“Why Do We Have to Learn This Baby Language?”

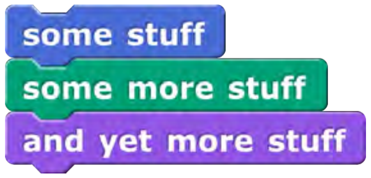
Brian Harvey, Teaching Professor Emeritus, University of California, Berkeley

I get it. You’re a teenager, it’s the first day of what was billed as a serious computer science course, and you’re confronted with drag-and-drop blocks in primary colors. You’ve seen this before, back when your age had only one digit, and you played around with Scratch. Been there, done that.

But Snap! (Build Your Own Blocks) isn’t Scratch. It’s a very serious programming language, in disguise. The reason for the disguise is that most programming courses spend most of their time and effort on the details of the *notation* used by whatever programming language they choose. Here’s the classic example: In your “grownup” programming language you write

```
some stuff; some more stuff; and yet more stuff;
```

These are three instructions to the computer, with semicolons in between. Now, what about that red semicolon at the end? If the language is Java, C, or C++, that semicolon is *required.* If it’s Perl, Pascal, or PL/I, that semicolon is *forbidden.* Not exactly easy to learn. But if, instead, the code looks like this:



then there’s no need to remember punctuation rules. The visual layout of the code *is* the notation.

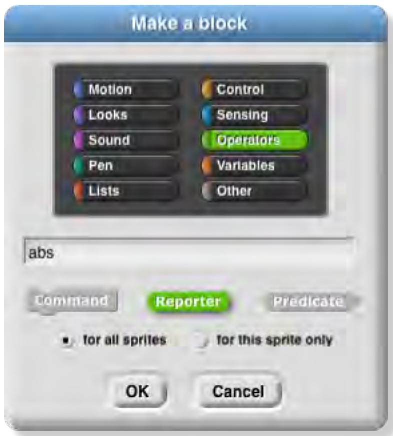
And, of course, there’s a benefit to following the visual design of Scratch: Many kids have grown up with Scratch, and so they can—*you* can—dive right in and use the parts of Snap*!* that are like Scratch, and then learn the new parts.

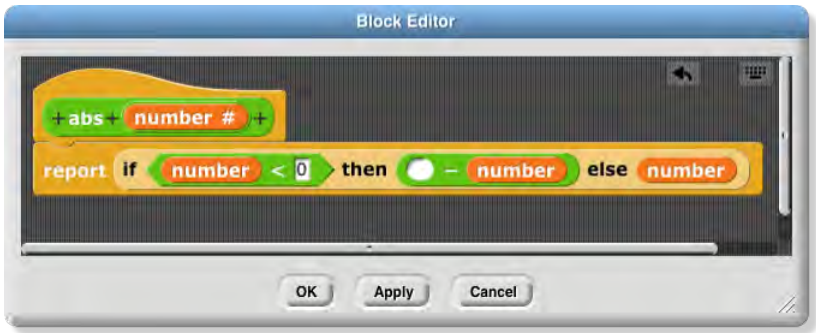
Snap*!* is an extended reimplementation of Scratch featuring first class procedures, first class lists, and first class sprites with inheritance. (What does “first class” mean? See page 4.) An earlier version, BYOB, was a modification of the Scratch 1.4 source code, but the current version is an entirely separate program, even though its user interface looks like Scratch 1.4 and it includes almost all of the Scratch 1.4 primitive blocks. BYOB was originally developed by Jens Mönig; as of BYOB 3.0, I joined as co-developer. (The code is still almost all written by Jens; I have contributed to the design, libraries, and documentation, as well as online mentoring of Snap*!* users.)

