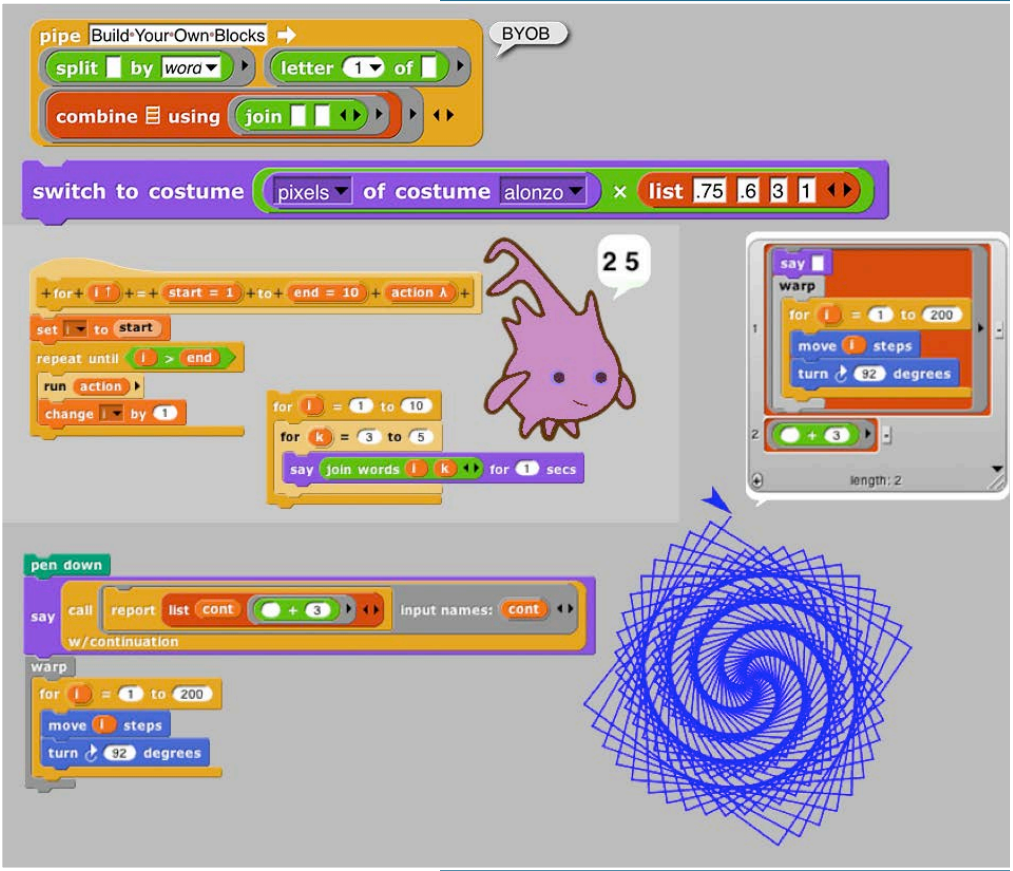
Build Your Own Blocks

6.0

SNAP! Reference Manual



Brian Harvey and Jens Mönig

Table of Contents

Acknowledgements

We have been extremely lucky in our mentors. Jens cut his teeth in the company of the Smalltalk pioneers: Alan Kay, Dan Ingalls, and the rest of the gang who invented personal computing and object oriented programming in the great days of Xerox PARC. He worked with John Maloney, of the MIT Scratch Team, who developed the Morphic graphics framework that's still at the heart of Snap!.

The brilliant design of Scratch, from the Lifelong Kindergarten Group at the MIT Media Lab, is crucial to Snap!. Our earlier version, BYOB, was a direct modification of the Scratch source code. Snap! is a complete rewrite, but its code structure and its user interface remain deeply indebted to Scratch. And the Scratch Team, who could have seen us as rivals, have been entirely supportive and welcoming to us.

Brian grew up at the MIT and Stanford Artificial Intelligence Labs, learning from Lisp inventor John McCarthy, Scheme inventors Gerald J. Sussman and Guy Steele, and the authors of the world's best computer science book, *Structure and Interpretation of Computer Programs,* Hal Abelson and Gerald J. Sussman with Julie Sussman, among many other heroes of computer science. (Brian was also lucky enough, while in high school, to meet Kenneth Iverson, the inventor of APL.)

In the glory days of the MIT Logo Lab, we used to say, "Logo is Lisp disguised as BASIC." Now, with its first class procedures, lexical scope, and first class continuations, Snap! is Scheme disguised as Scratch.

Two people have made such massive contributions to the implementation of Snap! that we have officially declared them members of the team: Michael Ball and Bernat Romagosa. Other close collaborators include Joan Guillén i Pelegay, who has contributed very careful and wise analysis of outstanding issues, including help in taming the management of translations to non-English languages, and Jadga Hügle, a colleague of Jens at SAP, who has energetically contributed to online mini-courses about Snap! and leading workshops for kids and for adults.

We have been fortunate to get to know an amazing group of brilliant middle school(!) and high school students through the Scratch Advanced Topics forum, several of whom have contributed code to Snap!: Kartik Chandra, Nathan Dinsmore, Connor Hudson, Ian Reynolds, and Dylan Servilla. Many more have contributed ideas and alpha-testing bug reports. UC Berkeley students who've contributed code include Michael Ball, Achal Dave. Kyle Hotchkiss, Ivan Motyashov, and Yuan Yuan. Contributors of translations are too numerous to list here, but they're in the "About…" box in Snap! itself.

This material is based upon work supported in part by the National Science Foundation under Grants No. 1138596, 1143566, and 1441075; and in part by MioSoft, Arduino.org, SAP, and YC Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funders.

Snap! Reference Manual

Version 6.0

Snap! (formerly BYOB) is an extended reimplementation of Scratch (http://scratch.mit.edu) that allows you to Build Your Own Blocks. It also features first class lists, first class procedures, first class sprites, first class costumes, first class sounds, and first class continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students.

In this manual we sometimes make reference to Scratch, e.g., to explain how some Snap! feature extends something familiar in Scratch. It's very helpful to have some experience with Scratch before reading this manual, but not essential.

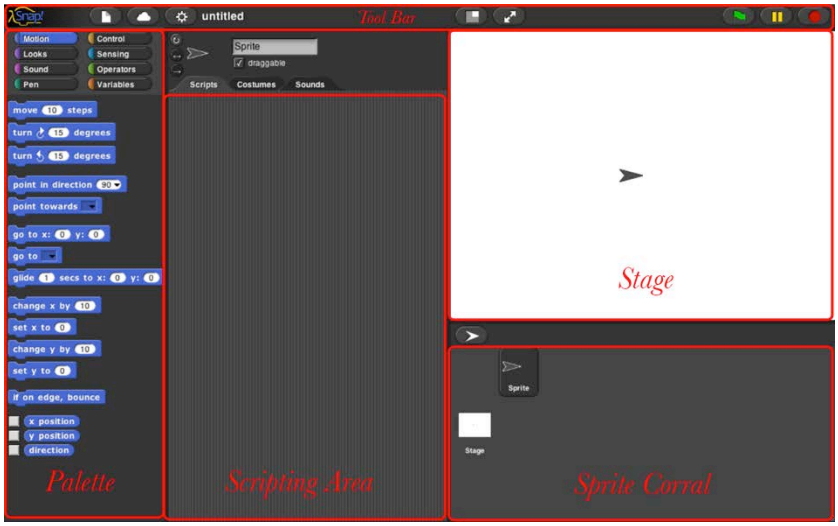
To run Snap!, open a browser window and connect to http://snap.berkeley.edu/run. The Snap! community web site at http://snap.berkeley.edu is not part of this manual's scope.

I. Blocks, Scripts, and Sprites

This section describes the Snap! features inherited from Scratch; experienced Scratch users can skip to subsection B.

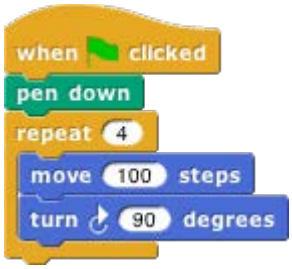
Snap! is a programming language—a notation in which you can tell a computer what you want it to do. Unlike most programming languages, though, Snap! is a *visual* language; instead of writing a program using the keyboard, the Snap! programmer uses the same drag-and-drop interface familiar to computer users.

Start Snap!. You should see the following arrangement of regions in the window:

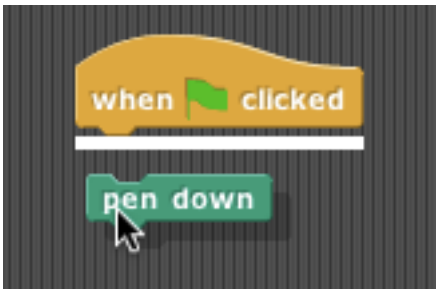


(The proportions of these areas may be different, depending on the size and shape of your browser window.)

A Snap! program consists of one or more *scripts*, each of which is made of *blocks*. Here's a typical script:



The five blocks that make up this script have three different colors, corresponding to three of the eight *palettes* in which blocks can be found. The palette area at the left edge of the window shows one palette at a time, chosen with the eight buttons just above the palette area. In this script, the gold blocks are from the Control palette; the green block is from the Pen palette; and the blue blocks are from the Motion palette. A script is assembled by dragging blocks from a palette into the *scripting area* in the middle part of the window. Blocks snap together (hence the name **Snap!** for the language) when you drag a block so that its indentation is near the tab of the one above it:



The white horizontal line is a signal that if you let go of the green block it will snap into the tab of the gold one.

Hat Blocks and Command Blocks

At the top of the script is a *hat* block, which indicates when the script should be carried out. Hat block names typically start with the word "**when**"; in the square-drawing example on page 4, the script should be run when the green flag near the right end of the **Snap!** tool bar is clicked. (The **Snap!** tool bar is part of the **Snap!** window, not the same as the browser's or operating system's menu bar.) A script isn't required to have a hat block, but if not, then the script will be run only if the user clicks on the script itself. A script can't have more than one hat block, and the hat block can be used only at the top of the script; its distinctive shape is meant to remind you of that.[1]

The other blocks in our example script are *command* blocks. Each command block corresponds to an action that **Snap!** already knows how to carry out. For example, the block 

tells the sprite (the arrowhead shape on the *stage* at the right end of the window) to move ten steps (a step is a very small unit of distance) in the direction in which the arrowhead is pointing. We'll see shortly that there can be more than one sprite, and that each sprite has its own scripts. Also, a sprite doesn't have to look like an arrowhead, but can have any picture as a *costume*. The shape of the **move** block is meant to remind you of a Lego™ brick; a script is a stack of blocks. (The word "block" denotes both the graphical shape on the screen and the procedure, the action, that the block carries out.)

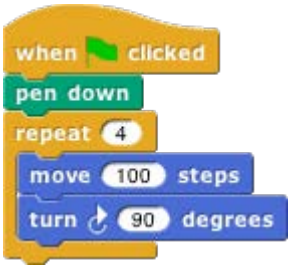
The number 10 in the **move** block above is called an *input* to the block. By clicking on the white oval, you can type any number in place of the 10. The sample script on the previous page uses 100 as the input value. We'll see later that inputs can have non-oval shapes that accept values other than numbers. We'll also see that you can compute input values, instead of typing a particular value into the oval. A block can have more than one input slot. For example, the **glide** block located about halfway down the Motion palette has three inputs.

[1] One of the hat blocks, the generic "when anything" block 

, is subtly different from the others. When the stop sign is clicked, this block no longer tests whether the condition in its hexagonal input slot is true, so the script beneath it will not run, until some *other* script in the project runs (because, for example, you click the green flag).

