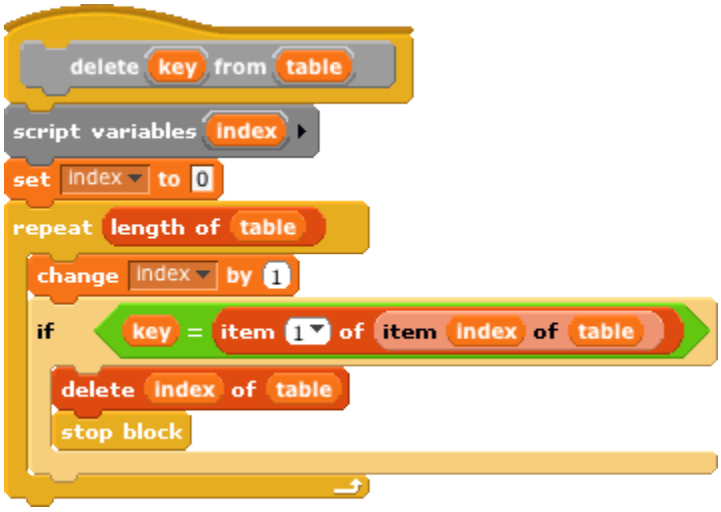
In the class/instance system above, it is necessary to design the complete behavior of a class before you can make any instances of the class. This is great for top-down design, but not great for experimentation. Here we sketch the implementation of a prototyping OOP system modeled after the behavior of BYOB sprites: children share properties of their parent unless and until a child changes a property, at which point the child gets a private copy. Since these explicitly constructed objects aren't sprites, they don't have system attributes; their properties consist of methods and local variables.

Because we want to be able to create and delete properties dynamically, we don't use BYOB variables to hold an object's variables or methods. Instead, each object has two *tables,* called **methods** and **data**, each of which is an *association list,* a list of two-item lists, in which each of the latter contains a *key* and a correspoinding *value.* We provide a lookup procedure to locate the key-value pair corresponding to a given key in a given table.



There are also commands to insert and delete entries:



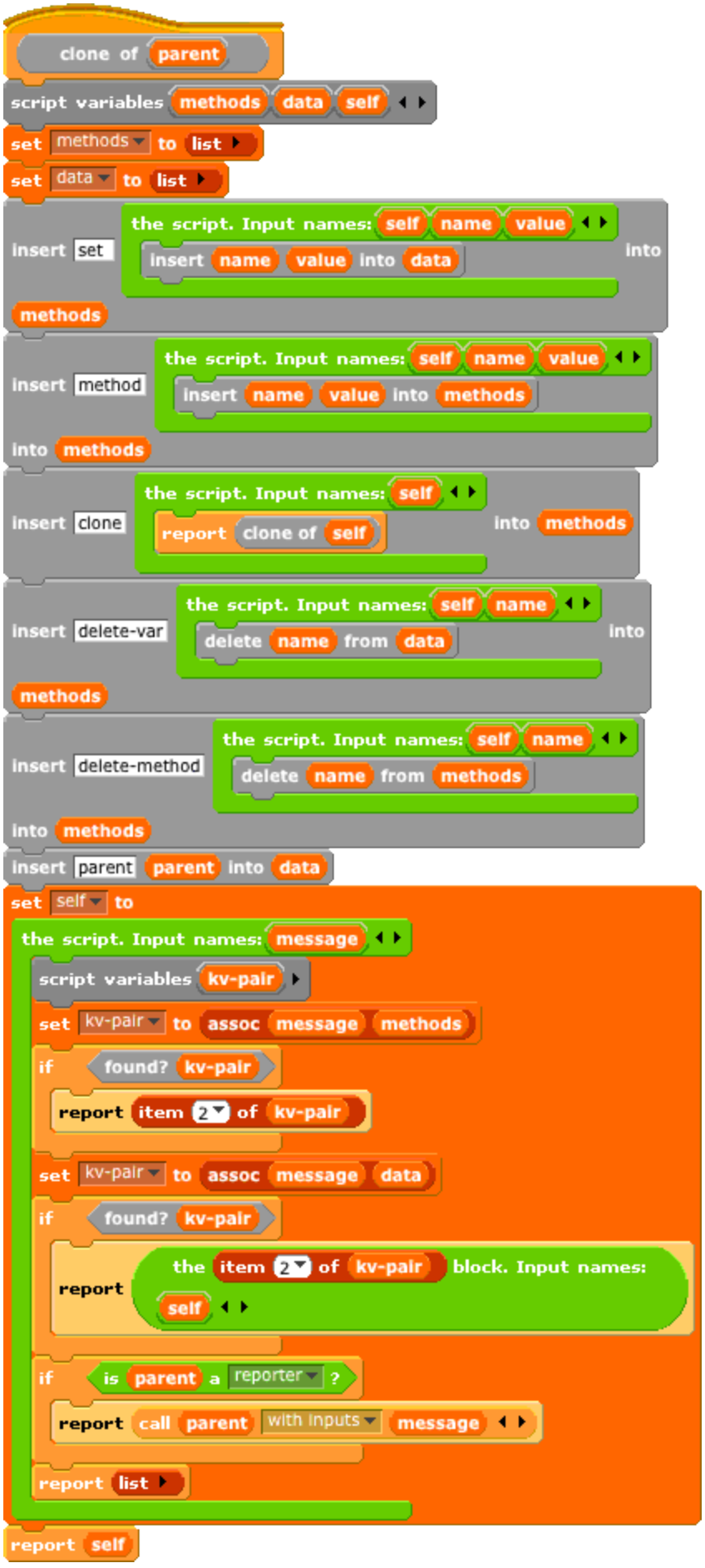


As in the class/instance version, an object is represented as a dispatch procedure that takes a message as its input and reports the corresponding method. When an object gets a message, it will first look for that keyword in its **methods** table. If it's found, the corresponding value is the method we want. If not, the object looks in its **data** table. If a value is found there, what the object returns is *not* that value, but rather a reporter method that, when called, will report the value. This means that what an object returns is *always* a method.

If the object has neither a method nor a datum with the desired name, but it does have a parent, then the parent (that is, the parent's dispatch procedure) is invoked with the message as its input. Eventually, either a match is found, or an object with no parent is found; the latter case is an error, meaning that the user has sent the object a message not in its repertoire.

Methods can take any number of inputs, as in the class/instance system, but in the prototyping implementation every method automatically gets the object to which the message was originally sent as its first input. We must do this so that if a method is found in the parent (or grandparent, etc.) of the original recipient, and that method refers to a variable or method, it will use the child's variable or method if the child has its own version.

The **clone of** block below takes an object as its input and makes a child object. It should be considered as an internal part of the implementation; the preferred way to make a child of an object is to send that object a **clone** message.