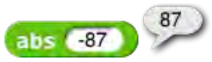
Scratch has been phenomenally successful at introducing young people to computer programming; part of that success has been the detailed, thoughtful design of each small piece of the language. In particular, they decided to *leave out* some features, to avoid confusing young users. The **tl;dr** explanation of Snap*!* is that we added those missing features. Snap*!* was designed to teach computer science courses to teenagers, while Scratch was designed to pull in younger children. A design choice may be good for the Scratch target audience, but not for the Snap*!* target audience, or vice versa. In every case, the Scratch Team was aware of the possibilities that they decided to omit. And from the beginning, don’t forget, Scratch has had parallel processing, with multiple scripts and sprites all running at once. This is a significant Big Idea that isn’t so readily usable in “grownup” languages.

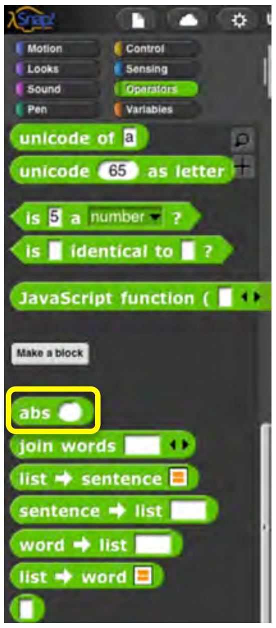
Build Your Own Blocks

The ability to build custom blocks was the first goal of BYOB, as reflected in its name. A limited version of this capability, allowing only command blocks, later became part of Scratch 2.0.







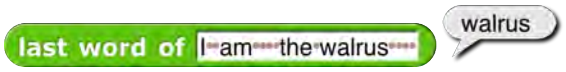


When you start to make a custom block, you select its palette color, any of the standard ones or a gray “Other” category, and its shape (command, reporter, or predicate). Like a variable, the block can belong to one sprite or be usable by any sprite. After you’ve made those choices, Snap*!* opens a *block editor* window, like the scripting area but just for this one block definition. After you’ve finished, the definition is not visible (although you can bring back the block editor, or keep it open while you debug the block), but the block is now found in the palette.

This is very different from the Scratch custom block interface. In Scratch, all custom blocks are found in their own palette category (“My Blocks”), and the scripts that define the blocks remain visible in the scripting area. It’s not that one of these interfaces is “right” and the other “wrong,” but they do reflect very different ideas about how custom blocks will be used.

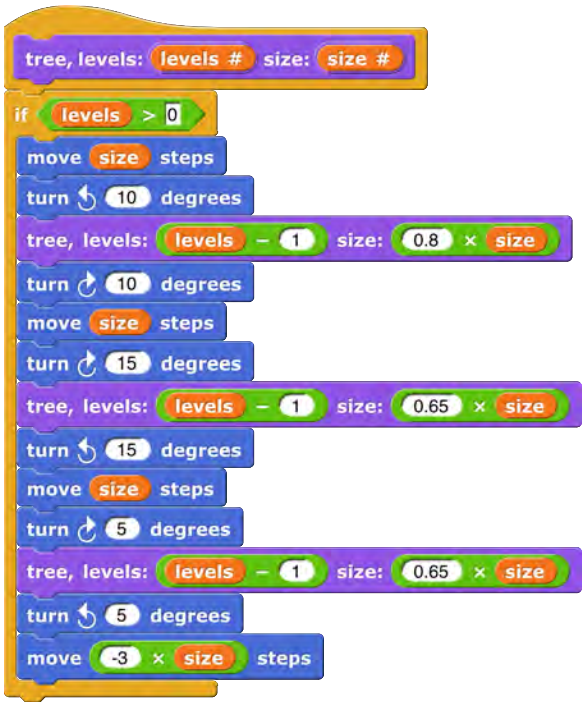
The Scratch design treats a custom block as *unusual*, very different from a primitive block. Scratch keeps block definitions visible in the scripting area so that all of your code is visible to you. But this is practical only if you define a small number of custom blocks. If you have 40 custom blocks in a project, your scripting area will be unmanageable.