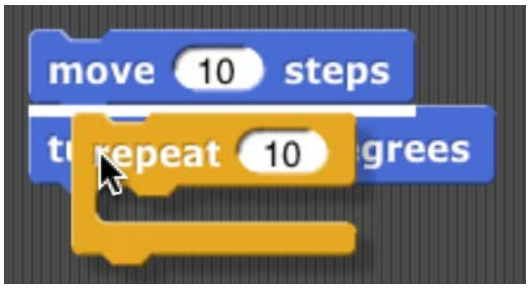
Most command blocks have that brick shape, but some, like the **repeat** block in the sample script, are *C-shaped*. Most C-shaped blocks are found in the Control palette. The slot inside the C shape is a special kind of input slot that accepts a *script* as the input.

In the sample script

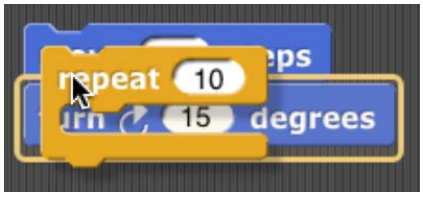


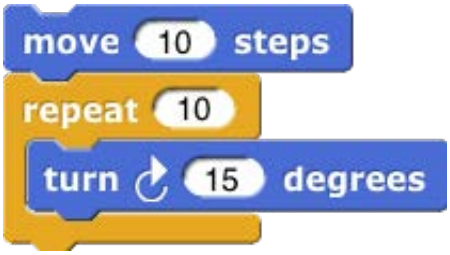
the **repeat** block has two inputs:
the number 4 and the script

C-shaped blocks can be put in a script in two ways. If you see a white line and let go, the block will be inserted into the script like any command block:

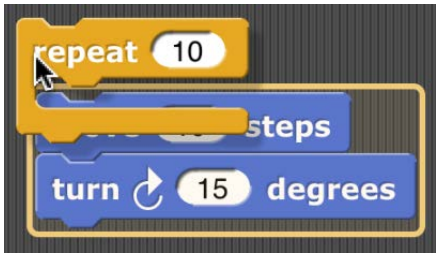


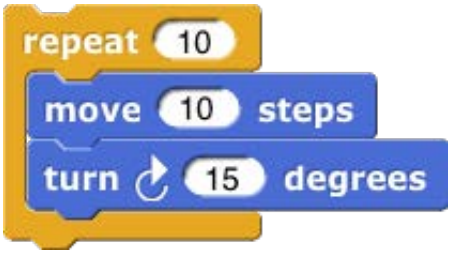
But if you see an orange halo and let go, the block will *wrap* around the haloed blocks:





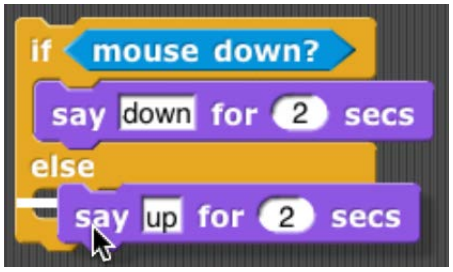
The halo will always extend from the cursor position to the bottom of the script:





If you want only some of those blocks, after wrapping you can grab the first block you don't want wrapped, pull it down, and snap it under the C-shaped block.

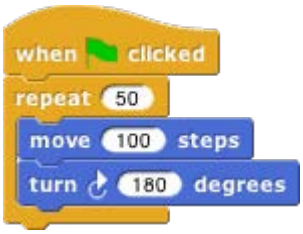
For "E-shaped" blocks with more than one C-shaped slot, only the first slot will wrap around existing blocks in a script, and only if that C-shaped slot is empty before wrapping. (You can fill the other slots by dragging blocks into the desired slot.)



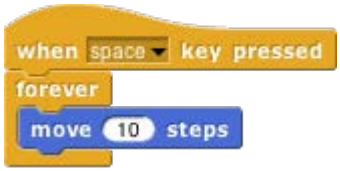
Sprites and Parallelism

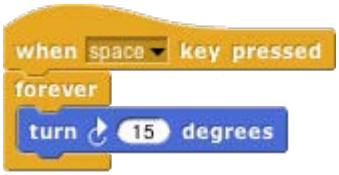
Just below the stage is the "new sprite" button . Click the button to add a new sprite to the stage. The new sprite will appear in a random position on the stage, with a random color, but always facing to the right.

Each sprite has its own scripts. To see the scripts for a particular sprite in the scripting area, click on the picture of that sprite in the *sprite corral* in the bottom right corner of the window. Try putting one of the following scripts in each sprite's scripting area:



When you click the green flag, you should see one sprite rotate while the other moves back and forth. This experiment illustrates the way different scripts can run in parallel. The turning and the moving happen together. Parallelism can be seen with multiple scripts of a single sprite also. Try this example:





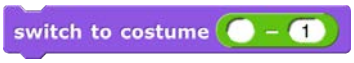
When you press the space key, the sprite should move forever in a circle, because the **move** and **turn** blocks are run in parallel. (To stop the program, click the red stop sign at the right end of the tool bar.)

Costumes and Sounds

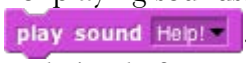
To change the appearance of a sprite, paint or import a new *costume* for it. To paint a costume, click on the **Costumes** tab above the scripting area, and click the paint button . The *Paint Editor* that appears is explained on page 113. There are three ways to import a costume. First select the desired sprite in the sprite corral. Then, one way is to click on the file icon in the tool bar, then choose the "**Costumes...**"menu item. You will see a list of costumes from the public media library, and can choose one. The second way, for a costume stored on your own computer, is to click on the file icon and choose the "**Import...**" menu item. You can then select a file in any picture format (PNG, JPEG, etc.) supported by your browser. The third way is quicker if the file you want is visible on the desktop: Just drag the file onto the Snap! window. In any of these cases, the scripting area will be replaced by something like this:

Just above this part of the window is a set of three tabs: Scripts, Costumes, and Sounds. You'll see that the Costumes tab is now selected. In this view, the sprite's *wardrobe,* you can choose whether the sprite should wear its Turtle costume or its Alonzo costume. (Alonzo, the Snap! mascot, is named after Alonzo Church, a mathematician who invented the idea of procedures as data, the most important way in which Snap! is different from Scratch.) You can give a sprite as many costumes as you like, and then choose which it will wear either by clicking in its wardrobe or by using the 

or 

block in a script. (Every costume has a number as well as a name. The **next costume** block selects the next costume by number; after the highest-numbered costume it switches to costume 1. The Turtle, costume 0, is never chosen by **next costume**.) The Turtle costume is the only one that changes color to match a change in the sprite's pen color. Protip: 

switches to the *previous* costume, wrapping like **next costume**.

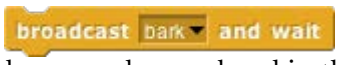
In addition to its costumes, a sprite can have *sounds;* the equivalent for sounds of the sprite's wardrobe is called its *jukebox*. Sound files can be imported in any format (WAV, OGG, MP3, etc.) supported by your browser. Two blocks accomplish the task of playing sounds. If you would like a script to continue running while the sound is playing, use the block 

. In contrast, you can use the 

block to wait for the sound's completion before continuing the rest of the script.

Inter-Sprite Communication with Broadcast

Earlier we saw an example of two sprites moving at the same time. In a more interesting program, though, the sprites on stage will *interact* to tell a story, play a game, etc. Often one sprite will have to tell another sprite to run a script. Here's a simple example:

In the 

block, the word "bark" is just an arbitrary name I made up. When you click on the downward arrowhead in that input slot, one of the choices (the only choice, the first time) is "**new**," which then prompts you to enter a name for the new broadcast. When this block is run, the chosen message is sent to *every* sprite, which is why the block is called "broadcast." In this program, though, only one sprite has a script to run when that broadcast is sent, namely the dog. Because the boy's script uses **broadcast and wait** rather than just **broadcast**, the boy doesn't go on to his next **say** block until the dog's script finishes. That's why the two sprites take turns talking, instead of both talking at once.

Notice, by the way, that the **say** block's first input slot is rectangular rather than oval. This means the input can be any text string, not only a number. In the text input slots, a space character is shown as a brown dot, so that you can count the number of spaces between words, and in particular you can tell the difference between an empty slot and one containing spaces. The brown dots are *not* shown on the stage when the block is run.

The stage has its own scripting area. It can be selected by clicking on the Stage icon at the left of the sprite corral. Unlike a sprite, though, the stage can't move. Instead of costumes, it has *backgrounds:* pictures that fill the entire stage area. The sprites appear in front of the current background. In a complicated project, it's often convenient to use a script in the stage's scripting area as the overall director of the action.

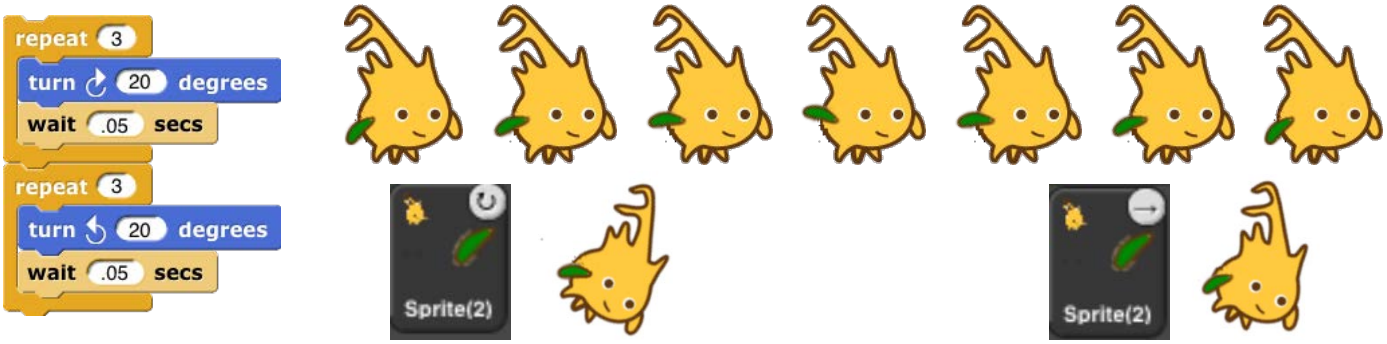
In Section VII, "Object-Oriented Programming with Sprites," you'll see how to send a message to a specific sprite using the **tell** and **ask** blocks, rather than to all sprites as **broadcast** does.

B. 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. Snap! allows one sprite to be designated as the *anchor* of the combined shape, with other sprites as its *parts*. To set up sprite nesting, drag the sprite corral icon of a *part* sprite onto the stage display (not the sprite corral icon!) of the desired *anchor* sprite.

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.) Sprite, representing Alonzo's body, is the anchor; Sprite(2) 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/dangling rotation flag* in the top right corner. In its initial, synchronous 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 circular arrow to a straight arrow, 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.



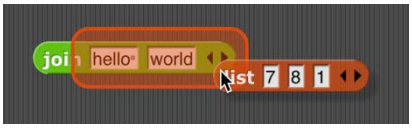
Top: turning the part: the green arm. Bottom: turning the anchor, with the arm synchronous (left) and dangling (right).

C. Reporter Blocks and Expressions

So far, we've used two kinds of blocks: hat blocks and command blocks. Another kind is the *reporter* block, which has an oval shape: (x position). It's called a "reporter" because when it's run, instead of carrying out an action, it reports a value that can be used as an input to another block. If you drag a reporter into the scripting area by itself and click on it, the value it reports will appear in a speech balloon next to the block:

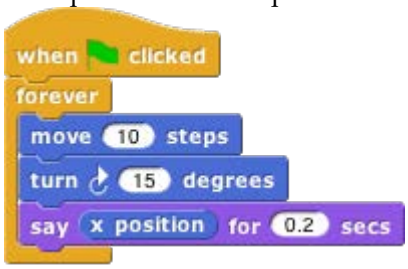
When you drag a reporter block over another block's input slot, a white "halo" appears around that input slot, analogous to the white line that appears when snapping command blocks together:

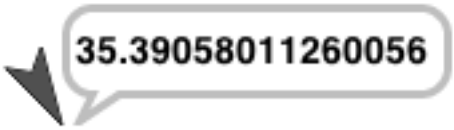
Don't drop the input over a *red* halo:



That's used for a purpose explained on page 52.

