# Bringing “No Ceiling” to Scratch: Can One Language Serve Kids and Computer Scientists?

**Brian Harvey,** *bh@cs.berkeley.edu*
Computer Science Division, University of California, Berkeley

**Jens Mönig,** *jens.moenig@miosoft.com*
Enterprise Applications Development, MioSoft Corporation

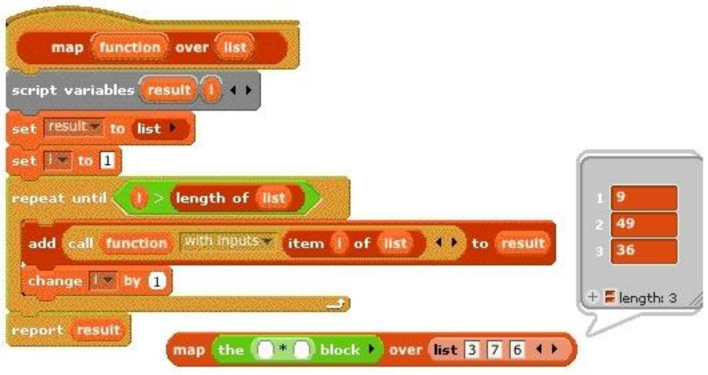
## Abstract

Scratch (http://scratch.mit.edu) is a computer programming language for children, with a graphical drag-and-drop user interface. It is a descendent of Logo, developed at the MIT Media Lab. A small but growing trend among universities is to develop computer science courses for non-majors using Scratch as the programming environment, because it isn’t threatening—the same reason it works for kids. Also, the visible use of multiple threads in Scratch provide a simple introduction to parallelism. One such course was piloted this year at the University of California, Berkeley: “The Beauty and Joy of Computing.”

But Scratch has weaknesses as a programming language. Most notably, it lacks procedures, so it can’t convey the impressive phenomenon of recursion, one of the central ideas of computer science (and also one of the central ideas of early Logo pedagogy). Its support for data structures is also weak. These weaknesses aren’t oversights; the designers of Scratch deliberately avoided cluttering the language with anything a child might find threatening.

To serve these two audiences, it has been proposed to split the Scratch community with two versions of the language, one for kids and one for advanced users. We believe that this is not necessary. By taking key ideas, such as procedures as first class data, from the Scheme language, we can add only a few features to Scratch and still make it powerful enough to support a serious introductory computer science curriculum. Furthermore, the graphical interface of Scratch makes the reification of procedures as data seem much less abstract and intimidating to novices.

Here is an example of writing a higher-order function MAP and using it with a reified procedure as an argument:



```
map function over list
script variables result i
set result to list
set i to 1
repeat until i > length of list
  add call function with inputs item i of list to result
  change i by 1
report result
```

```
map the ( * ) block over list 3 7 6
```

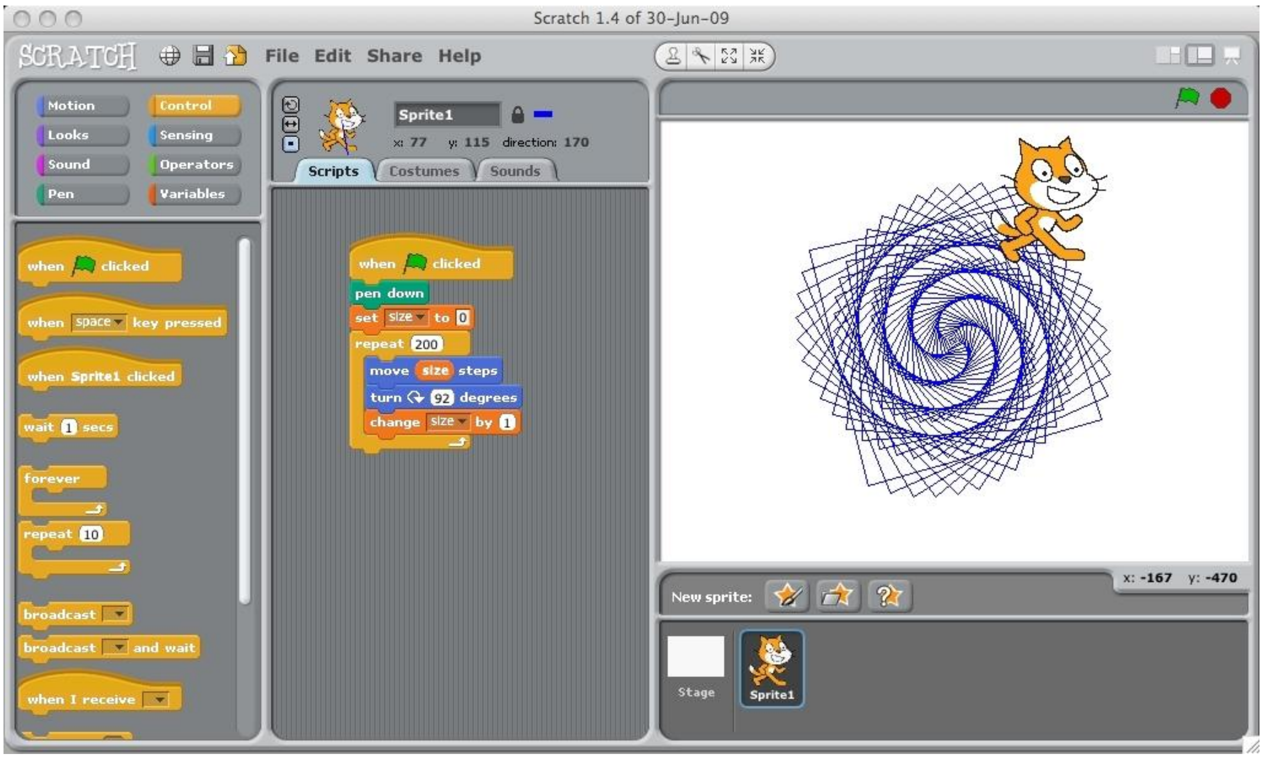
*Figure 1. MAP function definition and an example of its use*