The Snap*!* design tries to make custom blocks *ordinary*, just like primitive blocks; ideally, once you’ve written a block, you can forget it’s homemade. You can have as many of them as you want, without filling up your scripting area. This is important not just to keep your screen neat, but also to emphasize the idea of *abstraction*, perhaps the central idea in computer science: You write programs in an environment full of procedures that let you think about the things you want to do, rather than about how to implement them. You can *extend* Snap*!* with a block library, a collection of blocks for a particular purpose, written in Snap*!* itself. For example, below the abs block in the picture above are a collection of blocks that implement the idea of a *sentence:* a text such as “I am the walrus” is treated as four words, rather than 15 characters. “I am   the walrus   ” is the same sentence, even though it’s a different text string. The library has about a dozen blocks for tasks such as this one:

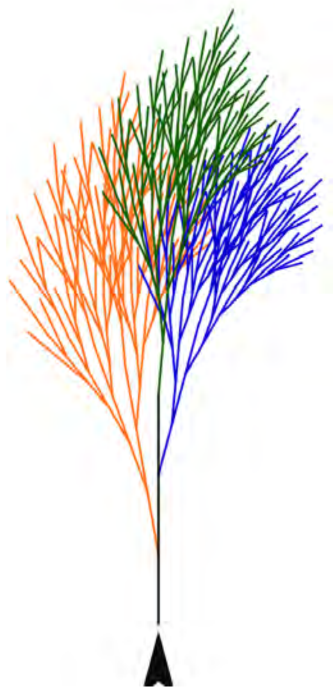


Inside that block are some ugly details about looking for space characters to work out where the last word is in the text. But you don't have to *read* those ugly details; you want to find the last word of a text, so you use that block, and it works.

Recursion. The ability to name a script by making a block out of it isn't just important for neatness and code reuse. It enables the block to use *itself* to solve a smaller subproblem. The classic example is a fractal tree. Here's the program and the result of running it:



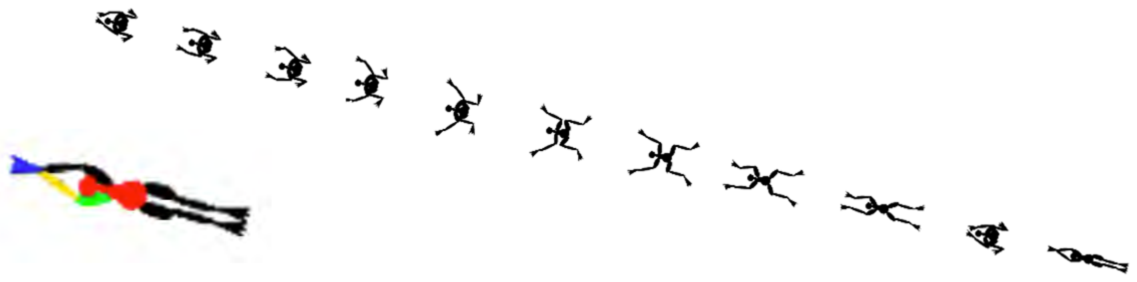
```
tree, levels: (levels #) size: (size #)
if <(levels) > 0>
  move (size) steps
  turn ↺ 10 degrees
  tree, levels: ((levels) − 1) size: (0.8 × (size))
  turn ↻ 10 degrees
  move (size) steps
  turn ↻ 15 degrees
  tree, levels: ((levels) − 1) size: (0.65 × (size))
  turn ↺ 15 degrees
  move (size) steps
  turn ↻ 5 degrees
  tree, levels: ((levels) − 1) size: (0.65 × (size))
  turn ↺ 5 degrees
  move (-3 × (size)) steps
```



(The program draws in all one color, but the picture is colored here to show you how a tree includes three smaller trees, drawn by the three recursive invocations: first orange, then blue, then green.)

Nested Sprites

Snap*!* gives you the ability to make a collection of sprites that move together. The canonical example is a skeleton in which each bone is attached to one closer to the spine; the entire skeleton can rotate together, or a particular bone can rotate relative to the whole.