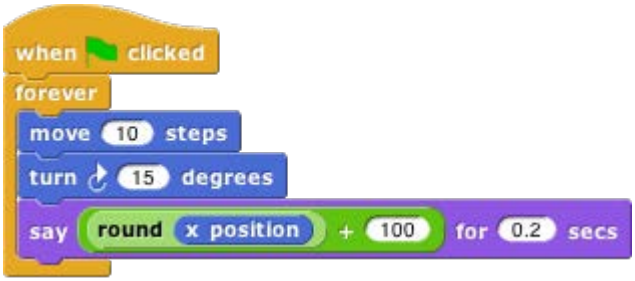
Here's a simple script that uses a reporter block:



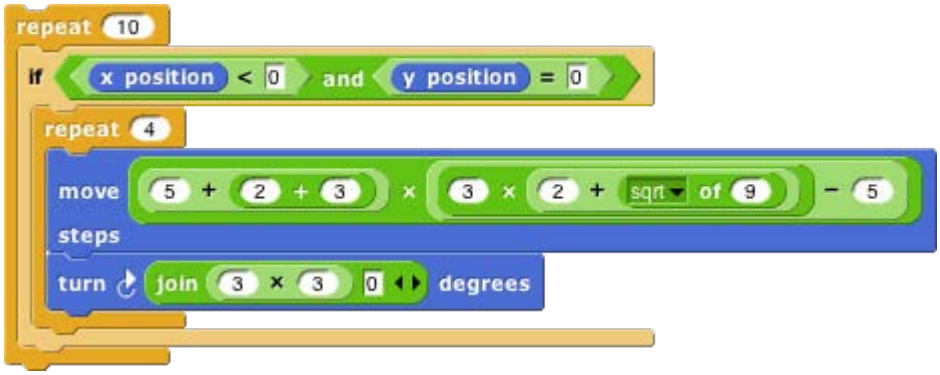


Here the **x position** reporter provides the first input to the **say** block. (The sprite's X position is its horizontal position, how far left (negative values) or right (positive values) it is compared to the center of the stage. Similarly, the Y position is measured vertically, in steps above (positive) or below (negative) the center.)

You can do arithmetic using reporters in the Operators palette:



The **round** block rounds 35.3905… to 35, and the **+** block adds 100 to that. (By the way, the **round** block is in the Operators palette, just like **+**, but in this script it's a lighter color with black lettering because Snap! alternates light and dark versions of the palette colors when a block is nested inside another block from the same palette:

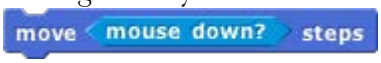


This aid to readability is called *zebra coloring.*) A reporter block with its inputs, maybe including other reporter blocks, such as 

, is called an *expression.*

D. Predicates and Conditional Evaluation

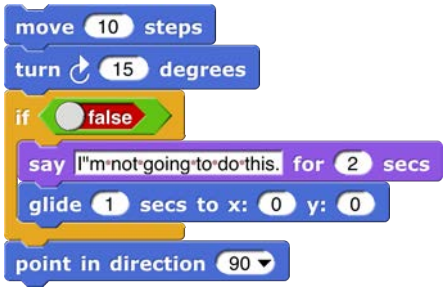
Most reporters report either a number, like  , or a text string, like  . A *predicate* is a special kind of reporter that always reports **true** or **false**. Predicates have a hexagonal shape:

The special shape is a reminder that predicates don't generally make sense in an input slot of blocks that are expecting a number or text. You wouldn't say  , although (as you can see from the picture) Snap! lets you do it if you really want. Instead, you normally use predicates in special hexagonal input slots like this one:

The C-shaped **if** block runs its input script if (and only if) the expression in its hexagonal input reports **true**.

A really useful block in animations runs its input script *repeatedly* until a predicate is satisfied:

If, while working on a project, you want to omit temporarily some commands in a script, but you don't want to forget where they belong, you can say



Sometimes you want to take the same action whether some condition is true or false, but with a different input value. For this purpose you can use the *reporter* **if** block:

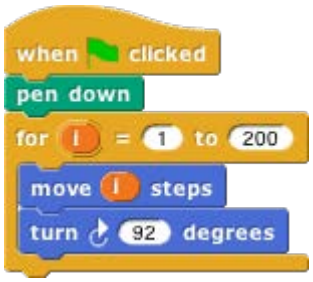
The technical term for a **true** or **false** value is a "Boolean" value; it has a capital B because it's named after a person, George Boole, who developed the mathematical theory of Boolean values. Don't get confused; a hexagonal block is a *predicate,* but the value it reports is a *Boolean.*

Another quibble about vocabulary: Many programming languages reserve the name "procedure" for Commands (that carry out an action) and use the name "function" for Reporters and Predicates. In this manual, a *procedure* is any computational capability, including those that report values and those that don't. Commands, Reporters, and Predicates are all procedures. The words "a Procedure type" are shorthand for "Command type, Reporter type, or Predicate type."

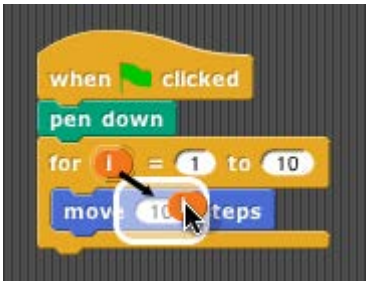
If you want to put a *constant* Boolean value in a hexagonal slot instead of a predicate-based expression, hover the mouse over the block and click on the control that appears: 

E. Variables

Try this script:



The input to the ***move*** block is an orange oval. To get it there, drag the orange oval that's part of the ***for*** block:



The orange oval is a *variable:* a symbol that represents a value. (I took this screenshot before changing the second number input to the ***for*** block from the default 10 to 200, and before dragging in a ***turn*** block.) ***For*** runs its script input repeatedly, just like ***repeat***, but before each repetition it sets the variable **i** to a number starting with its first numeric input, adding 1 for each repetition, until it reaches the second numeric input. In this case, there will be 200 repetitions, first with **i**=1, then with **i**=2, then 3, and so on until **i**=200 for the final repetition. The result is that each ***move*** draws a longer and longer line segment, and that's why the picture you see is a kind of spiral. (If you try again with a turn of 90 degrees instead of 92, you'll see why this picture is called a "squiral.")

The variable **i** is created by the ***for*** block, and it can only be used in the script inside the block's C-slot. (By the way, if you don't like the name **i**, you can change it by clicking on the orange oval without dragging it, which will pop up a dialog window in which you can enter a different name:

"**l**" isn't a very descriptive name; you might prefer "**length**" to indicate its purpose in the script. "**l**" is traditional because mathematicians tend to use letters between **i** and **n** to represent integer values, but in programming languages we don't have to restrict ourselves to single-letter variable names.)

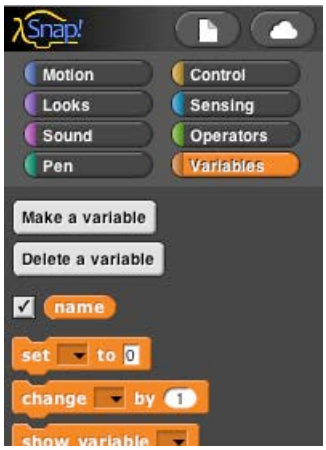
Global Variables

You can create variables "by hand" that aren't limited to being used within a single block. At the top of the Variables palette, click the "***Make a variable***" button:

This will bring up a dialog window in which you can give your variable a name:

The dialog also gives you a choice to make the variable available to all sprites (which is almost always what you want) or to make it visible only in the current sprite. You'd do that if you're going to give several sprites individual variables *with the same name,* so that you can share a script between sprites (by dragging it from the current sprite's scripting area to the picture of another sprite in the sprite corral), and the different sprites will do slightly different things when running that script because each has a different value for that variable name.

If you give your variable the name "***name***" then the Variables palette will look like this:



There's now a "***Delete a variable***" button, and there's an orange oval with the variable name in it, just like the orange oval in the ***for*** block. You can drag the variable into any script in the scripting area. Next to the oval is a checkbox, initially checked. When it's checked, you'll also see a *variable watcher* on the stage:

When you give the variable a value, the orange box in its watcher will display the value.

How *do* you give it a value? You use the **set** block:

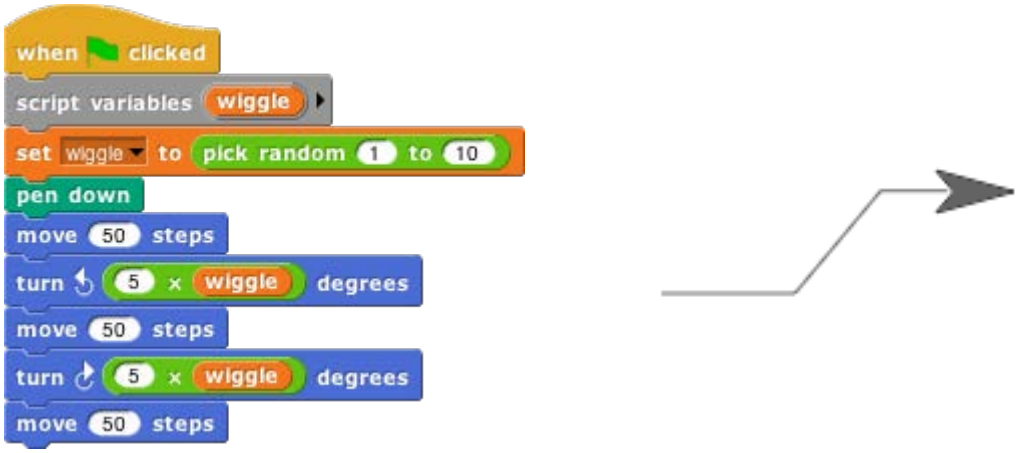
Note that you *don't* drag the variable's oval into the **set** block! You click on the downarrow in the first input slot, and you get a menu of all the available variable names.

If you do choose "For this sprite only" when creating a variable, its block in the palette looks like this: 

The *location*-pin icon is a bit of a pun on a sprite-*local* variable. It's shown only in the palette.

Script Variables

In the name example above, our project is going to carry on an interaction with the user, and we want to remember their name throughout the project. That's a good example of a situation in which a *global* variable (the kind you make with the "**Make a variable**" button) is appropriate. Another common example is a variable called "**score**" in a game project. But sometimes you only need a variable temporarily, during the running of a particular script. In that case you can use the **script variables** block to make the variable:



As in the **for** block, you can click on an orange oval in the **script variables** block without dragging to change its name. You can also make more than one temporary variable by clicking on the right arrow at the end of the block to add another variable oval:

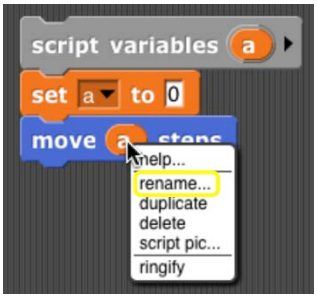
Renaming variables

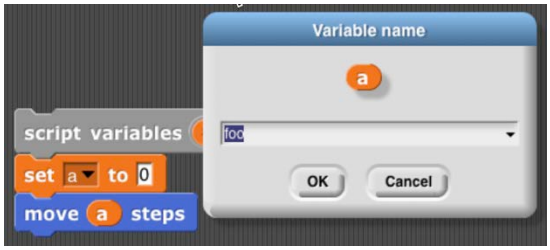
There are several reasons why you might want to change the name of a variable:

1. It has a default name, such as the "**a**" in **script variables** or the "**i**" in **for**.
2. It conflicts with another name, such as a global variable, that you want to use in the same script.
3. You just decide a different name would be more self-documenting.