### Keywords

Scratch; computer science; lambda; first class procedures; education; programming

## Introduction: Scratch

Scratch (Resnick et al., 2009) is a programming language for children in which no keyboard skill is needed, because the primitive program elements are available in drag-and-drop menus:



```
when clicked
pen down
set size to 0
repeat 200
    move size steps
    turn 92 degrees
    change size by 1
```

*Figure 2. The Scratch menus, scripting area, and stage*

Scratch illustrates object oriented programming in the form of multiple animated sprites with shapes taken from its own library or imported from any picture file. Each sprite has its own script area and its own local state variables. A “broadcast” primitive provides a rudimentary form of message passing, but with the limitations that messages can’t be directed to an individual sprite, and the corresponding method scripts can’t take arguments or return values.

These limitations are a deliberate part of the Scratch design, which has a a primary goal that every aspect of the language should be intuitive even to young children. As another example, until version 1.3, Scratch had no data aggregation mechanism at all; the lists added in 1.3 are, by default, visible on the stage, so that their behavior is apparent. Deep structures (lists of lists) are ruled out.

### Scratch goes to the university.

In recent years, computer science departments at several universities have been making efforts to attract more students. One motivation for this has been the underrepresentation of women and minorities among computer scientists; in recent years a more general motivation has been a decline in the total number of computer science majors, even as job opportunities in the field have been increasing. Some countries, including the United States, have declared solving the shortage of students in scientific fields and computer science specifically to be a national priority.

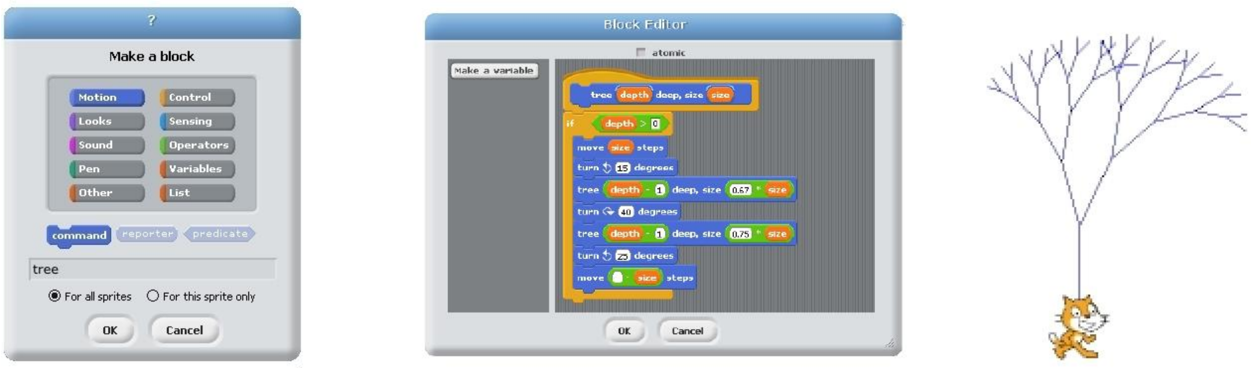
The effort has taken many forms. One example is a shift from a narrow focus on programming techniques to a “breadth first” or “applications first” curriculum. But the form relevant to this paper is the search for a programming language that avoids the syntactic complexities that drive away beginners. Scratch is one of several languages trying to meet this need; others are Alice (Pausch, 1995) and Stagecast (Smith and Cypher, 1998).

Courses using Scratch have been developed at Harvard University (Malan and Leitner, 2007), and the College of New Jersey (Wolz et. al., 2008). One of the authors of this paper (Harvey) is developing such a course along with Daniel Garcia and a team of students led by Colleen Lewis at the University of California, Berkeley. A pilot version of the course was taught in Fall, 2009 (Shafer, 2009).