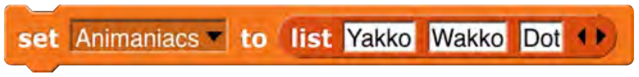


The swimmer is made of 13 sprites. In the enlargement at the left, four of the sprites have been colored. The red sprite, the head and torso of the swimmer, is the "anchor" sprite. Four sprites are attached as "parts" of that anchor: the upper arms and legs. (The left upper arm is colored green above.) Each of those is an anchor for the corresponding lower arm or leg; the left lower arm is yellow above. Finally, each lower arm or leg is the anchor for a hand or foot, such as the blue left hand above.

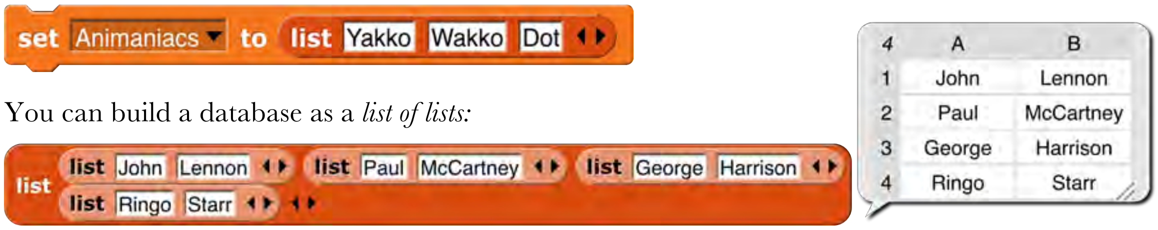
First class lists

A data type is *first class* in a language if it can be used in all the same ways as any other data type. For example, in Scratch it's easy to make a list of numbers, but you can't make a list of *lists*; numbers are first class, but lists aren't. One of the slogans of Snap*!* is "**Everything first class.**" In Scratch a list is essentially a kind of variable: It has a name and a value. The only difference is that the value has sub-values (the items). Scratch provides a red oval for the list, like the orange one for a variable, but the value it reports isn't the actual list; it's a *string*, the concatenation of all the items, separated by spaces. This works because only strings and numbers can be items in a Scratch list.

Scratch lists combine two capabilities that are logically separate. The main capability of a list is that it can add, change, delete, and report items. The second capability is associating a name with the list, and that should instead be the job of an ordinary variable. If you want a list with items "Yakko," "Wakko," and "Dot," and you want to call the list "Animaniacs," you should be able to say

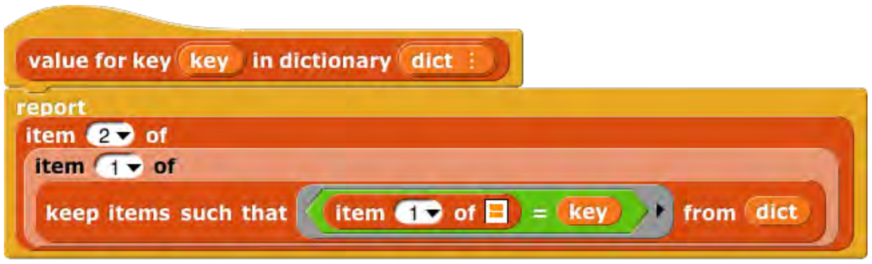


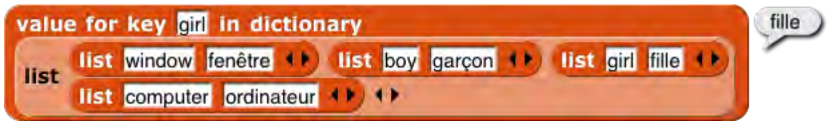
You can build a database as a *list of lists:*



4	A	B
1	John	Lennon
2	Paul	McCartney
3	George	Harrison
4	Ringo	Starr

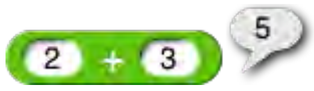
You can implement other data structures, such as a dictionary:

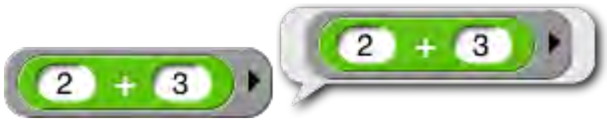

```
value for key (key) in dictionary (dict)
report
  item 2 of
    item 1 of
      keep items such that (item 1 of [] = key) from dict
```



First class procedures

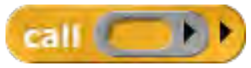
First class lists are all you need to invent any data structure. But we'd also like to be able to invent a new *control* structure—maybe you want `repeat while` instead of `repeat until`, for example. For that we need *procedures* (blocks and scripts) as first class data. The gray rings above make that possible. A picture is worth 1000 words:

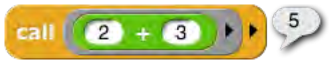




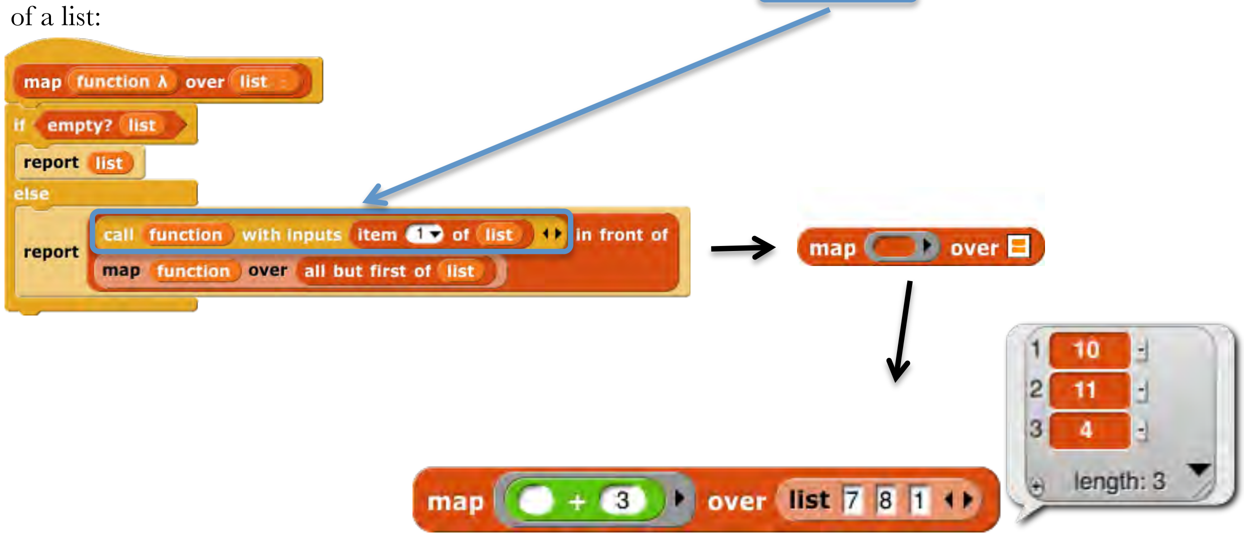
On the left, clicking the unringed 2+3 gives 5.
On the right, putting 2+3 in a ring gives ***the*** `2 + 3` ***block itself.***

The reason to create a ringed procedure is that later you'll want to call it.





Users can create their own control structures, e.g., abstracting a repeated call of a function for each item of a list:


map function λ over list
if empty? list
report list
else
report call function with inputs item 1 of list in front of map function over all but first of list
map ( ) over
map (+ 3) over list 7 8 1
1 10
2 11
3 4
length: 3

When reading the map example, did you understand right away that each list item is plugged into the empty slot in the block? Elementary school teachers all know that if you show an eight-year-old "*x*+3=7" and ask "what's *x*?" you're likely to get a blank stare, but if you show the same kid "□ +3=7" and ask "what number goes in the square?" you're very likely to hear "four." Snap*!* uses that knowledge in the design of the notation for functions. When you call a gray-ringed function with an input, call looks for an empty input slot inside the ring, and that's where the input value is inserted. This notation makes simple function calling obvious, in the Scratch spirit of trying to use the visual representation to make ideas obvious.

