

The Snap! Programming System

Bernat Romagosa i Carrasquer

September 18, 2017

1 Introduction

Snap! was developed by Jens Mönig and Brian Harvey under the umbrella of the University of California at Berkeley, with the objective of bringing the power of the Scheme programming language and its computer science concepts into a Scratch-like environment. In fact, for a bunch of years, Snap! was developed as a modified version of Scratch. The ability to construct your own blocks out of other blocks was one of the foundational principles of this modification of Scratch, which is why, initially, they named it BYOB (standing for Build Your Own Blocks). In Snap!, blocks built by the user bear the same weight as any other preexisting ones. In other words, Snap! lets us build custom expressions that look and behave like primitive ones.

Nowadays, Snap! is a separate project with a codebase that is completely independent from Scratch, and with an outstanding number of new features and particularities that make it suitable for a serious study of computer science, as well as for research projects. However, Snap! still keeps intact everything that Scratch is about and, in a programming language jargon style, we could very well call it a dialect of the latter.

So, before diving into what makes Snap! special, let us see what it inherits from Scratch and how it can be used for a playful introduction to programming and computer science.

2 What’s Scratch-Like in Snap!

2.1 The User Interface

In the Snap! user interface, we quickly recognise three major column-shaped areas that resemble those in Scratch. The leftmost one is called the *palette*, and that is where we can find all the primitive expressions, in the shape of blocks. To build our programs, we will be dragging these blocks into the big, central area. This striped gray blank space at the center is called the *scripting area*, and it is where we combine these blocks together to give behavior to objects in the system.

Like in Scratch, this central column features three different tabs. Aside from the default one that we have just described, labeled *Scripts*, we can also choose between *Costumes* and *Sounds*. These two additional tabs let us deal with images that our objects can “wear” as costumes, or sounds that our objects can play.

The third column is a very important one, and it is, in turn, divided into two rows. The white rectangle at the top right is what we call the *Stage*: a programmable micro-world where our objects interact with each other (and with us) according to the code we have composed for them. When you first open the Snap! editor, the Stage only contains one arrowhead-shaped object,

in homage to Logo, where the programmable *Turtle* was also represented as an arrow. We call these objects *sprites*, and we can see how many of them there are, what their names are, and a small thumbnail previewing what they look like at the bottom of the Stage, in an area we call the *corral*. At the center left of the corral, we can see a thumbnail of the Stage itself, that can also be programmed.

2.2 Live, Parallel Blocks

Some blocks-based environments translate graphical blocks into text-based languages that are then fed into an interpreter or compiler. In that sense, these environments do not constitute actual programming languages, but graphical, blocks-based façades for existing text-based languages. In such environments, we very often have to begin our programs with a predefined “start” block, and run them by using some kind of “run” button or instruction.

One of the essential differences between these blocks-based language representations and Scratch (and thus Snap!) is that the latter does not translate blocks into any other language. Scratch and Snap! are full-fledged languages all by themselves, with an evaluator that actually looks up the graphical blocks and runs them in-place.

This lets us build programs that have multiple start points, and even run arbitrary pieces of code (block stacks) at our own will, whenever we want to. Clicking on a block or block stack will run it immediately, and if that block is meant to return a value, it will actually pop up a speech bubble showing that value. Scratch and Snap! are live also in the sense that we can modify code while it runs, without having to stop our scripts and restart them again to see these changes take effect.

Another key particularity of these languages lies in their parallel execution model, allowing us to run many processes at the same time. The evaluator takes care of scheduling the execution of these processes for us and makes sure that -in the vast majority of the cases- we do not have to worry about race conditions, deadlocks or any other quirks of concurrent programming.

2.3 Events

As explained in the previous subsection, processes can be started by just clicking on blocks and block stacks, and it is strongly encouraged to play with the language in that way, as it allows the programmer to experiment with code in an interactive way. However, sometimes we are going to need scripts to be triggered automatically upon given conditions, not just when we decide to click on them.