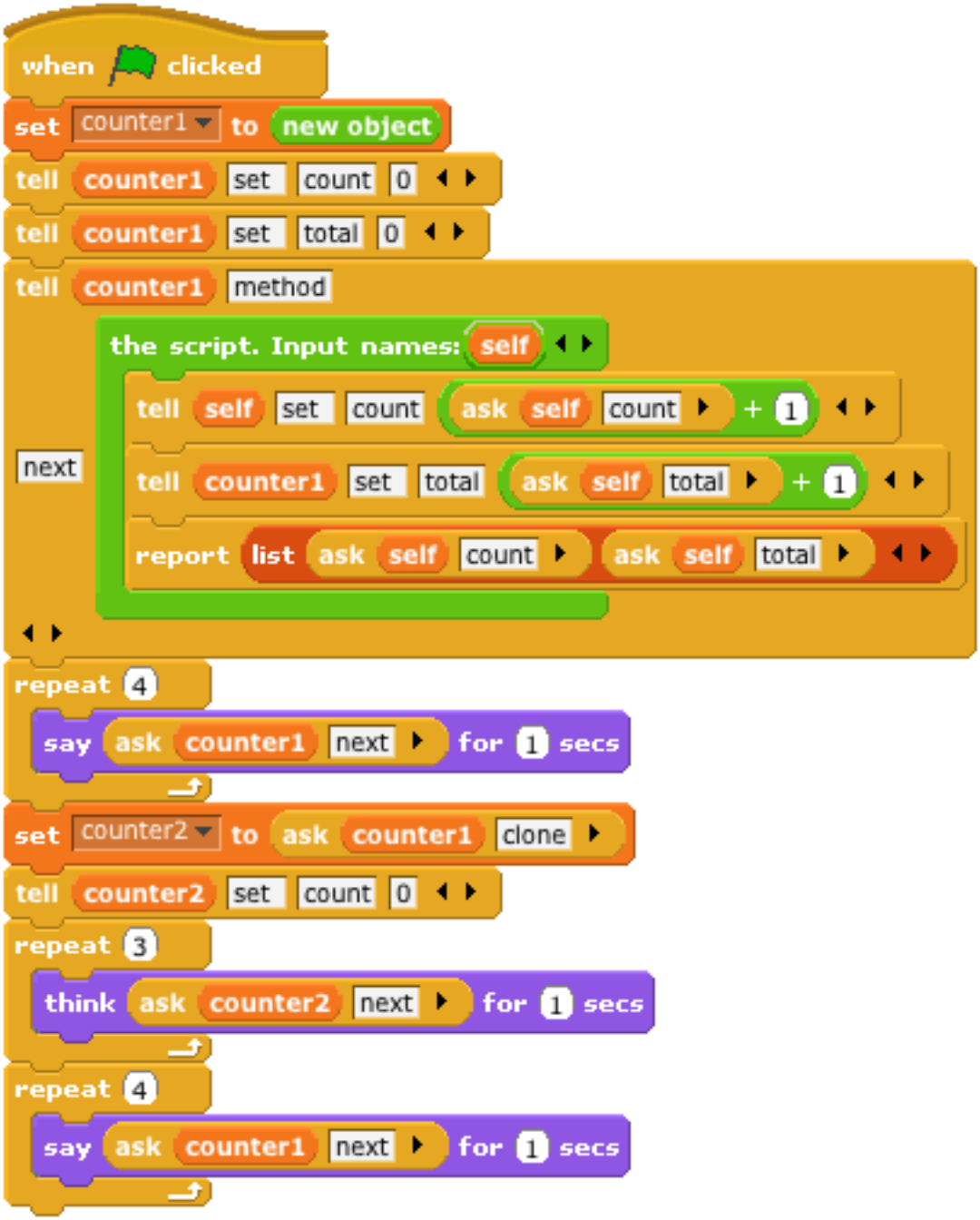

clone of parent
script variables methods data self
set methods to list
set data to list
insert set
the script. Input names: self name value
insert name value into data
into
methods
insert method
the script. Input names: self name value
insert name value into methods
into methods
insert clone
the script. Input names: self
report clone of self
into methods
insert delete-var
the script. Input names: self name
delete name from data
into
methods
insert delete-method
the script. Input names: self name
delete name from methods
into methods
insert parent parent into data
set self to
the script. Input names: message
script variables kv-pair
set kv-pair to assoc message methods
if found? kv-pair
report item 2 of kv-pair
set kv-pair to assoc message data
if found? kv-pair
report the item 2 of kv-pair block. Input names:
self
if is parent a reporter ?
report call parent with inputs message
report list
report self

Every object is created with predefined methods for **set**, **method**, **delete-var**, **delete-method**, and **clone**. It has one predefined variable, **parent**. Objects without a parent are created by calling **new object**:

As before, we provide procedures to call an object's dispatch procedure and then call the method. But in this version we provide the desired object as the first method input:

The script on the next page demonstrates how this prototyping system can be used to make counters. We start with one prototype counter, called **counter1**. We count this counter up a few times, then create a child **counter2** and give it its own **count** variable, but *not* its own **total** variable. The **next** method always sets **counter1**'s **total** variable, which therefore keeps count of the total number of times that *any* counter is incremented. Running this script should **[**say**]** and **(**think**)** the following lists:

[1 1] [2 2] [3 3] [4 4] (1 5) (2 6) (3 7) [5 8] [6 9] [7 10] [8 11]


when clicked
set counter1 to new object
tell counter1 set count 0
tell counter1 set total 0
tell counter1 method
the script. Input names: self
tell self set count ask self count + 1
tell counter1 set total ask self total + 1
report list ask self count ask self total
next
repeat 4
say ask counter1 next for 1 secs
set counter2 to ask counter1 clone
tell counter2 set count 0
repeat 3
think ask counter2 next for 1 secs
repeat 4
say ask counter1 next for 1 secs

VII. Miscellaneous features

BYOB 3 also introduces a few blocks that aren't about first class lists or procedures:

These blocks, one for commands and one for reporters, can be inserted into a script to begin true single stepping, opening a debugging window in which each step is displayed along with the opportunity to examine variables, step into the next block, step "over" the next block (not single stepping until it finishes), or continue normal evaluation.

These blocks report the constant values **true** and **false**, getting around the need to use text strings as Boolean values.

This block is used to check the type of any value. The dropdown lists all the available types: **number**, **text**, **Boolean**, **list**, **command**, **reporter**, **predicate**. Any value has exactly one of these types.

These blocks convert between single-character text strings (the character can be a letter, digit, or punctuation mark

even though the block name says "letter") and numeric codes used to represent the characters internally (the ASCII code). They're useful, for example, when you want to use a character to index into a list of values; if you have a list containing the words "Alpha," "Bravo," "Charlie," "Delta," and so on, you can turn a letter of the alphabet into the corresponding word by taking the ASCII for that letter minus the ASCII for "A," plus 1.

This block takes a list and reports its value as a text string. It is provided for compatibility with Scratch 1.4 projects; the Scratch **list** block reports the contents of a list as a single text string with all the items concatenated. When reading a Scratch project, BYOB replaces any **list** block in a script with this block.

The **copy of** block makes a new list containing the same items as the input list, more quickly than copying elements one by one. This block does not make copies of any sublists (lists that are items of the overall list); the same sublist appears in the original input and in the copy, so changing an item of a sublist in one of these lists affects the other.

The **script variables** block can be used in any script, whether in a sprite's script area or in the Block Editor. It creates one or more variables that are local to that script; they can be used as temporary storage during the running of the script, or may become permanent (but still only usable within the body of the script) if the script reports a procedure that makes reference to them. For scripting area blocks, this is a new capability; for the Block Editor it's an alternative to the former Make a Variable button inside the Block Editor.