First Class Sprites with Inheritance

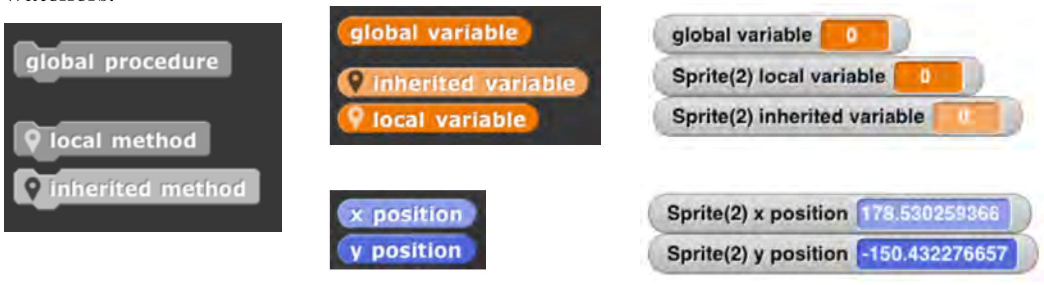
Object oriented programming is a style based around the abstraction *object:* a collection of data and methods (procedures, which in Snap*!* are just more data) that you interact with by sending it a message (just a name, maybe in the form of a text string, and perhaps additional inputs). The object responds to the message by carrying out a method, which may or may not report a value back to the asker. Some people emphasize the *data hiding* aspect of OOP (because each object has local variables that other objects can access only by sending request 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 Snap*!* users it's the simulation aspect that's important. The Snap*!* approach is therefore less restrictive than that of some other OOP languages; we give objects easy access to each others' data and methods.

Scratch provides a very natural set of objects, namely, sprites. Object oriented programming rests on three legs:

- **Persistent local state.** An object must be able to remember its history from one action to the next, in variables whose names are private to that object. Scratch sprites have this, in the form of "for this sprite only" variables.
- **Message passing.** One object has to be able to ask another object to do something, by sending it a message. Scratch doesn't exactly have this capability, but a sprite can broadcast a message to *all* sprites. Snap*!* adds true message passing, in which messages can be sent to individual sprites. They take the form of blocks or scripts, which are carried out using the local blocks and variables of the receiver.
- **Inheritance.** It's very common for two or more objects to share some methods (local procedures). Object oriented programming would be hopelessly complicated if the methods had to be duplicated by hand in each object. So object oriented languages provide a way for one object to *inherit* methods and data from another. Starting in Scratch 2.0, there is a restricted form of inheritance: It's possible to clone a sprite; the resulting sprite is a *copy* of the original, and is temporary; when the program stops running, the temporary clones are deleted. Temporary clones are good for things like the bricks in a Breakout game; they all have the same behavior, just different positions, and can be recreated at the start of each new game. But sometimes a project needs *permanent* clones; one example would be the four ghosts in a Pacman game, each of which has a slightly different behavior (methods). Snap*!* provides both kinds of cloning: Clones created in a program, using the create a clone of myself block or, since sprites are first class data in Snap*!*, the new a new clone of myself reporter, are temporary. Clones created from the user interface, e.g., by right-clicking on a sprite icon in the sprite corral and choosing the "clone" option from the context menu, are permanent; they appear in the sprite corral, are independently editable, and are saved with the project.

A Snap*!* clone is *not* a *copy* of its parent; it actually shares the attributes of the parent. So, for example, if a costume of the parent is deleted, it's gone in the child also. However, if something is changed *in the child,* then the connection of that attribute with the parent is broken, and the child has a separate one of whatever it is. If the child imports a new costume, its entire wardrobe (the list of all its costumes) is copied from the parent, then the new costume is added, and thereafter the two sprites are independent with respect to costumes.