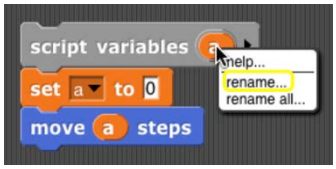
In the first and third case, you probably want to change the name everywhere it appears in that script, or even in all scripts. In the second case, if you've already used both variables in the script before realizing that they have the same name, you'll want to look at each instance separately to decide which ones to rename. Both of these operations are possible by right-clicking or control-clicking on a variable oval.

If you right-click on an orange oval in a context in which the variable is *used,* then you are able to rename just that one orange oval:

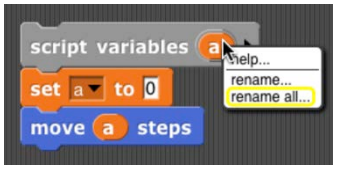




If you right-click on the place where the variable is *defined* (a **script variables** block, the orange oval for a global variable in the Variables palette, or an orange oval that's built into a block such as the "**i**" in **for**), then you are given two renaming options, "rename" and "rename all." If you choose "rename," then the name is changed only in that one orange oval, as in the previous case:

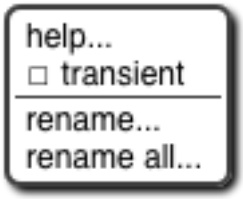


But if you choose "rename all," then the name will be changed throughout the scope of the variable (the script for a script variable, or everywhere for a global variable):



Transient variables

So far we've talked about variables with numeric values, or with short text strings such as someone's name. But there's no limit to the amount of information you can put in a variable; in Chapter IV you'll see how to use *lists* to collect many values in one data structure, and in Chapter VIII you'll see how to read information from web sites. When you use these capabilities, your project may take up a lot of memory in the computer. If you get close to the amount of memory available to **Snap!,** then it may become impossible to save your project. (Extra space is needed to convert from **Snap!**'s internal representation to the form in which projects are exported or saved.) If your program reads a lot of data from the outside world that will still be available when you use it next, you might want to have values containing a lot of data removed from memory before saving the project. To do this, right-click or control-click on the orange oval in the Variables palette, to see this menu:



You already know about the rename options, and **help…** displays a help screen about variables in general. Here we're interested in the check box next to **transient**. If you check it, this variable's value will not be saved when you save your project. Of course, you'll have to ensure that when your project is loaded, it recreates the needed value and sets the variable to it.

F. Debugging

Snap! provides several tools to help you debug a program. They center around the idea of *pausing* the running of a script partway through, so that you can examine the values of variables.

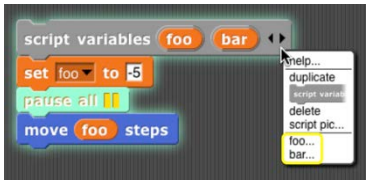
The pause button

The simplest way to pause a program is manually, by clicking the pause button in the top right corner of the window. While the program is paused, you can run other scripts by clicking on them, show variables on stage with the checkbox next to the variable in the Variables palette or with the **show variable** block, and do all the other things you can generally do, including modifying the paused scripts by adding or removing blocks. The button changes shape to and clicking it again resumes the paused scripts.

Breakpoints: the pause all block

The pause button is great if your program seems to be in an infinite loop, but more often you'll want to set a *breakpoint*, a particular point in a script at which you want to pause. The pause all block, near the bottom of the Control palette, can be inserted in a script to pause when it is run. So, for example, if your program is getting an error message in a particular block, you could use **pause all** just before that block to look at the values of variables just before the error happens.

The **pause all** block turns bright cyan while paused. Also, during the pause, you can right-click on a running script and the menu that appears will give you the option to show watchers for temporary variables of the script:



But what if the block with the error is run many times in a loop, and it only errors when a particular condition is true—say, the value of some variable is negative, which shouldn't ever happen. In the iteration library (see page 26 for more about how to use libraries) is a breakpoint block that lets you set a *conditional* breakpoint, and automatically display the relevant variables before pausing. Here's a sample use of it:

(In this contrived example, variable **zot** comes from outside the script but is relevant to its behavior.) When you continue (with the pause button), the temporary variable watchers are removed by this breakpoint block before resuming the script. The breakpoint block isn't magic; you could alternatively just put a **pause all** inside an **if**.[1]

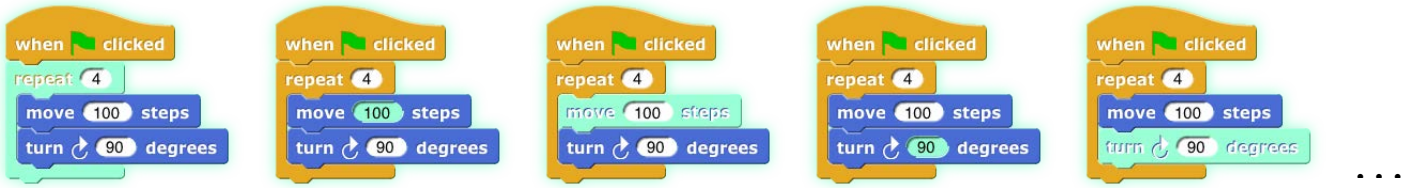
---

[1] The **hide variable** and **show variable** blocks can also be used to hide and show primitives in the palette. The pulldown menu doesn't include primitive blocks, but there's a generally useful technique to give a block input values it wasn't expecting using **run** or **call**:

In order to use a block as an input this way, you must explicitly put a ring around it, by right-clicking on it and choosing **ringify**. More about rings in Chapter VI.

Visible stepping

Sometimes you're not exactly sure where the error is, or you don't understand how the program got there. To understand better, you'd like to watch the program as it runs, at human speed rather than at computer speed. You can do this by clicking the *visible stepping button* (), before running a script or while the script is paused. The button will light up () and a speed control slider will appear in the toolbar. When you start or continue the script, its blocks and input slots will light up cyan one at a time:



In this simple example, the inputs to the blocks are constant values, but if an input were a more complicated expression involving several reporter blocks, each of those would light up as they are called. Note that the input to a block is evaluated before the block itself is called, so, for example, the **100** lights up before the **move**.

The speed of stepping is controlled by the slider. If you move the slider all the way to the left, the speed is zero, the pause button turns into a step button , and the script takes a single step each time you push it. The name for this is *single stepping*.

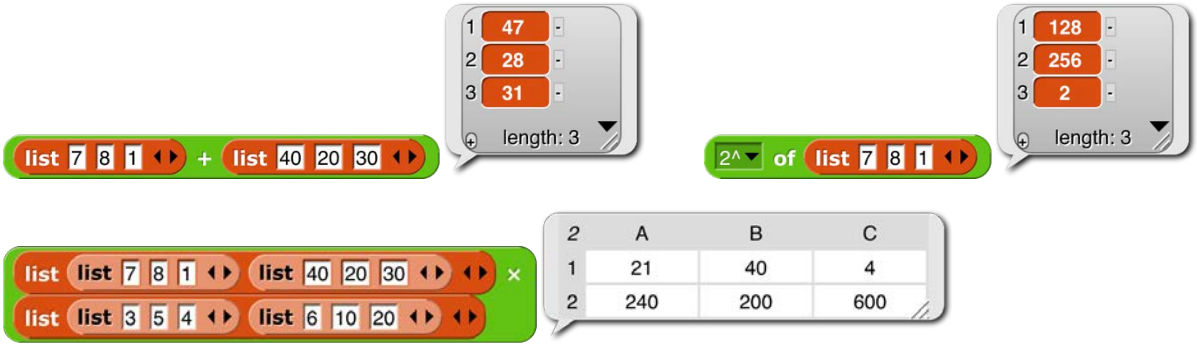
If several scripts that are visible in the scripting area are running at the same time, all of them are stepped in parallel. However, consider the case of two **repeat** loops with different numbers of blocks. While not stepping, each script goes through a complete cycle of its loop in each display cycle, despite the difference in the length of a cycle. In order to ensure that the visible result of a program on the stage is the same when stepped as when not stepped, the shorter script will wait at the bottom of its loop for the longer script to catch up.

When we talk about custom blocks in Chapter III, we'll have more to say about visible stepping as it affects those blocks.

G. Hyperblocks

A *scalar* is anything other than a list. The name comes from mathematics, where it means a magnitude without direction, as opposed to a vector, which points toward somewhere. A scalar function is one whose domain and range are scalars, so all the arithmetic operations are scalar functions, but so are the text ones such as **letter** and the Boolean ones such as **not**.

The major new feature in Snap! 6.0 is that the domain and range of most scalar function blocks is extended to multi-dimensional lists, with the underlying scalar function applied termwise:



(Lists are explained in detail in Section IV, page 34.) Mathematicians, note in the last example above that the result is just a termwise application of the underlying function (7×3, 8×5, etc.), *not* matrix multiplication. For a dyadic (two-input) function, if the lengths don't agree, the length of the result (in each dimension) is the length of the shorter input:

2	A	B	C
1	21	40	
2	240	200	600

However, if the *number of dimensions* differs in the two inputs, then the number of dimensions in the result agrees with the *higher*-dimensional input; the lower-dimensional one is used repeatedly in the missing dimension(s):

3	A	B	C
1	42	80	20
2	240	200	600
3	6	20	

(7×6. 8×10, 1×20, *40×6, 20×10,* etc.). In particular, a *scalar* input is paired with every scalar in the other input:

2	A	B	C
1	l	o	w
2	r	o	d

One important motivation for this feature is how it simplifies and speeds up media computation, as in this shifting of the Alonzo costume to be bluer:

Each pixel of the result has ¾ of its original red and green, and three times its original blue (with its transparency unchanged). By putting some sliders on the stage, you can play with colors dynamically:

There are a few naturally scalar functions that have already had specific meanings when applied to lists and therefore are not hyperblocks: **=** and ***identical to*** (because they compare entire structures, not just scalars, always reporting a single Boolean result), ***and*** and ***or*** (because they don't evaluate their second input at all if the first input determines the result), ***join*** (because it converts non-scalar (and other non-text) inputs to text string form), and ***is a*** (type) (because it applies to its input as a whole). Blocks whose inputs are "natively" lists, such as

length of and in front of, are never hyperblocks. The **item of** block has a special set of rules, designed to preserve its pre-hyperblock meaning and also provide a useful behavior when given a list as its first (index) input:

1. If the index is a number, then **item of** reports the indicated top-level item of the list input; that item may be a sublist, in which case the entire sublist is reported:

2. If the index is a list of numbers (no sublists), then **item of** reports a list of the indicated top-level items (rows, in a matrix):

3. If the index is a one-item list whose item is a list of numbers, then **item of** reports a list of the indicated lowest-level item slices (*columns,* in a matrix); this is often useful with databases in spreadsheet format:

4. If the index is a more-than-one-item list of lists of numbers, then **item of** reports an array of only those scalars whose position in the list input matches the index input in all dimensions, with the lowest-level dimension *first* and the top-level dimension *last:*

5. If a list of list of numbers includes an empty sublist, then all items are chosen along that dimension:

The idea of extending the domain and range of scalar functions to include arrays comes from the language APL. (All the great programming languages are based on mathematical ideas. Our primary ancestors are Smalltalk, based on models, and Lisp, based on lambda calculus. Prolog, a great language not (so far) influencing Snap!, is based on logic. And APL, now joining our family, is based on linear algebra, which studies vectors and matrices. Those *other* programming languages are based on the weaknesses of computer hardware.) Hyperblocks are not the whole story about APL, which also has mixed-domain functions and higher order functions. Some of what's missing is provided in the APL library.