In languages like Scratch or Snap!, the way to bind scripts to events is to crown them with what we call a *hat block*. What these hat blocks do is to start

whatever script is attached to them when the event in question occurs.

Examples of such events include pressing a key, moving the mouse, clicking on an object, receiving a broadcast message or pressing a particular button in the user interface. Additionally, in Snap! we also have a generic “when” hat block that lets us define arbitrary trigger conditions for a script.

3 What Sets Snap! Apart from Scratch

3.1 First Class Heterogeneous Lists

Snap! takes from the Scheme programming language the idea of first class lists as the only necessary complex data structure, out of which we can build any other data structure that we can imagine. When we say that something is first class we mean that it can be used in the same way as any of the primitive data types in the system, such as a number or a string. This implies that, in Snap!, we can use lists as if they were, for example, numbers.

As such, we can assign lists to variables, use them as parameters to blocks, have functions return them, or broadcast and receive them as messages.

Since lists are first class, and since something that is first class can be used as if it was a simple number, it must be possible to have lists contain other lists. In fact, by embedding lists inside of other lists is how we can compose any imaginable data type.

In order for this to work, though, we also need the language to let us store whatever we want into lists, be it a number, a string, another list, a block or a sprite. That is, lists should be able to hold anything that the system exposes as first class. Not only that, but we need lists to be heterogeneous, meaning that we need to be able to store different data types all mixed together as elements of the same list.

Having these properties we could, for instance, implement binary trees as lists where the first element holds the left branch of each node, the second element holds the root, and the third element holds the right branch.

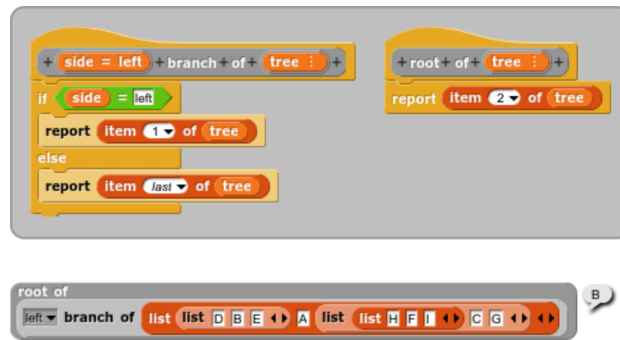


Figure 1: An example of a binary tree implementation. Top: definition of the branch and root accessor blocks. Bottom: example usage with a tree-list.

To make the tree structure easier to understand, we could build a block that hides away the internal list representation. Thanks to Snap!’s support for unicode characters and line breaks in block labels, we could even build a block that resembles a tree branch.

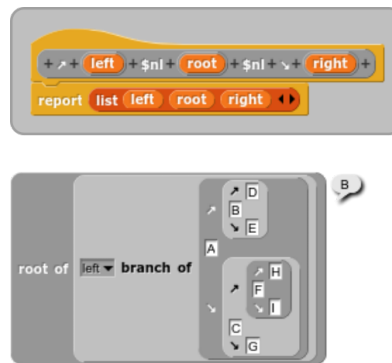


Figure 2: A more visually appealing tree representation. Top: definition of the tree block. Bottom: example usage.

Similarly, a stack could be implemented as a list where we only can only either add a new element or retrieve (and remove) the latest element added.

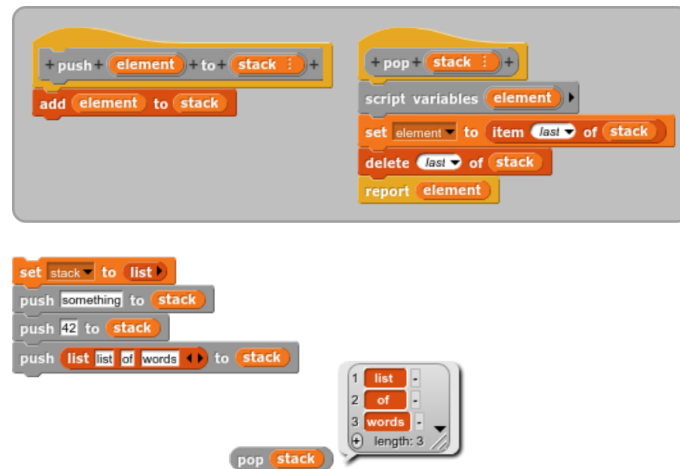


Figure 3: A possible stack implementation. Top: definition of the push and pop blocks. Bottom: creating and filling a stack with some elements, and popping the last one.