It could be confusing if a child shares some attributes of its parent but not others. To help avoid this confusion, attributes represented by palette blocks are *ghosted* in the child's palette and in variable watchers:



Prototyping vs. Class/Instance OOP

Readers who have programmed in class/instance languages may feel that what's described here isn't OOP. But classes aren't necessary, and are actually harmful in a dynamic, interpreted language that encourages tinkering. Class/instance programmers typically design their entire class hierarchy before actually writing code. That's an appropriate discipline for a large group of programmers working on the same program, but not for one kid designing a project on the fly. What the kid wants to do is build, for example, a dog sprite, and program it with behaviors similar to those of his or her pet Cocker Spaniel. *Then* the kid decides to add a Rottweiler to the project. It should inherit most of the behaviors of the Cocker Spaniel, but with a few differences. This OOP structure is called *prototyping* or *prototype-based OOP.*

If you actually want classes, make a sprite, give it children, then hide the parent. The parent is then effectively a class.

Another way in which this OOP design is nontraditional is that it uses blocks (procedures) as messages, and *any* block can be used, not just the ones that the receiver has declared as public methods. As explained earlier, in the Snap*!* design, data hiding is less important than simulation, the use of objects as metaphors for some real system of agents.

Note: OOP does not *require* a special OOP language. If a language has first class procedures and lexical scope, then it's possible to build an object system in the language. See the Snap*!* Reference Manual for details.

First Class Continuations

Prior to joining the Snap*!* project, I had spent 20 years teaching computer science using the best computer science book ever written, *Structure and Interpretation of Computer Programs* (*SICP*). That course uses the Scheme programming language, although it's not *about* Scheme, because it's the language that provides the greatest versatility in supporting different programming styles with the least amount of syntax to get in the way. Scheme has been the model for BYOB/Snap*!* language design since BYOB 3.0. Another of our slogans is **"Snap*!* is Scheme disguised as Scratch."**

BYOB 3.0 was perhaps 90% of a Scheme implementation, enough to teach a *SICP*-based course, but missing two main capabilities. One, macros, has been *half*-implemented since BYOB 3.0, namely the ability for a custom block to have unevaluated inputs. (The other half, the ability to inject code into the caller of the block, is still to come.) The second big advanced capability is first class continuations.

The *continuation* of a block in a script is the part of the computation that remains to be done when that block has been run. For a command block, this generally means the blocks below it in the script (plus whatever remains of the script that called this block, and so on). For a reporter, it means the block to which this block provides an input, plus whatever comes below it.

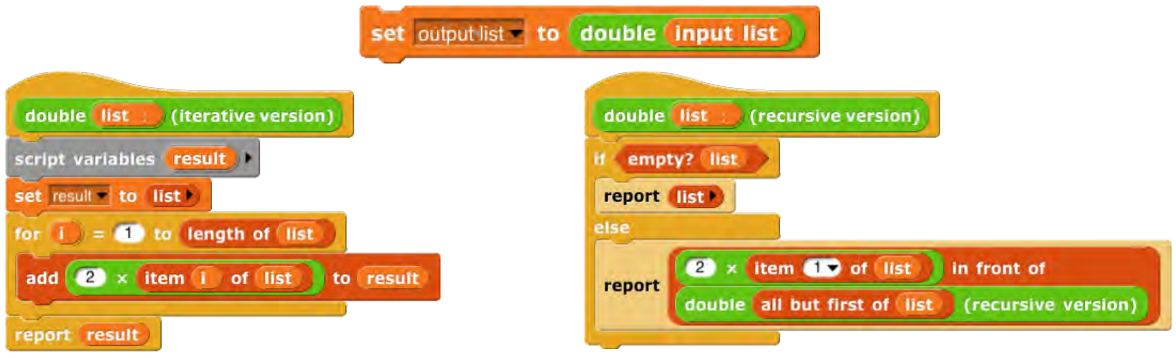
Continuations themselves are not a special feature of Scheme. Any part of a computation, in any language, has a continuation—whatever is left to do after it finishes. But in most languages, the programmer isn't called upon to think of the continuation as a *thing*, although the implementor of the programming language does think about continuations. What's special in Scheme is that it treats continuations as first class data. Giving the programmer access to continuations allows users to write any control structure; the most common examples are nonlocal exit (catch and throw in the Snap*!* tools library) and a thread scheduler.