H. Etcetera

This manual doesn't explain every block in detail. There are many more motion blocks, sound blocks, costume and graphics effects blocks, and so on. You can learn what they all do by experimentation, and also by reading the "help screens" that you can get by right-clicking or control-clicking a block and selecting "**help…**" from the menu that appears. If you forget what palette (color) a block is, but you remember at least part of its name, type control-F and enter the name in the text block that appears in the palette area.

Here are the primitive blocks that don't exist in Scratch:

reports a new costume consisting of everything that's drawn on the stage by any sprite. Right-clicking the block in the scripting area gives the option to change it to 

if vector logging is enabled. See page 103.

Print characters in the given point size on the stage, at the sprite's position and in its direction. The sprite moves to the end of the text. (Right, that's not always what you want, but you can save the sprite's position before using it, and sometimes you need to know how big the text turned out to be, in turtle steps.) If the pen is down, the text will be underlined.

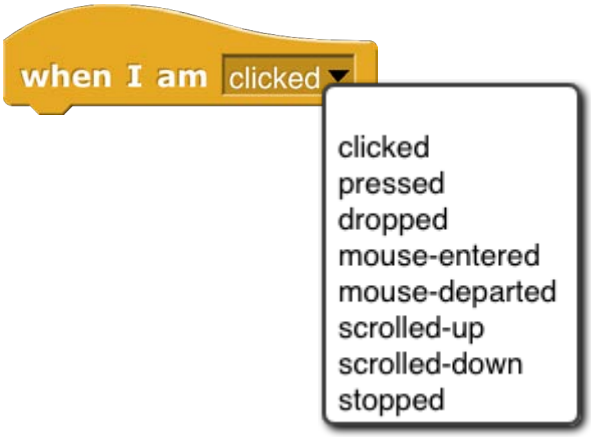
See page 6.

See page 17.

runs only this script until finished.

Reporter version of the **if/else** primitive command block. Only one of the two branches is evaluated, depending on the value of the first input.

Looping block like **repeat** but with an index variable.



Extended mouse interaction events, sensing clicking, dragging, hovering, etc. The "**stopped**" option triggers when all scripts are stopped, as with the stop button; it is useful for robots whose hardware interface must be told to turn off motors. A **when I am stopped** script can run only for a limited time.

Like broadcast, but to a single sprite or the stage.

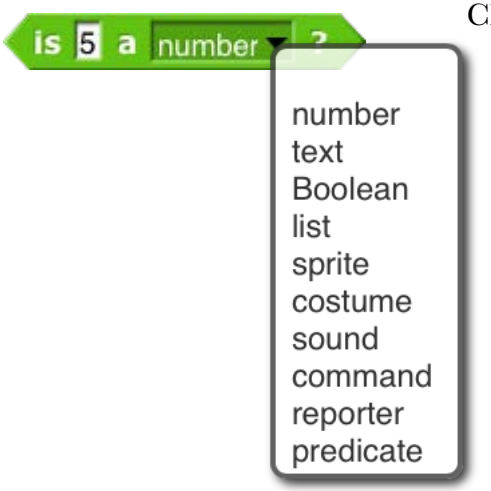
See page 75.

reports the value of a graphics effect.

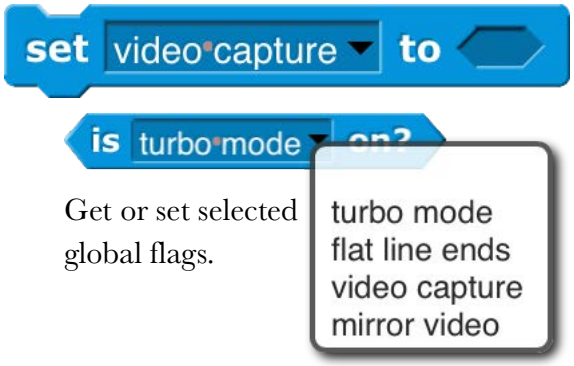
Constant **true** or **false** value. See page 12.

Create a primitive using JavaScript.

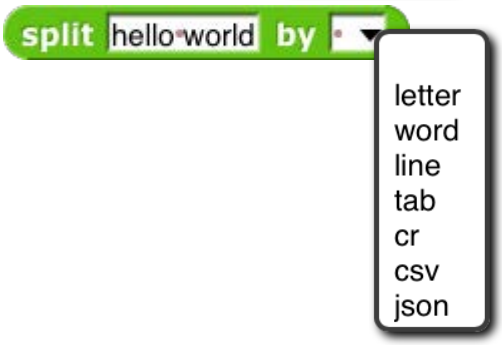
The **at** block lets you examine the screen pixel directly behind the rotation center of a sprite, the mouse, or an arbitrary (x,y) coordinate pair dropped onto the second menu slot. The first four items of the left menu let you examine the color visible at the position. The "**sprites**" option reports a list of all sprites, including this one, any point of which overlaps this sprite's rotation center (behind or in front).



Checks the data type of a value.

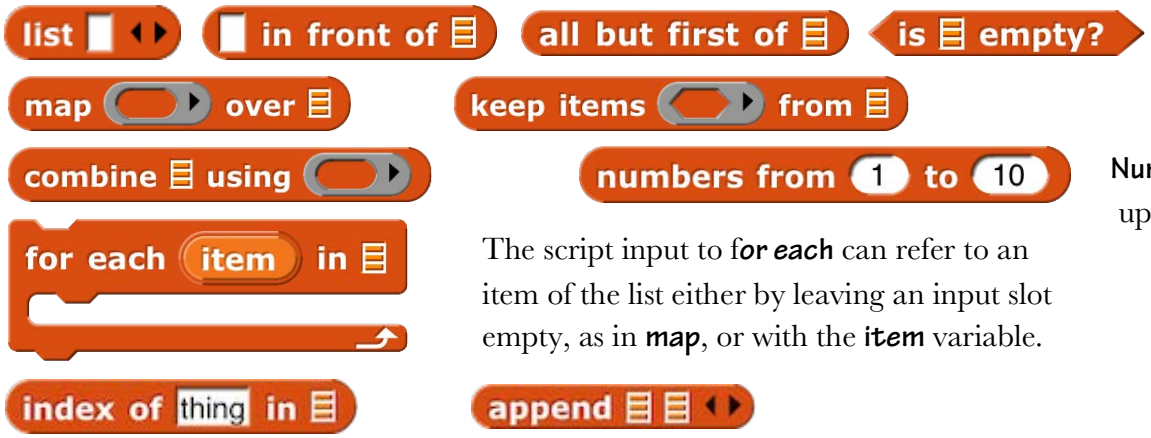


Get or set selected global flags.



Turn the text into a list, using the second input as the delimiter between items. The default delimiter, indicated by the brown dot in the input slot, is a single space character. "**Letter**" puts each character of the text in its own list item. "**Word**" puts each word in an item. (Words are separated by any number of consecutive space, tab, carriage return, or newline characters.) "**Line**" is a newline character (0xa); "**tab**" is a tab character (0x9); "**cr**" is a carriage return (0xd). "**Csv**" and "**json**" split formatted text into lists of lists; see page 41.

First class list blocks (see Section IV, page 34):



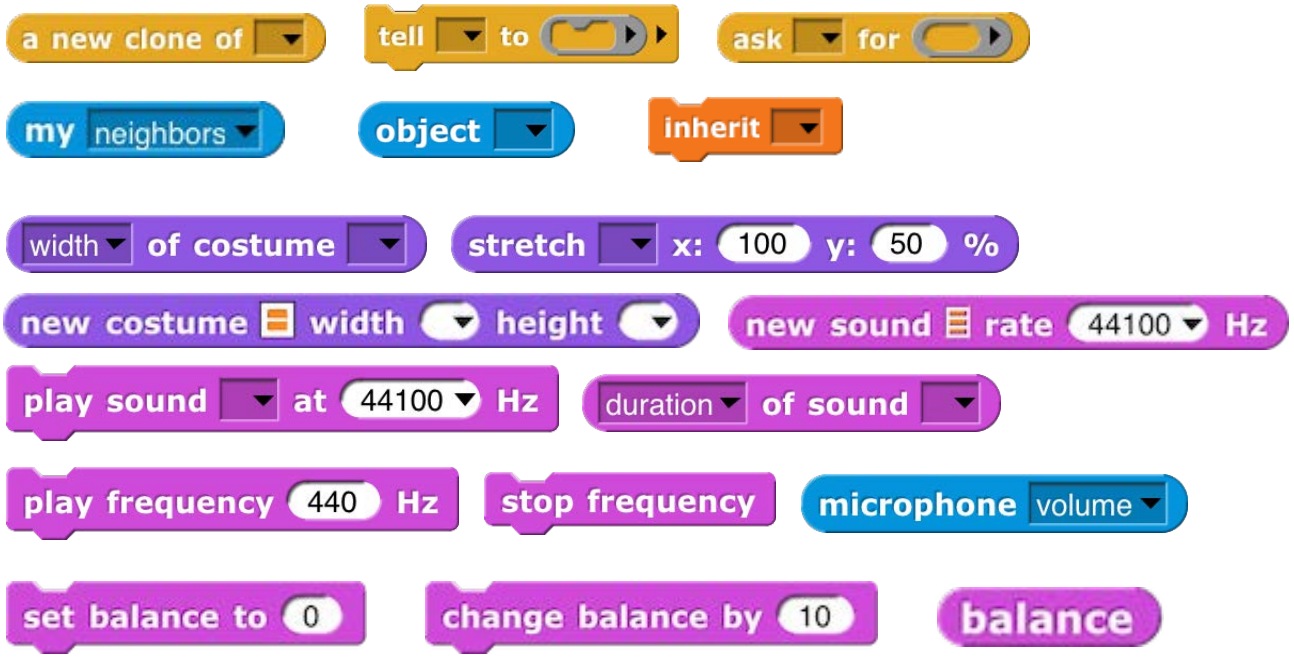
The script input to *for each* can refer to an item of the list either by leaving an input slot empty, as in *map*, or with the *item* variable.

Numbers from will count up or down.

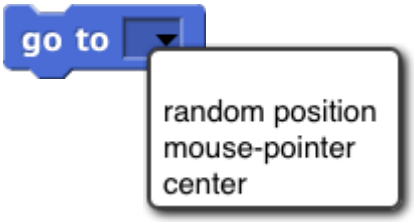
First class procedure blocks (see Section VI, page 49):

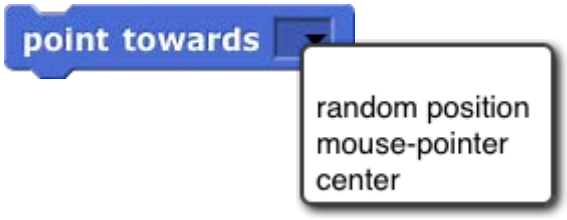
First class continuation blocks (see Section X, page 77):

First class sprite, costume, and sound blocks (see Section VII, page 57):



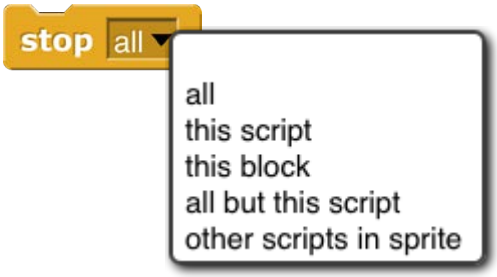
These aren't new blocks but they have a new feature:



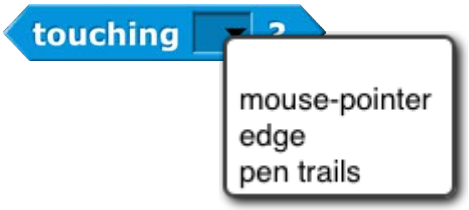


These accept two-item (x,y) lists as input, and have extended menus:

"**Center**" means the center of the stage, the point at coordinates (0,0). "**Direction**" is in the ***point in direction*** sense, the direction that would leave this sprite pointing toward another sprite, the mouse, or the center.