Associations can be thought of as lists where the first element is the key and the second element is the value, and thus dictionaries can be thought of as lists of associations.

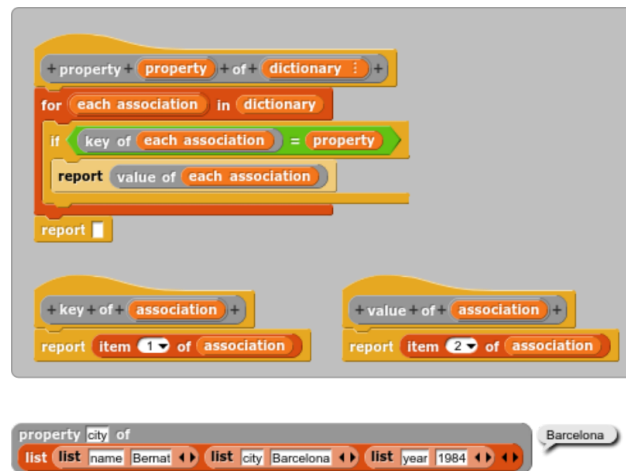


Figure 4: Dictionary data structure. Top: implementation of the key, value and property accessors. Bottom: looking up the property "city" in a dictionary with three associations.

In a similar fashion, by just grouping elements into lists and building custom blocks that abstract away the complexity of these structures, we can implement any other data structure, be it a heap, a hash table or a set.

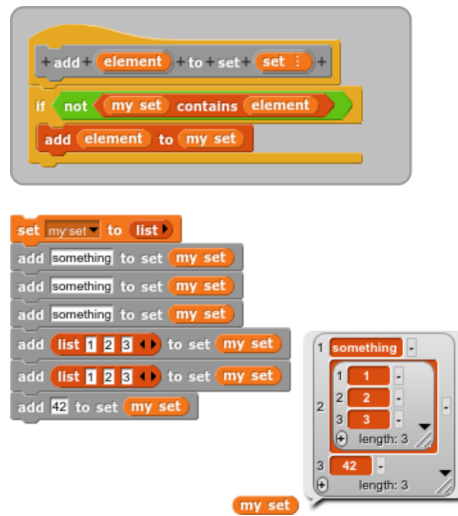


Figure 5: Example set implementation. Top: definition of a block that adds elements to a set. Bottom: trying out the new block and confirming it does not allow any duplicates.

3.1.1 Tables

Snap! offers two different visual representations of lists. The first one is box-shaped and displays all items contained in the list sequentially from top to bottom. This representation reflects changes to the list contents in real time, making it very useful for live inspection of objects and data structures.

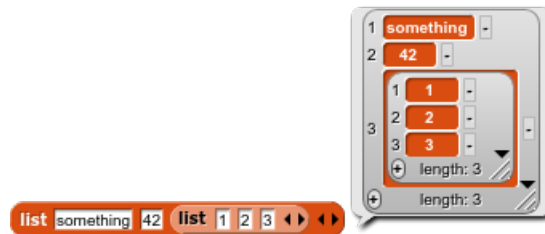


Figure 6: Traditional list representation.

The second representation is the default one for lists of lists where the first inner lists has more than two items. This representation is not live, so changes to the original objects are not reflected in real time, but it is extremely fast and is capable of displaying hundreds of thousands of rows with almost no performance penalty, which makes it suitable for inspecting big data sets and very long collections.