Berkeley's approach to introductory computer science has been profoundly shaped by the seminal text *Structure and Interpretation of Computer Programs* (Abelson and Sussman, 1996). In our new course, we wanted to preserve the big ideas we gleaned from their curriculum, including recursion and higher order procedures as organizing tools for flow of control. But Scratch, without the ability to define procedures, wouldn't support such a curriculum.

## Build Your Own Blocks

Fortunately for the Berkeley effort, the other author of this paper (Mönig) developed an extension to Scratch called BYOB (Build Your Own Blocks) that solved the first problem (recursion) by allowing users to create new Scratch procedures:



```
tree (depth) deep, size (size)
if <(depth) > 0>
    move (size) steps
    turn ↻ 15 degrees
    tree ((depth) - 1) deep, size (0.67 * (size))
    turn ↺ 40 degrees
    tree ((depth) - 1) deep, size (0.75 * (size))
    turn ↻ 25 degrees
    move ( - (size)) steps
```

Figure 3. Initial block creation dialog, editing the block, and the final result

Using BYOB, the Berkeley course team was able to develop a pilot curriculum including recursion and teach it to an audience of non-computer-science major 18- and 19-year-olds.

## First Class Data

Here is a motivating example for the software extensions discussed below: To introduce recursion, we wanted to use an old Logo demonstration developed by Paul Goldenberg. He begins by defining a few simple shapes:

```
to square                           to hex                         to star
repeat 4 [forward 10 right 90]      repeat 6 [forward 7 right 60]  repeat 5 [forward 12 right 144]
end                                 end                            end
```

Using these, he presents students with a non-recursive procedure that draws a V shape with a randomly chosen decoration at each end:

```
to vee
  left 45 forward 100
  run pick [square hex star]
  back 100 right 90 forward 100
  run pick [square hex star]
  back 100 left 45
end
```

Since Logo instructions are just text, it's straightforward to take a procedure name and RUN it. Paul runs VEE several times until students are accustomed to the pictures it draws. Then he edits the definition of VEE:

```
to vee
  left 45 forward 100
  run pick [square hex star vee]
  back 100 right 90 forward 100
  run pick [square hex star vee]
  back 100 left 45
end
```

He asks students to predict what will happen when the PICK procedure chooses VEE; most students draw a two-level structure. Then he runs it:

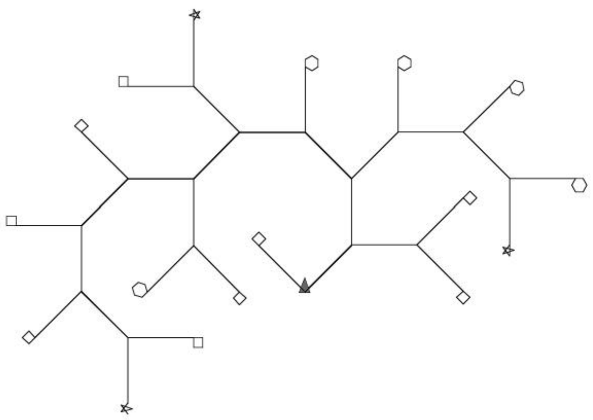


*Figure 4. Result of running the recursive VEE procedure*

Students' surprise at the complexity of the result leads into an understanding of the possibility of arbitrarily deep recursion. The use of randomness in selecting the decorations at the endpoints eliminates the need for an explicit base case in the procedure, making it easier to read and focusing students' attention on the recursive case rather than the base case.

The graphical nature of Scratch programming makes the VEE example both harder and easier in BYOB. Since a Logo program is just text, the same operations that manipulate sentences (lists of words) also manipulate programs (lists of instructions). The input to RUN is just text. In Scratch, a program is a combination of user interface elements that aren't directly representable as data. This difference required some awkwardness in the programming; the VEE procedure must broadcast a message, one of SQUARE, HEX, etc., and there must be scripts saying that when the sprite receives the SQUARE message it should run the SQUARE procedure, and similarly for the others. On the other hand, the graphical nature of BYOB program blocks *should* make it possible to show the repertoire of VEE decorations in a way that would make the program structure immediately obvious:

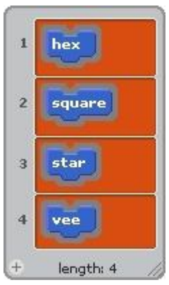


*Figure 5. A visually apparent list of Scratch blocks (procedures)*

This at-the-time-imaginary BYOB list of blocks is pedagogically preferable to Logo's list of *names* of procedures, because it eliminates the potentially confusing translation from the language's syntax (what you say in your program to refer to a value) to its semantics (the value itself)—in this case, the notation for a procedure versus the procedure itself. In BYOB, a program block shape is still a notation, but one that can't be confused with text; it has no meaning other than the procedure it represents. The pursuit of VEE's list of blocks led to the collaboration between the authors described in this paper.

The desire to put procedures into a list is an example of the general principle, due to Christopher Strachey, of *first class data.* A data type is considered first class in a programming language if instances of that data type can be

- the value of a variable
- a member of a data aggregate (array or list)
- an argument to a procedure
- the value returned by a procedure

As in most languages, numbers and text strings are first class in Scratch; procedures (blocks) are not. A procedure can't play any of the four roles listed above.