The feature is embodied in two blocks, run w/continuation and call w/continuation.

Linked lists

The data structure that Scratch (and therefore Snap*!*) calls a "list" is actually implemented as a *dynamic array,* i.e., a contiguous block of memory that is expanded by copying as needed. A *linked list* is a different data structure made out of *pairs;* each pair contains one list item plus a pointer to another pair, so the entire list contains as many pairs as items, and they don't have to be contiguous in memory.

For example, given a list of numbers input list , the goal is to make a list in which every item is two times the corresponding item of the input.



```
set output list to double input list
```

```
double list (iterative version)
script variables result
set result to list
for i = 1 to length of list
    add 2 × item i of list to result
report result
```

```
double list (recursive version)
if empty? list
    report list
else
    report 2 × item 1 of list in front of double all but first of list (recursive version)
```

In the iterative version, the add command block extends the same list by one item each time it's invoked. This kind of list access is fastest with dynamic arrays.

In the recursive version, the in front of block creates a new pair, whose left half is a new item, and whose right half is (a pointer to) the result of the recursive call, namely a sublist of the result. This is fastest with linked lists.

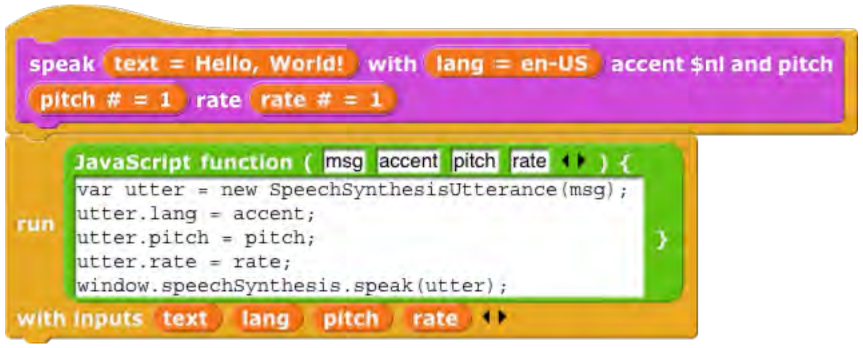
It is a goal of Snap*!* that users not have to know anything about data structures, but should still have their project run as fast as possible. Therefore, Snap*!* automatically creates linked lists when the reporters (item, all but first of, in front of) are used, but creates dynamic arrays when the commands (add, insert, replace, delete) are used.

JavaScript in custom blocks

Do you *still* think you know everything Snap*!* has to teach? In that case, you'll like the

JavaScript function ( ) { }

block. In the following example, never mind right now what it actually does; instead, notice how inputs are passed from the Snap*!* environment to the JavaScript world.



```
speak text = Hello, World! with lang = en-US accent $nl and pitch pitch # = 1 rate rate # = 1
run JavaScript function ( msg accent pitch rate ) {
    var utter = new SpeechSynthesisUtterance(msg);
    utter.lang = accent;
    utter.pitch = pitch;
    utter.rate = rate;
    window.speechSynthesis.speak(utter);
}
with inputs text lang pitch rate
```

This example comes from a text-to-speech library. Many Snap*!* libraries have been written by advanced users; writing libraries is a way you can explore JavaScript programming while remaining in, and contributing to, the Snap*!* community.