	A	B	C
51096			
29699	13.167	13.515	19.478
29700	13.167	22.760	-8.413
29701	13.167	40.553	-23.381
29702	13.167	47.460	-22.445
29703	13.170	-12.486	22.396
29704	13.171	17.586	0.467
29705	13.175	47.490	-17.131
29706	13.178	-27.679	20.058
29707	13.178	23.012	-8.666
29708	13.179	-38.655	34.414
29709	13.179	7.592	22.348
29710	13.182	27.132	-12.199
29711	13.182	35.833	-14.275
29712	13.183	5.491	22.540
29713	13.185	-19.876	20.893
29714	13.185	17.397	18.020
29715	13.186	-28.474	20.197
29716	13.186	15.073	19.032
29717	13.186	18.752	14.897
29718	13.189	18.790	16.310
29719	13.190	29.371	-13.306

Figure 7: Tabular representation of a huge list of 51.096 rows where each row is a list containing three numbers.

3.2 First Class Procedures

One of the main foundational ideas behind Snap! is to let the programmer extend the system by building blocks that look and behave exactly like the ones provided by default. We have already seen examples of new block definitions in the previous section, when implementing different data structures, but we have not explained them in detail.

To create a new block in Snap! we need to either head to the *Variables* category and click on the *Make a block* button, or right click (command-click on MacOSX) anywhere in the scripting pane and select *make a block...* from the contextual menu.

This will open a dialog where we will be asked to select a category and give a name to the new block. Additionally, we will be able to choose among three different kinds of blocks, namely commands, reporters or predicates.

Command blocks are the ones that can be stacked to other blocks. They are called commands because they perform an action. Reporters, on the other hand, are blocks that can be embedded into input slots of other blocks, and they return values. Predicates are a particular case of reporters, in particular

the kind that return a Boolean value.

As an example, we will be creating a new block that draws a cross, and we will place it under the *Pen* category. Since this block performs an action, it will be a command block, and since we want every sprite in the project to be able to run it, we will select *for all sprites*.



Figure 8: Creating a new *cross* global command block under the *Pen* category.

Accepting this dialog will show another dialog where we can define our block.

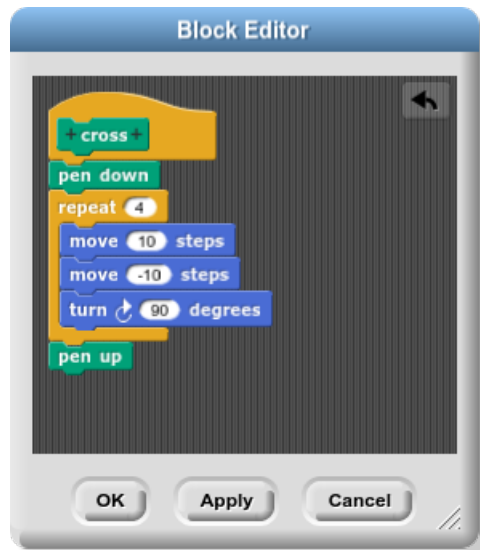


Figure 9: The *cross* block sets the pen down, then moves the sprite in a cross pattern, then sets the pen back up again.

Blocks in Snap! can have parameters, which we can use to modify their behavior accordingly. We could, for instance, add a *length* argument to the *cross* block that defines how long the four arms of the resulting cross drawing are. To do so, we need to edit the block by selecting *edit...* from its contextual menu (right-click on the block for GNU/Linux or Windows users, and Command-click for MacOSX users), then add the missing text and parameters by using the little plus-sign-shaped buttons on the block label.

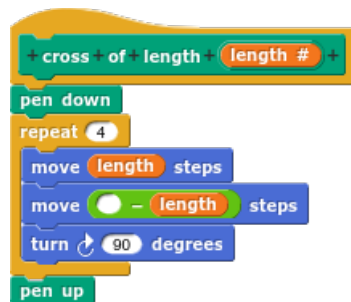


Figure 10: The new *cross* block definition, after having added a new *of length* text part and a *length* parameter to it. The pound symbol next to the parameter indicates that we are expecting it to be a number.