## First class lists

Consider the following menu for an ice cream shop:

- Sizes: small, medium, large
- Flavors: chocolate pudding, pumpkin, root beer ripple, ginger
- Media: cone, cup

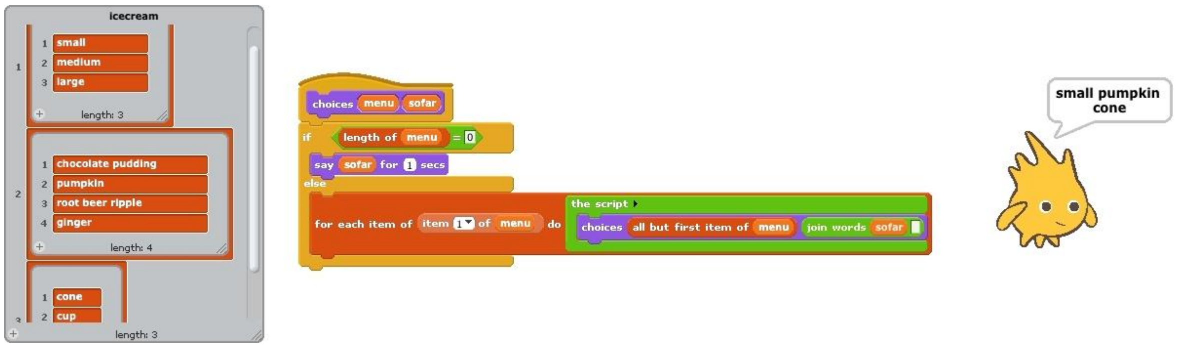
The natural representation of this menu in a computer program is as a *list of lists.* In Logo it would look like this:

[[small medium large] [[chocolate pudding] pumpkin [root beer ripple] ginger] [cone cup]]

A short Berkeley Logo program takes such a menu list and generates all possible combinations:

```
▪ to choices :menu [:sofar []]
▪   if emptyp :menu [print :sofar stop]
▪   foreach first :menu [(choices butfirst :menu sentence :sofar ?)]
▪ end
```

This program couldn't be written in Scratch 1.4, but it can be in the new BYOB:



```
choices (menu) (sofar)
if <(length of (menu)) = 0>
  say (sofar) for 1 secs
else
  for each item of (item 1 of (menu)) do (the script ▸ choices (all but first item of (menu)) (join words (sofar) [ ]))
```

*Figure 6. The menu as a list of lists, the CHOICES program, and the result of running it*

Given lists of lists, any other desired data structures can be implemented: hash tables, trees, heaps, and so on. Thus, there is no need to clutter the basic Scratch menus with such derived types; they can be defined in libraries that can be loaded only when needed.

## First Class Procedures

The pilot version of our new course included only half of the planned eventual topics. One that we left out not only for lack of time but because Scratch/BYOB wouldn't support it was higher order procedures: procedures that take other procedures as arguments. These have proven to be a powerful capability of Logo, and one that users can implement themselves. They are useful, for example, as an alternative to recursion or to looping with index variables when the programmer wants to process all of the elements of a list in a uniform way. For example, the MAP function takes two inputs: *a function* and a list. It returns a list computed by applying the given function to each element of the given list. In Logo, a function can be represented as text, either as the name of a defined function or as a list containing a Logo expression. But in the original BYOB, there was no way to encapsulate a procedure as data that can be input to another procedure—blocks are not first class.

To fix this, we have added to BYOB an equivalent to the Lisp LAMBDA, a way to turn procedures into data. In the Scratch context we need two versions, one for an individual block that's most useful for reporters (functions) and another for scripts, most useful for commands (action scripts):

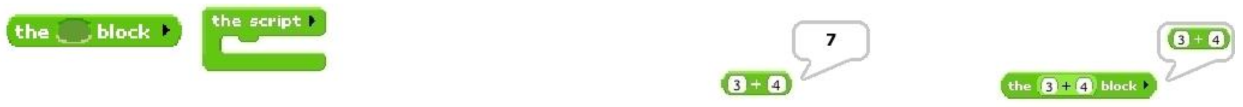


*Figure 7. Procedure encapsulation blocks, value of a block vs. its encapsulation*

Once we have blocks and scripts as first class data, we want to be able to run them, the equivalent of Lisp's APPLY procedure. This, too, takes two forms in Scratch, one for commands and one for reporters:

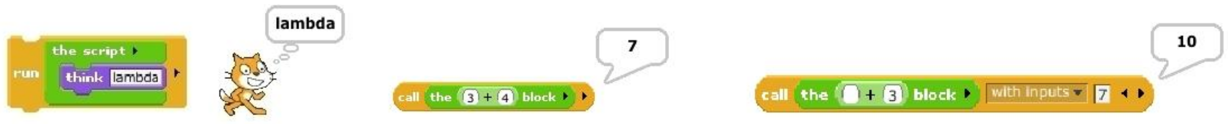
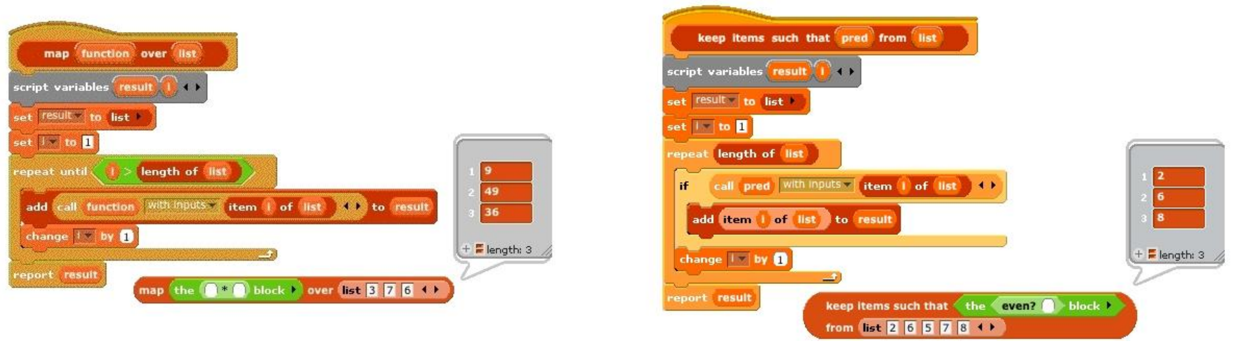

run the script think lambda
lambda
call the 3 + 4 block
7
call the ( ) + 3 block with inputs 7
10

*Figure 8. The RUN and CALL blocks, without and with arguments*

Figure 8 shows that the arrow at the right end of the RUN and CALL blocks can be clicked to expose as many slots as desired for providing arguments to the procedure being called. In almost all cases, BYOB can figure out where in the procedure the argument(s) should be used, but the arrow at the end of the THE BLOCK and THE SCRIPT blocks can be clicked to expose slots in which explicit formal parameters can be added and then dragged into the encapsulated block or script if necessary.

Encapsulating a block and then de-encapsulating it again by calling it may seem futile, but this combination allows us to write arbitrary higher order functions:



```
map function over list
script variables result i
set result to list
set i to 1
repeat until i > length of list
    add call function with inputs item i of list to result
    change i by 1
report result

map the ( ) * ( ) block over list 3 7 6
1 9
2 49
3 36
length: 3

keep items such that pred from list
script variables result i
set result to list
set i to 1
repeat length of list
    if call pred with inputs item i of list
        add item i of list to result
    change i by 1
report result

keep items such that the even? ( ) block from list 2 6 5 7 8
1 2
2 6
3 8
length: 3
```

*Figure 9. The MAP and KEEP higher order functions, with examples of use*

## Argument type declarations

Scratch blocks use different shapes and colors to distinguish among three data types in the argument slots:

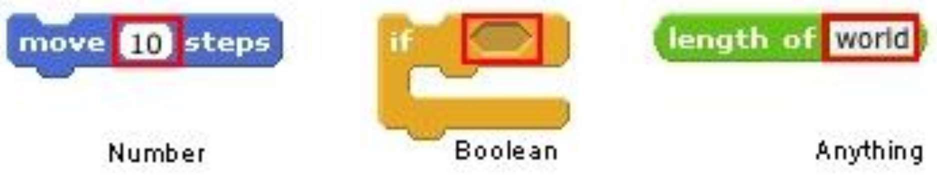


*Figure 10. Scratch type shapes.*

Also, the blocks themselves come in three shapes: the jigsaw-puzzle-piece commands such as MOVE, the oval reporters such as LENGTH OF, and the hexagonal predicates that fit into the hexagonal Boolean slots.

We have not added to the three block shapes, but because we've added lists and procedures as first class data types, we have created new input shapes for them:

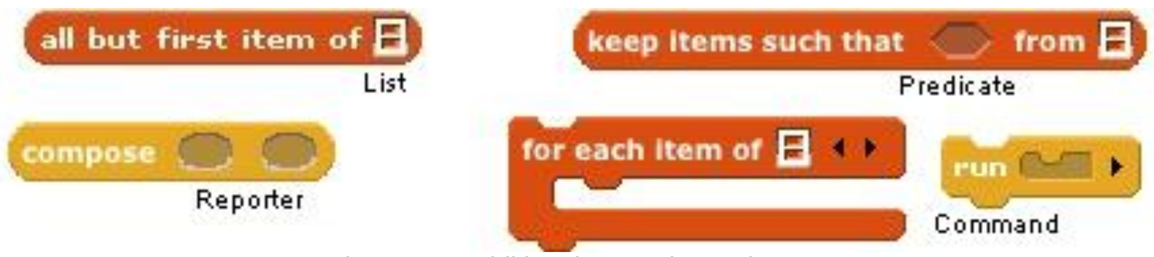
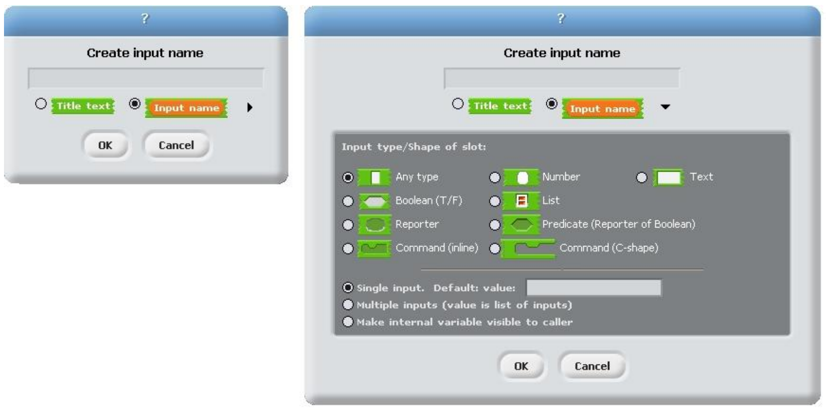


*Figure 11. Additional BYOB input shapes.*

Since we represent reified blocks and scripts visually with grey borders, these shapes are similar to the shapes of the data that match them.

Our goal is that these input type declarations help the user, not bind the user. Also, this entire feature is optional; by default, the block editor makes all input slots be of type Anything, and the programmer must access a special menu to choose a more restrictive type:



?
Create input name
○ Title text [✓ Input name]
OK Cancel

?
Create input name
○ Title text [✓ Input name]
Input type/Shape of slot:
[✓ Any type] ○ Number ○ Text
○ Boolean (T/F) ○ List
○ Reporter ○ Predicate (Reporter of Boolean)
○ Command (inline) ○ Command (C-shape)
[✓ Single input. Default: value:]
○ Multiple inputs (value is list of inputs)
○ Make internal variable visible to caller
OK Cancel

*Figure 12. Short and long input name menus*

For input slots declared to be a procedure type (command, reporter, or predicate), we allow a special drag-and-drop technique that wraps an implicit THE BLOCK or THE SCRIPT around the input block or script, shown as a grey border:

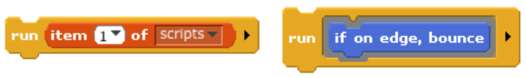


*Figure 13. Normal and implicit-script slot filling*

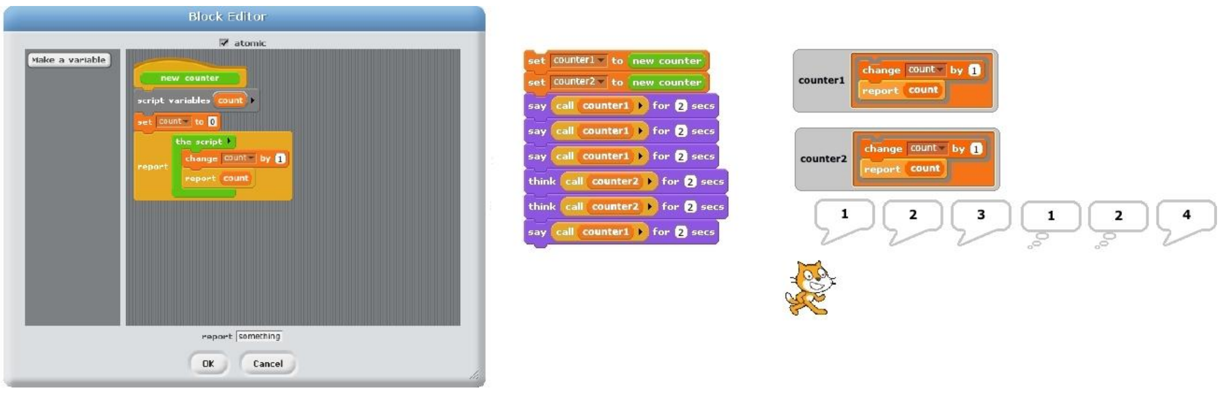
## Object Oriented Programming

It may come as no surprise that adding function encapsulation to a language allows it to support functional programming style. But this augmented BYOB also supports object oriented programming, in two ways. First, Scratch has a natural set of objects: the sprites that it uses to control animation. BYOB gives sprites object-like behaviours beyond those designed into Scratch. Second, first class procedures allow the explicit programming of classes and instances. Students who build objects themselves may arguably understand the nature of object oriented programming better than those who encounter an OOP language as a black box.

Central to the OOP paradigm is the idea of message passing. Scratch was designed with a very simplified version: The Scratch programmer can broadcast a message to all sprites, but can't direct the message to a specific sprite. Also, Scratch messages take no arguments, and the scripts that respond to a message cannot return a value to the sender. But both of these limitations are transcended once a script can be the value of a variable.