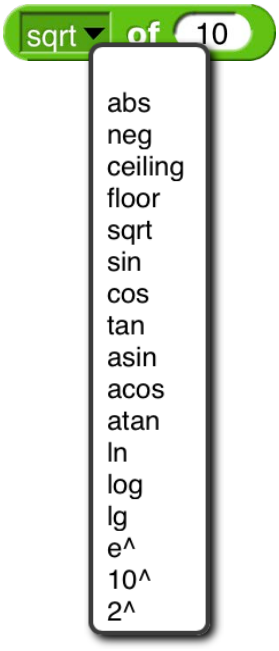


The **stop** block has two extra menu choices. ***Stop this block*** is used inside the definition of a custom block to stop just this invocation of this custom block and continue the script that called it. ***Stop all but this script*** is good at the end of a game to stop all the game pieces from moving around, but keep running this script to provide the user's final score. The last two menu choices add a tab at the bottom of the block because the current script can continue after it.



The new "***pen trails***" option is true if the sprite is touching any drawn or stamped ink on the stage. Also, **touching** will not detect hidden sprites, but a hidden sprite can use it to detect visible sprites.

The ***video*** block has a ***snap*** option that takes a snapshot and reports it as a costume.



The "***neg***" option is a monadic negation operator, equivalent to 

. "***lg***" is $\log_2$.

Blocks only for the Stage:

II. Saving and Loading Projects and Media

After you've created a project, you'll want to save it, so that you can have access to it the next time you use Snap!. There are two ways to do that. You can save a project on your own computer, or you can save it at the Snap! web site. The advantage of saving on the net is that you have access to your project even if you are using a different computer, or a mobile device such as a tablet or smartphone. The advantage of saving on your computer is that you have access to the saved project while on an airplane or otherwise not on the net. Also, cloud projects are limited in size, but you can have all the costumes and sounds you like if you save locally. This is why we have multiple ways to save.

In either case, if you choose "Save as…" from the File menu. You'll see something like this:

(If you are not logged in to your Snap! cloud account, ***Computer*** will be the only usable option.) The text box at the bottom right of the Save dialog allows you to enter project notes that are saved with the project.

A. Local Storage

Click on ***Computer*** and Snap!'s Save Project dialog window will be replaced by your operating system's standard save window. If your project has a name, that name will be the default filename if you don't give a different name. Another, equivalent way to save to disk is to choose "Export project" from the File menu.

B. Creating a Cloud Account

The other possibility is to save your project "in the cloud," at the Snap! web site. In order to do this, you need an account with us. Click on the Cloud button ( ) in the Tool Bar. Choose the "Signup…" option. This will show you a window that looks like the picture at the right.

You must choose a user name that will identify you on the web site, such as ***Jens*** or ***bh***. If you're a Scratch user, you can use your Scratch name for Snap! too. If you're a kid, don't pick a user name that includes your family name, but first names or initials are okay. Don't pick something you'd be embarrassed to have other users (or your parents) see! If the name you want is already taken, you'll have to choose another one. You must also supply a password.

We ask for your month and year of birth; we use this information only to decide whether to ask for your own email address or your parent's email address. (If you're a kid, you

shouldn't sign up for anything on the net, not even Snap!, without your parent's knowledge.) We do not store your birthdate information on our server; it is used on your own computer only during this initial signup. We do not ask for your *exact* birthdate, even for this one-time purpose, because that's an important piece of personally identifiable information.

When you click OK, an email will be sent to the email address you gave, asking you to verify (by clicking a link) that it's really your email address. We keep your email address on file so that, if you forget your password, we can send you a password-reset link. We will also email you if your account is suspended for violation of the Terms of Service. We do not use your address for any other purpose. You will never receive marketing emails of any kind through this site, neither from us nor from third parties. If, nevertheless, you are worried about providing this information, do a web search for "temporary email."

Finally, you must read and agree to the Terms of Service. A quick summary: Don't interfere with anyone else's use of the web site, and don't put copyrighted media or personally identifiable information in projects that you share with other users. And we're not responsible if something goes wrong. (Not that we *expect* anything to go wrong; since Snap! runs in JavaScript in your browser, it is strongly isolated from the rest of your computer. But the lawyers make us say this.)

C. Saving to the Cloud

Once you've created your account, you can log into it using the "Login…" option from the Cloud menu:

Use the user name and password that you set up earlier. If you check the "Stay signed in" box, then you will be logged in automatically the next time you run Snap! from the same browser on the same computer. Check the box if you're using your own computer and you don't share it with siblings. *Don't* check the box if you're using a public computer at the library, at school, etc.

Once logged in, you can choose the "Cloud" option in the "Save Project" dialog shown on page 25. You enter a project name, and optionally project notes; your project will be saved online and can be loaded from anywhere with net access. The project notes will be visible to other users if you publish your project.

D. Loading Saved Projects

Once you've saved a project, you want to be able to load it back into Snap!. There are two ways to do this:

1. If you saved the project in your online Snap! account, choose the "Open…" option from the File menu. Choose the "Cloud" button, then select your project from the list in the big text box and click OK, or choose the "Computer" button to open an operating system open dialog. (A third button, "Examples," lets you choose from example projects that we provide. You can see what each of these projects is about by clicking on it and reading its project notes.)

2. If you saved the project as an XML file on your computer, choose "Import…" from the File menu. This will give you an ordinary browser file-open window, in which you can navigate to the file as you would in other software. Alternatively, find the XML file on your desktop, and just drag it onto the Snap! window.

The second technique above also allows you to import media (costumes and sounds) into a project. Just choose "Import…" and then select a picture or sound file instead of an XML file.

If your project on the cloud is missing, empty, or otherwise broken: In the "Open…" box, enter your project name, then push the "Recover" button. *Do this right away,* because we save only the version before the most recent, and the latest before today. So don't keep saving bad versions; "Recover" right away.

Snap! can also import projects created in BYOB 3.0, or in Scratch 1.4 or (with some effort; see our web site) 2.0. Almost all such projects work correctly in Snap!, apart from a small number of incompatible blocks. BYOB 3.1 projects that don't use first class sprites work, too.

If you saved projects in an earlier version of Snap! using the "Browser" option, then a **Browser** button will be shown in the Open dialog to allow you to retrieve those projects. But you can save them only with the **Computer** and **Cloud** options.

E. Private and Public Projects

By default, a project you save in the cloud is private; only you can see it. There are two ways to make a project available to others. If you **share** a project, you can give your friends a project URL (in your browser's URL bar after you open the project) they can use to read it. If you **publish** a project, it will appear on the Snap! web site, and the whole world can see it. In any case, nobody other than you can ever overwrite your project; if others ask to save it, they get their own copy in their own account.

III. Building a Block

The first version of Snap! was called BYOB, for "Build Your Own Blocks." This was the first and is still the most important capability we added to Scratch. (The name was changed because a few teachers have no sense of humor. ☹ You pick your battles.) Scratch 2.0 and later also has a partial custom block capability.

A. Simple Blocks

In every palette, at or near the bottom, is a button labeled "**Make a block**." Also, floating near the top of the palette is a plus sign.

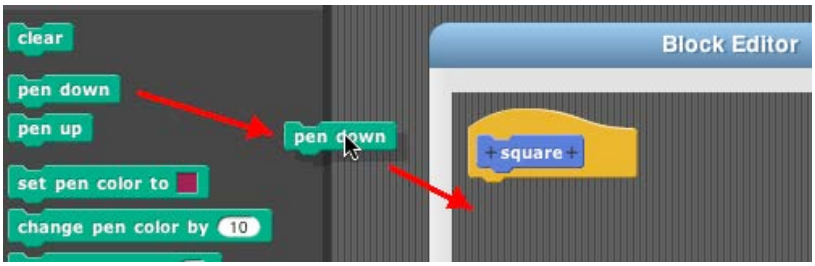
Clicking either one will display a dialog window in which you choose the block's name, shape, and palette/color. You also decide whether the block will be available to all sprites, or only to the current sprite and its children. Note: You can also enter the "Make a block" dialog by right-click/control-click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

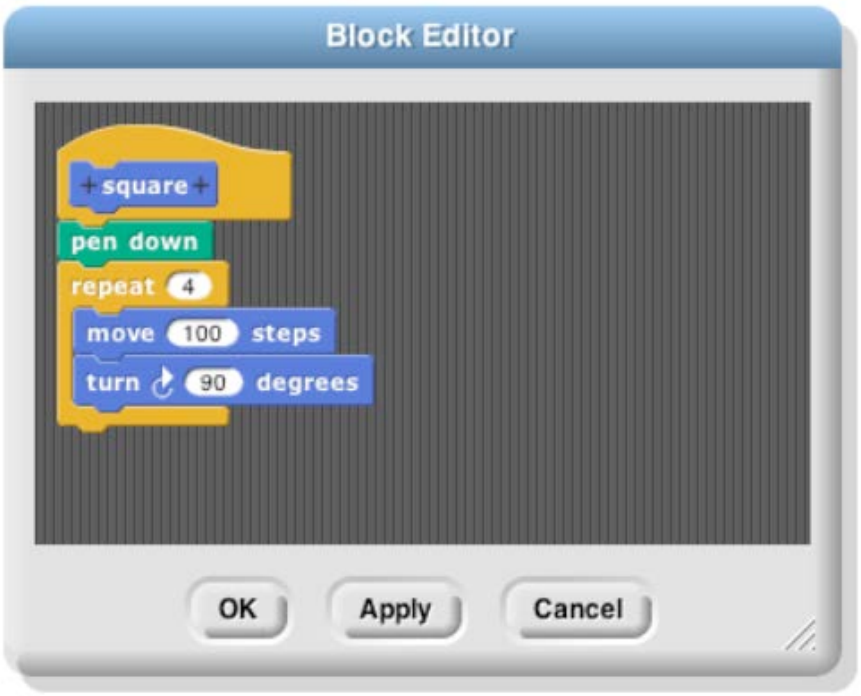
In this dialog box, you can choose the block's palette, shape, and name. With one exception, there is one color per palette, e.g., all Motion blocks are blue. But the Variables palette includes the orange variable-related blocks and the red list-related blocks. Both colors are available, along with an "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

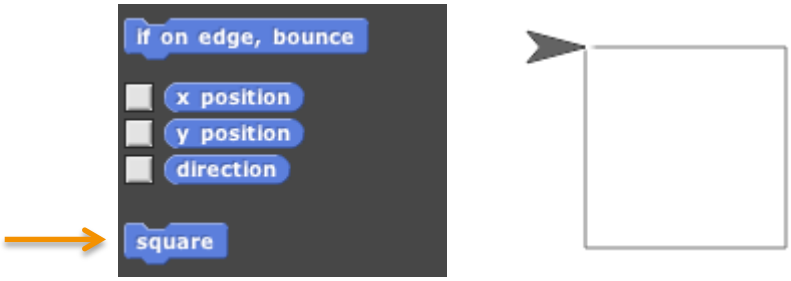
hexagonal blocks are Predicates, which is the technical term for reporters that report Boolean (true or false) values.