This new block can be now used as if it was any other block in the system, and will be graphically and functionally indistinguishable from the native ones.

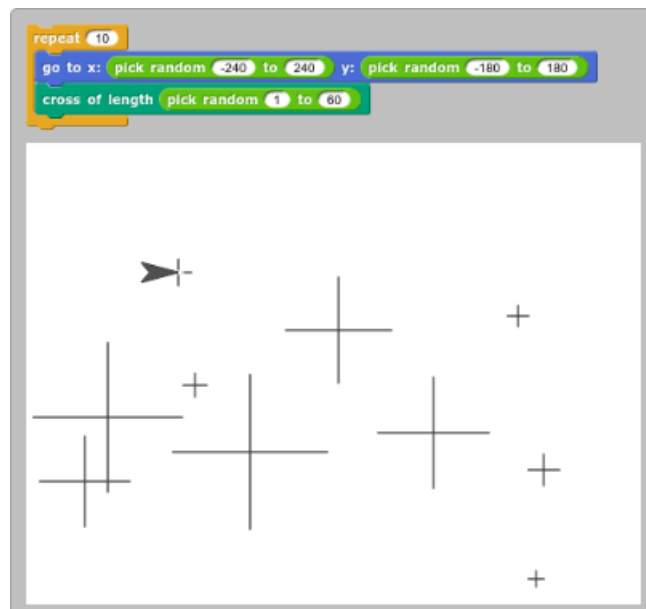


Figure 11: The *cross* block being used to draw crosses of random arm length.

Custom blocks being first class means that we can also use them to define other custom blocks, or even to define themselves recursively. That allows us to

build a new *fractal cross* block that makes use of itself to draw a fractal cross figure, where each arm ends in a cross half the size of the previous one and results in a square pattern.

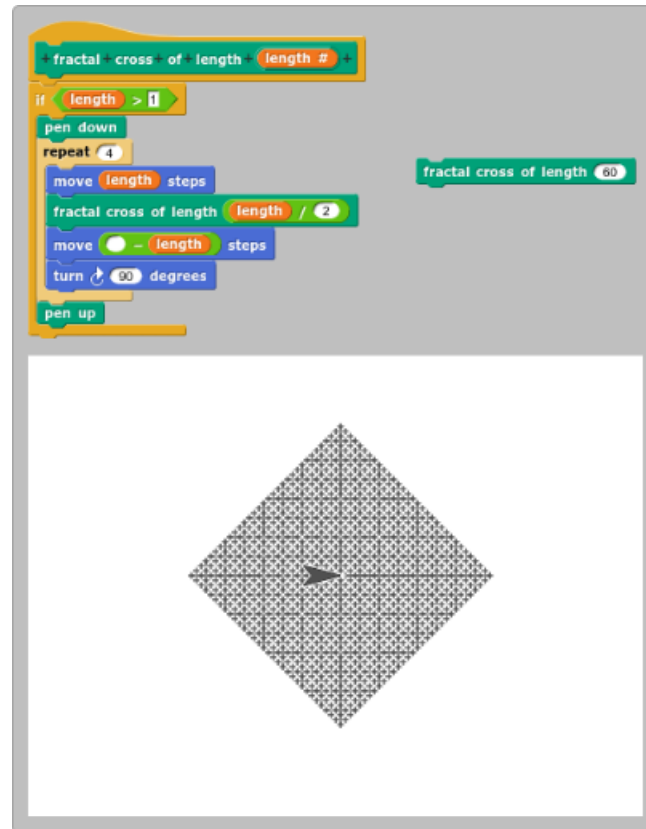


Figure 12: The new *fractal cross* block definition, plus an example of the drawing it generates when used with an arm length of 60.