The Scratch "Sensing" menu includes a reporter block called "*variable* OF *sprite*." The original intent of this feature is, for example, to allow one sprite to find out the X and Y coordinates of another sprite's position. But when this block is used in the input slot of RUN or CALL, it allows one sprite to invoke a method in another sprite! (We have added a variant of RUN called LAUNCH that starts a new thread to run the method asynchronously.) The fact that this very useful OOP capability was automatically implied by a mechanism we created for a different purpose shows how powerful the idea of procedure encapsulation is. To prepare to accept a message, a sprite must merely create a local variable by that name, whose value is a method script for that message.

Here is the simplest illustration of the second approach to OOP in BYOB. We are going to use procedures to represent both the class and the instances for a COUNTER class:



```
new counter
script variables count
set count to 0
report the script
    change count by 1
    report count
report something
```

```
set counter1 to new counter
set counter2 to new counter
say call counter1 for 2 secs
say call counter1 for 2 secs
say call counter1 for 2 secs
think call counter2 for 2 secs
think call counter2 for 2 secs
say call counter1 for 2 secs
```

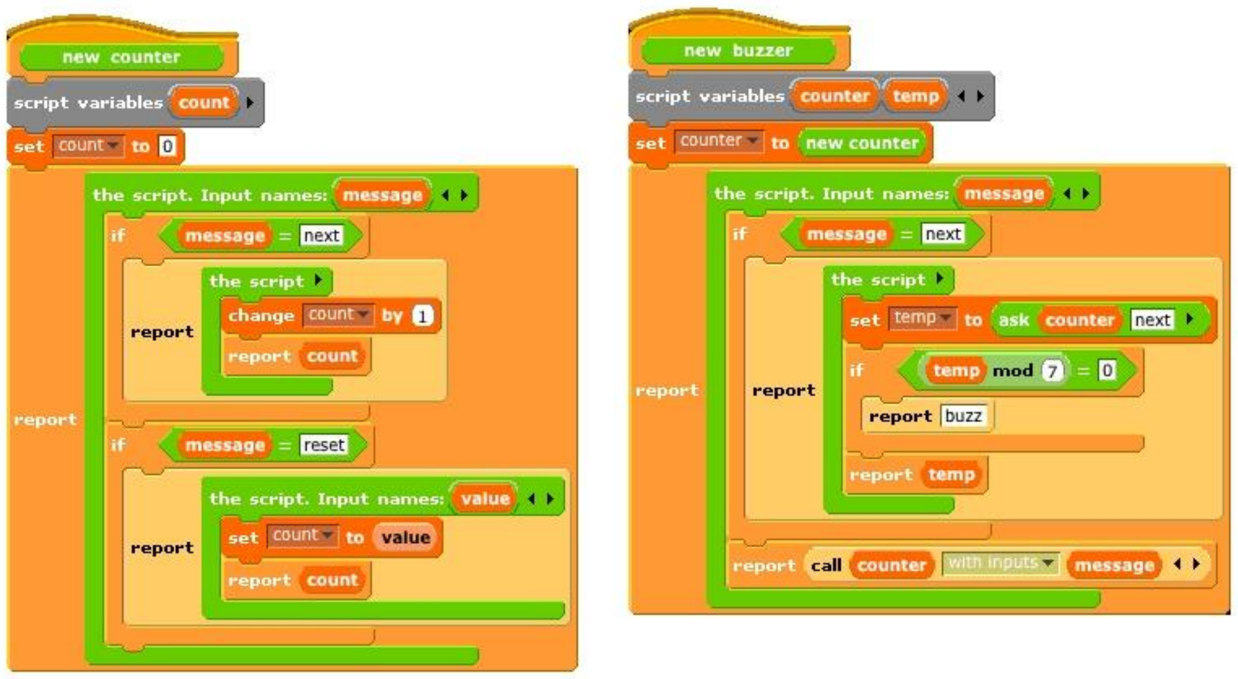
```
counter1  change count by 1
          report count
counter2  change count by 1
          report count
```

Figure 14. The COUNTER class, two instances, and the result of several calls to the instances

Procedure NEW COUNTER represents the class. Every call to any BYOB block creates new block variables. For the most part, these block variables are temporary; when the block's script completes and the block returns to its caller, nothing points to the block variables and their space is reclaimed. But this procedure returns a procedure! Since the latter refers to the COUNT variable, that variable is *not* temporary, but acts as a persistent local state variable for the counter instance. The procedure created by the THE SCRIPT block represents an instance. In Figure 14, we create two counter instances, each with its own COUNT variable. Each of them starts at 0 and is increased each time the *instance* (COUNTER1 or COUNTER2) is called. The figure shows that COUNTER1 remembers its count even after COUNTER2 is called.