Suppose you want to make a block named "square" that draws a square. You would choose Motion, Command, and type "**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 I am 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, then click OK:





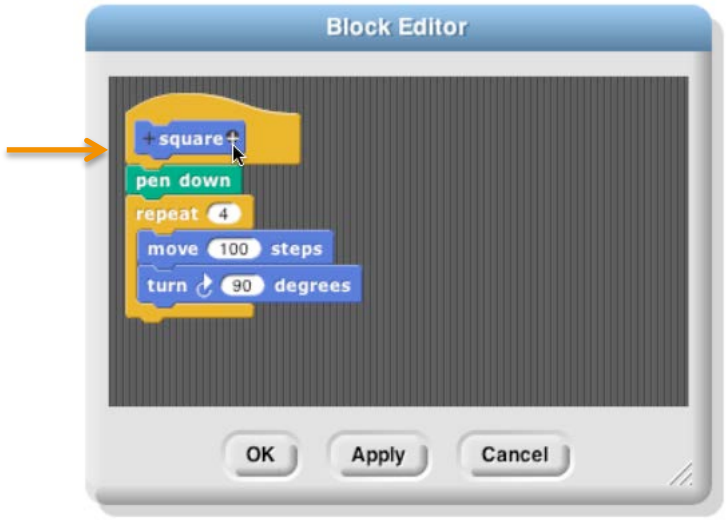
Your block appears at the bottom of the Motion palette. Here's the block and the result of using it:



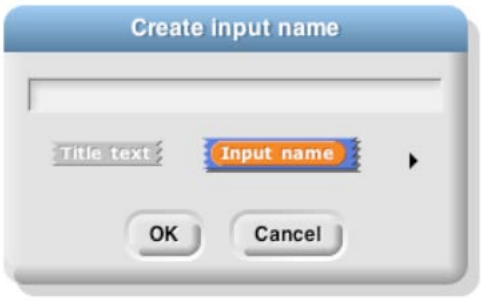
[1] This use of the word "prototype" is unrelated to the *prototyping object oriented programming* discussed later.

Custom Blocks with Inputs

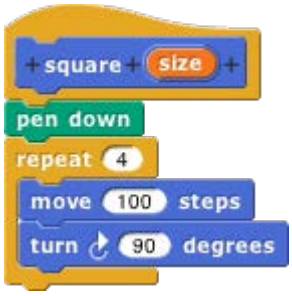
But suppose you want to be able to draw squares of different sizes. Control-click or right-click on the block, choose "**edit**," and the Block Editor will open. Notice the plus signs before and after the word **square** in the prototype block. If you hover the mouse over one, it lights up:



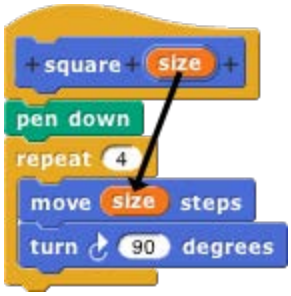
Click on the plus on the right. You will then see the "input name" dialog:



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.

Editing Block Properties

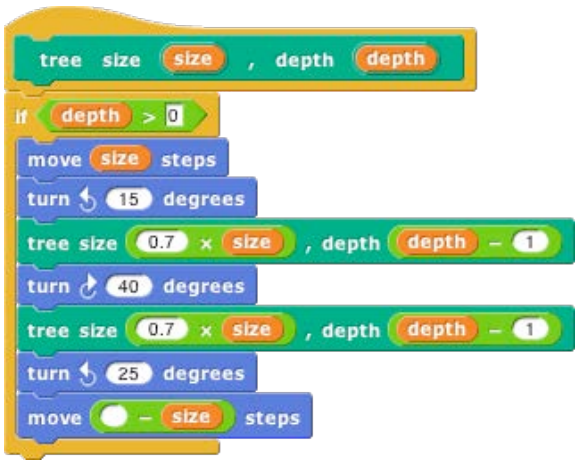
What if you change your mind about a block's color (palette) or shape (command, reporter, predicate)? If you click in the hat block at the top that holds the prototype, but not in the prototype itself, you'll see a window in which you can change the color, and *sometimes* the shape, namely, if the block is not used in any script, whether in a scripting area or in another custom block. (This includes a one-block script consisting of a copy of the new block pulled out of the palette into the scripting area, seeing which made you realize it's the wrong category. Just delete that copy (drag it back to the palette) and then change the category.)

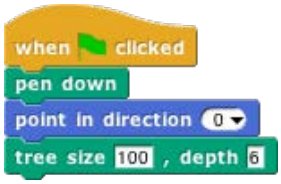
If you right-click/control-click the hat block, you get this menu:

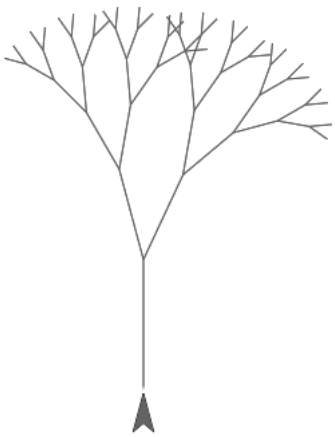
"Script pic" exports a picture of the script. (Many of the illustrations in this manual were made that way.) "Translations" opens a window in which you can specify how your block should be translated if the user chooses a language other than the one in which you are programming. "Block variables" lets you create a variant of script variables for this block: A script variable is created when a block is called, and it disappears when that call finishes. What if you want a variable that's local to this block, as a script variable is, but doesn't disappear between invocations? That's a block variable. If the definition of a block includes a block variable, then every time that (custom) block is dragged from the palette into a script, the block variable is created. Every time *that copy* of the block is called, it uses the same block variable, which preserves its value between calls. Other copies of the block have their own block variables. "Edit" does the same thing as regular clicking, as described in the previous paragraph.

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 you added inputs to the block since opening the editor, click Apply before finding the block in the palette.)

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 example, 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 another tree of size 70, 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 this block** block that has a similar purpose, ending the block invocation early, for command blocks. (By contrast, the **stop this script** block stops not only the current block invocation, but also the entire toplevel script that called it.)

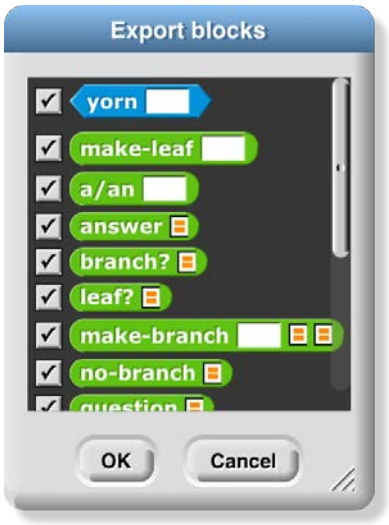
Here's a slightly more compact way to write the factorial function:

For more on recursion, see *Thinking Recursively* by Eric Roberts. (The original edition is ISBN 978-0471816522; a more recent *Thinking Recursively in Java* is ISBN 978-0471701460.)

C. Block Libraries

When you save a project (see Section II above), any custom blocks you've made are saved with it. But sometimes you'd like to save a collection of blocks that you expect to be useful in more than one project. Perhaps your blocks implement a particular data structure (a stack, or a dictionary, etc.), or they're the framework for building a multilevel game. Such a collection of blocks is called a *block library.*

To create a block library, choose "Export blocks…" from the File menu. You then see a window like this:



The window shows all of your global custom blocks. You can uncheck some of the checkboxes to select exactly which blocks you want to include in your library. (You can right-click or control-click on the export window for a menu that lets you check or uncheck all the boxes at once.) Then press OK. An XML file containing the blocks will appear in your Downloads location.

To import a block library, use the "Import…" command in the File menu, or just drag the XML file into the Snap! window.

Several block libraries are included with Snap!; for details about them, see page 89.

D. Custom blocks and Visible Stepping

Visible stepping normally treats a call to a custom block as a single step. If you want to see stepping inside a custom block you must take these steps *in order:*

1. Turn on Visible Stepping.
2. Select "**Edit**" in the context menu(s) of the block(s) you want to examine.
3. Then start the program.

The Block Editor windows you open in step 2 do not have true editing capability. You can tell because there is only one "OK" button at the bottom, not the usual three buttons. Use the button to close these windows when done stepping.

IV. 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, numbers and text strings are first class. You can put a number in a variable, use one as the input to a block, call 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 Snap! 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 Snap! users themselves.

Note that it's a data *type* that's first class, not an individual value. Don't think, for example, that some lists are first class, while others aren't. In Snap!, lists are first class, period.

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. Shift-clicking changes by three at a time.

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

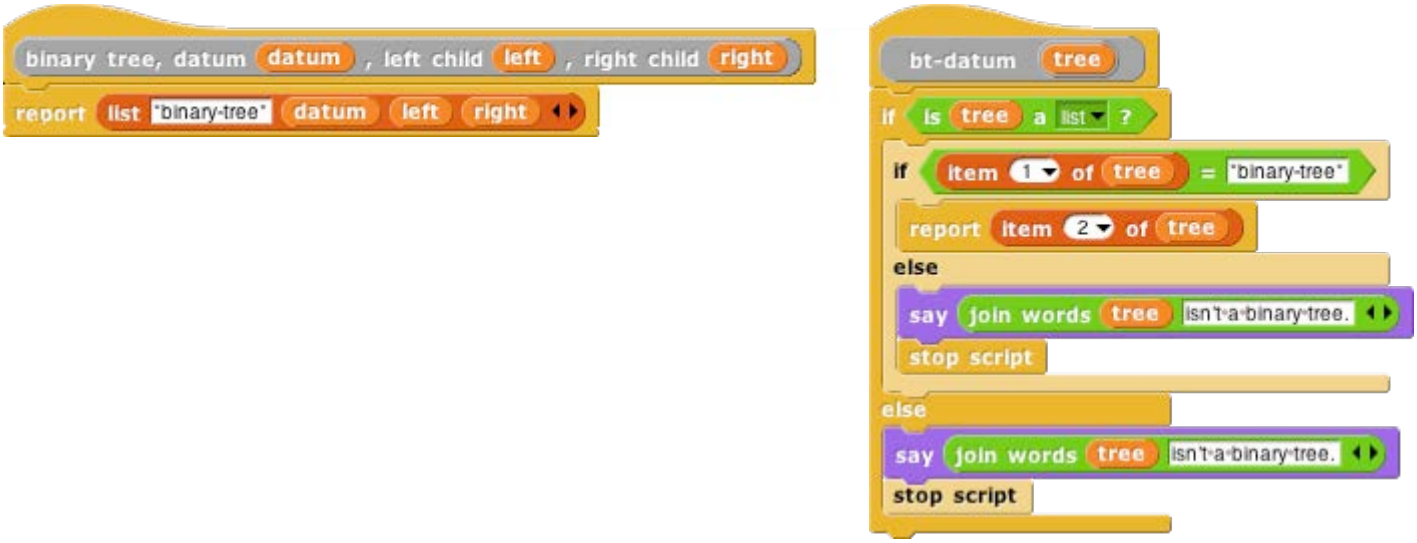
Snap! does not have a "Make a list" button like the one in Scratch. If you want a global "named list," make a global variable and use the **set** block to put a list into the variable.

B. Lists of Lists

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

Notice that this list is presented in a different format from the "She Loves You" list above. A two-dimensional list is called a *table* and is by default shown in *table view*. We'll have more to say about this later.

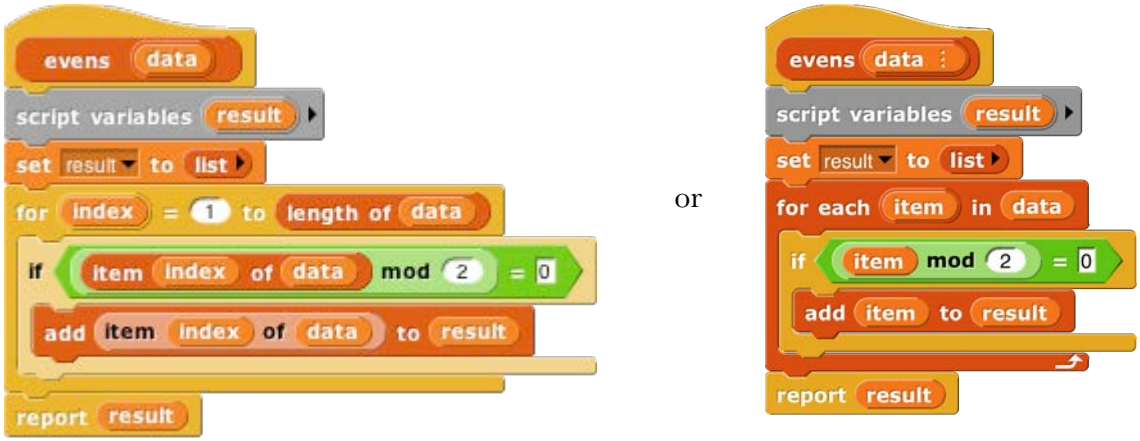
We can also build any classic computer science data structure out of lists of lists, by defining *constructors* (blocks to make an instance of the structure), *selectors* (blocks to pull out a piece of the structure), and *mutators* (blocks to change the contents of the structure) as needed. Here we create binary trees with selectors that check for input of the correct data type; only one selector is shown but the ones for left and right children are analogous.