3.3 Closures

Closures are programming constructs that allow us to deal with code as if it was data. *Full closures* capture the context in which they are created, including the values of any variables in their scope. A full closure is, in a way, like taking a snapshot of a piece of code at a particular moment in time, so that invoking that closure later on will be like running it at the moment of its creation, including the state of the program at that past moment.

This lets us play with code in a substantially different way, even allowing us to define our own control structures. In fact, some computer languages - like Smalltalk- implement all of their control structures by means of closures, without the need of any primitive expressions or reserved keywords.

Snap! represents closures as *ring blocks*. A ring is a special graphical struc-

ture that encloses scripts and turns them into first class data.



Figure 13: A *ringified* addition block.

A *ring* is, actually, a procedure without a name, and clicking on the right-pointing arrow on its right edge will let us add parameters to this procedure.



Figure 14: A *ringified* addition block with a parameter.

But these procedures are special in that clicking on them will not result in them running. As explained before, these constructs convert code into data, so clicking on them will actually just return the data they hold (that is, the closure itself). In other words, if we understand procedures as verbs, then putting a ring around a verb is turning it into a noun.

To run these, we need to use one of the two special blocks Snap! provides. Namely, *run* and *call*. The first one is meant to be used to execute a ring as if it was a command block, whereas the second one will execute it as if it was a reporter. That is, the second one -*call*- will expect the ring to return a value.



Figure 15: Calling the ringified addition block with one parameters and getting back the result the adding 1 to that parameter.

We have just created an anonymous procedure that adds 1 to any number, but up to here it seems that rings only add complexity to something we could have very much easily achieved by just using the addition block by itself, so let us find an example that shows the value of closures a little bit better.

Rings turn any stack of blocks into a parametrizable function that can be treated as data and, as such, can be passed to another procedure as a parameter, in the same way that we can pass a number, a string or a boolean as a parameter. This allows us to, for instance, map the items of a list into a new list where each item has been applied a function, without the need of using any iterators or indexes. From a mathematical point of view, this could be seen as the application of a function over a vector.

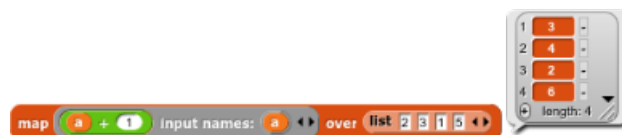


Figure 16: Mapping the *add 1* anonymous procedure to a list, and getting back a new list where each item has been incremented by 1.

As a little syntax sugar goodie, Snap! will automatically fill up any empty input inside the body of a ring, letting us rewrite the previous code in a shorter form with exactly the same functionality.



Figure 17: The same application of map over a list, but this time with automatic parameter filling.

This automatic parameter filling lets us easily map a list into another list where each item maps to its own square.



Figure 18: Using the automatic parameter filling feature to square all numbers in a list.

As a matter of fact, the *map* block is actually implemented in Snap! itself and, internally, it takes a function as a parameter and iterates over a list to apply that function to each of its items.



Figure 19: A simple, iterative implementation of the *map* block.

As a curiosity, even the *for each* block used inside the *map* block has been implemented as a custom block that takes a function as a parameter and runs it for each item in a list.

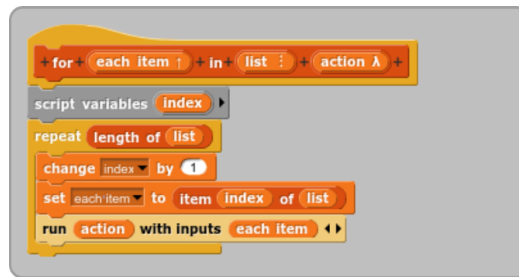


Figure 20: A simple, iterative implementation of the *for each* block.

These functions that take other functions as parameters and apply them to every item on a list are called *Higher Order Functions*, and they represent one of the most useful application of closures in computer science. Indexes and iterations are processes that are much closer to the way computers work than to the way humans think, which is why, in functional programming, *HOFs* are the natural way to transform lists and perform operations over their items.