## Message Passing

The simple counter in Figure 15 shows how the ability of a procedure to return another procedure gives us the persistent local state variables that OOP requires. But to follow the object metaphor faithfully requires message passing—the ability to ask an object to do one of a repertoire of actions. It's not hard to extend the basic idea in Figure 14 to allow messages by representing an object by a *dispatch procedure* that takes a message (just a word) as its argument and returns a method (a procedure).



```
new counter
script variables count
set count to 0
report
    the script. Input names: message
        if message = next
            report
                the script
                    change count by 1
                    report count
        if message = reset
            report
                the script. Input names: value
                    set count to value
                    report count
```

```
new buzzer
script variables counter temp
set counter to new counter
report
    the script. Input names: message
        if message = next
            report
                the script
                    set temp to ask counter next
                    if temp mod 7 = 0
                        report buzz
                    report temp
        report call counter with inputs message
```

*Figure 15. Message passing counter, and inheritance by delegation*

The NEW COUNTER block in Figure 15 represents a class with two messages, NEXT and RESET. Each instance of the class is a dispatch procedure, created by the large outer THE SCRIPT block. Each of the two inner THE SCRIPT blocks creates a method. The NEXT method increments and reports the count. The RESET method takes an argument, and sets the count to that value. Sending an object a message is a two-step process: First call the dispatch procedure with the message to get the method, then call the method.

## Inheritance

Message passing and local state variables provide the essence of the OOP paradigm, but to make it practical we need inheritance, the ability to reuse existing methods. The NEW BUZZER example in Figure 15 illustrates the easiest way to implement inheritance, namely delegation. A buzzer object is just like a counter object, except that if the new count is divisible by 7, it returns the word BUZZ instead of the number. Only the NEXT method must be changed; in all other ways a buzzer should behave like a counter. (In this simplified example there is only one other method, RESET, but you should imagine a class with many methods.) Every instance of a buzzer contains within it an instance of the counter class. (This is accomplished by the SET block near the top of the script.) The buzzer's dispatch procedure has an explicit script for the one message it handles differently (NEXT); if any other message is received, the buzzer merely forwards the message to its internal counter.

## Conclusion

We believe that we can support a wide range of introductory computer science courses in Scratch with very few additions to its repertoire of primitive blocks:

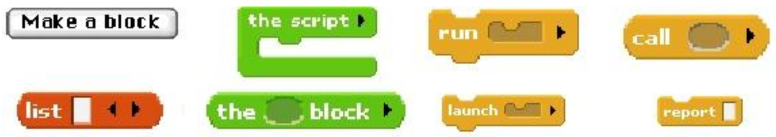


*Figure 16. All we added*

We had to consider and solve many technical problems, such as scope of variables, but our claim is that none of these will be visible to the typical Scratch programmer. The key is to begin with the ideal of first class data in mind, and to recognize in particular that procedures as data enable us to write *in Scratch itself* the many computer science examples that would otherwise have to be added as primitive blocks.

We are not suggesting that the children who are Scratch's main audience will start writing recursive procedures (at least, not the youngest of them), higher order functions, or object classes. What we argue is that the additional features we need to support computer science students will be inobtrusive. (Computer science students will, no doubt, use libraries of blocks written by their teachers, and some of those might be cluttered. But that won't affect the core of the language.) And there are many benefits to keeping the most expert Scratch programmers as part of the same community as the beginners; a separate Scratch for older users would lose those benefits. (As we are writing this paper in January, the software is just getting to the point at which we'll feel ready to try it out on kids. We hope to have some experiences to report by the summer.)

We gratefully acknowledge the ideas we borrowed from the Scheme language (Steele and Sussman, 1975) and from *Structure and Interpretation of Computer Programs* (Abelson and Sussman, 1996), as well as the brilliant inventions of the Scratch team at MIT.

## References

Abelson, H., and Sussman, G.J., with Sussman, J., *Structure and Interpretation of Computer Programs,* 2nd edition, MIT Press, 1996.

Malan, D., and Leitner, H., *Scratch for Budding Computer Scientists,* SIGCSE Procedings, 2007.

Pausch, R., Burnette, T., Capeheart, A.C., Conway, M., Cosgrove, D., DeLine. R., Durbin. J., Gossweiler, R., Koga, S., and White, J., *Alice: Rapid Prototyping System for Virtual Reality*, IEEE Computer Graphics and Applications, May 1995.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). *Scratch: Programming for All.* Communications of the ACM, vol. 52, no. 11, pp. 60-67 (Nov. 2009).

Shafer, R. *Oh! The Beauty and Joy of Computing,* Innovations Volume 3, Issue 10, UC Berkeley College of Engineering, December 2009. See also *http://inst.eecs.berkeley.edu/~cs10*

Smith, D.C., and Cypher, A., *Making Programming Easier for Children,* in Druin, A., ed., *The Design of Children's Technology,* Morgan Kaufmann, San Francisco, 1998.

Steele, G.L., and Sussman. G.J. *Scheme: An interpreter for the extended lambda calculus*. Memo 349, MIT Artificial Intelligence Laboratory, 1975.

Wolz, U., Maloney, J., and Pulimood, S.J., *"Scratch" Your Way to Introductory CS,* ACM SIGCSE Bulletin, Volume 40 , Issue 1, March 2008.