C. Functional and Imperative List Programming

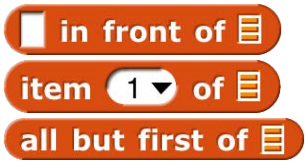
There are two ways to create a list inside a program. Scratch users will be familiar with the *imperative* programming style, which is based on a set of command blocks that modify a list:

As an example, here are two blocks that take a list of numbers as input, and report a new list containing only the even numbers from the original list:[1]



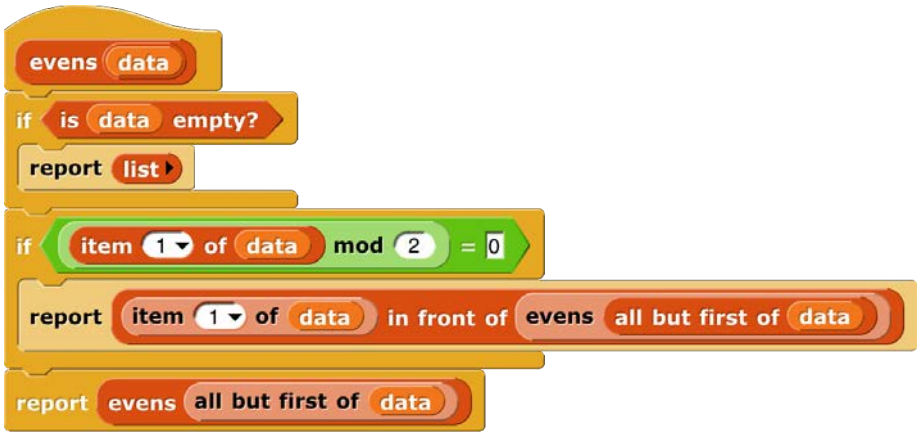
In this script, we first create a temporary variable, then put an empty list in it, then go through the items of the input list using the **add ... to (result)** block to modify the result list, adding one item at a time, and finally report the result.

Functional programming is a different approach that is becoming important in "real world" programming because of parallelism, i.e., the fact that different processors can be manipulating the same data at the same time. This makes the use of mutation (changing the value associated with a variable, or the items of a list) problematic because with parallelism it's impossible to know the exact sequence of events, so the result of mutation may not be what the programmer expected. Even without parallelism, though, functional programming is sometimes a simpler and more effective technique, especially when dealing with recursively defined data structures. It uses reporter blocks, not command blocks, to build up a list value:



[1] Note to users of earlier versions: From the beginning, there has been a tension in our work between the desire to provide tools such as **for** (used in this example) and the higher order functions introduced on the next page as primitives, to be used as easily as other primitives, and the desire to show how readily such tools can be implemented in Snap! itself. This is one instance of our general pedagogic understanding that learners should both use abstractions and be permitted to see beneath the abstraction barrier. Until now, we have used the uneasy compromise of a library of tools written in Snap! and easily, but not easily enough, loaded into a project. By *not* loading the tools, users or teachers could explore how to program them. In 5.0 we made them true primitives, partly because that's what some of us wanted all along and partly because of the increasing importance of fast performance as we explore "big data" and media computation. But this is not the end of the story for us. In a later version, after we get the design firmed up, we intend to introduce "hybrid" primitives, implemented in high speed Javascript but with an "Edit" option that will open, not the primitive implementation, but the version written in Snap!. The trick is to ensure that this can be done without dramatically slowing users' projects.

In a functional program, we often use recursion to construct a list, one item at a time. The **in front of** block makes a list that has one item added to the front of an existing list, *without changing the value of the original list.* A nonempty list is processed by dividing it into its first item (**item 1 of**) and all the rest of the items (**all but first of**), which are handled through a recursive call:



Snap! uses two different internal representations of lists, one (dynamic array) for imperative programming and the other (linked list) for functional programming. Each representation makes the corresponding built-in list blocks (commands or reporters, respectively) most efficient. It's possible to mix styles in the same program, but if *the same list* is used both ways, the program will run more slowly because it converts from one representation to the other repeatedly. (The **item ( ) of [ ]** block doesn't change the representation.) You don't have to know the details of the internal representations, but it's worthwhile to use each list in a consistent way.

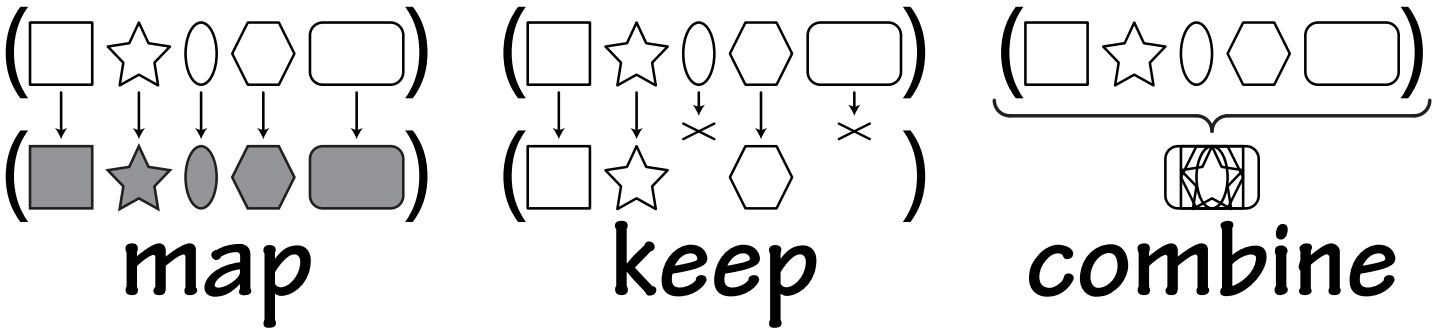
D. Higher Order List Operations and Rings

There's an even easier way to select the even numbers from a list:

The **keep** block takes a Predicate expression as its first input, and a list as its second input. It reports a list containing those elements of the input list for which the predicate returns **true**. Notice two things about the predicate input: First, it has a grey ring around it. Second, the **mod** block has an empty input. **Keep** puts each item of its input list, one at a time, into that empty input before evaluating the predicate. (The empty input is supposed to remind you of the "box" notation for variables in elementary school: □+3=7.) The grey ring is part of the **keep** block as it appears in the palette:

What the ring means is that this input is a block (a predicate block, in this case, because the interior of the ring is a hexagon), rather than the value reported by that block. Here's the difference:

Evaluating the **=** block without a ring reports **true** or **false**; evaluating the block *with* a ring reports the block itself. This allows **keep** to evaluate the **=** predicate repeatedly, once for each list item. A block that takes another block as input is called a *higher order* block (or higher order procedure, or higher order function).



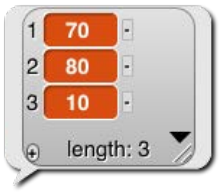
Snap! provides four higher order function blocks for operating on lists:

You've already seen **keep**. **Find first** is similar, but it reports just the first item that satisfies the predicate, not a list of all the matching items. It's equivalent to 

but faster because it stops looking as soon as it finds a match. If there are no matching items, it returns 

(not the empty list because a list of lists is more common than a list of Booleans, so the empty list could be a value matching a predicate).

Map takes a Reporter block and a list as inputs. It reports a new list in which each item is the value reported by the Reporter block as applied to one item from the input list. That's a mouthful, but an example will make its meaning clear:



By the way, we've been using arithmetic examples, but the list items can be of any type, and any reporter can be used. We'll make the plurals of some words:

These examples use small lists, to fit the page, but the higher order blocks work for any size list.

An *empty* gray ring represents the *identity function,* which just reports its input. Leaving the ring in **map** empty is "the most concise way to make a copy of a list.

The third higher order block, ***combine***, computes a single result from *all* the items of a list, using a *two-input* reporter as its second input. In practice, there are only a few blocks you'll ever use with ***combine***:

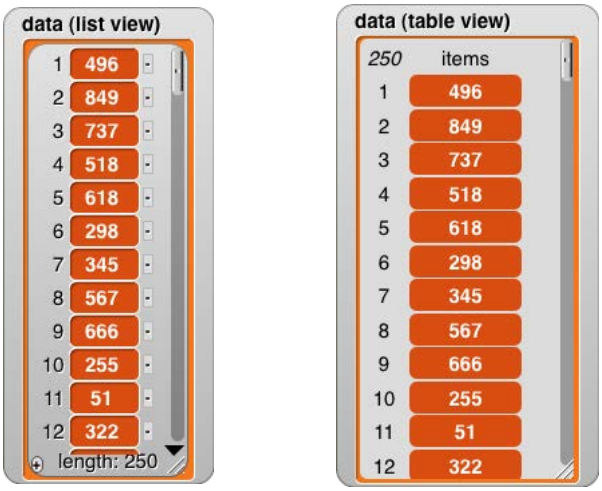
These blocks take the sum of the list items, take their product, string them into one word, combine them into a sentence (with spaces between items), see if all items of a list of Booleans are true, or see if any of the items is true.

Why + but not –? It only makes sense to combine list items using an *associative* function: one that doesn't care in what order the items are combined (left to right or right to left). (2+3)+4 = 2+(3+4), but (2−3)−4 ≠ 2−(3−4).

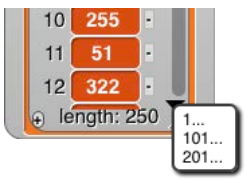
The functions **map**, **keep**, and **find first** have an advanced mode with rarely-used features: If their function input is given explicit input names (by clicking the arrowhead at the right end of the gray ring; see page 53), then it will be called for each list item with *three* inputs: the item (as usual), the item's position in the input list (its index), and the entire input list. No more than three input names can be used in this context.

E. Table View vs. List View

We mentioned earlier that there are two ways of representing lists visually. For one-dimensional lists (lists whose items are not themselves lists) the visual differences are small:



For one-dimensional lists, it's not really the appearance that's important. What matters is that the *list view* allows very versatile direct manipulation of the list through the picture: you can edit the individual items, you can delete items by clicking the tiny buttons next to each item, and you can add new items at the end by clicking the tiny plus sign in the lower left corner. (You can just barely see that the item deletion buttons have minus signs in them.) Even if you have several watchers for the same list, all of them will be updated when you change anything. On the other hand, this versatility comes at an efficiency cost; a list view watcher for a long list would be way too slow. As a partial workaround, the list view can only contain 100 items at a time; the downward-pointing arrowhead opens a menu in which you can choose which 100 to display.



By contrast, because it doesn't allow direct editing, the *table view* watcher can hold thousands of items and still scroll through them efficiently. The table view has flatter graphics for the items to remind you that they're not clickable to edit the values.

Right-clicking on a list watcher (in either form) gives you the option to switch to the other form. The right-click menu also offers an ***open in dialog…*** option that opens an *offstage* table view watcher, because the watchers can take up a lot of stage space that may make it hard to see what your program is actually doing. Once the offstage dialog box is open, you can close the stage watcher. There's an OK button on the offstage dialog to close it if you want. Or you can right-click it to make *another* offstage watcher, which is useful if you want to watch two parts of the list at once by having each watcher scrolled to a different place.

Table view is the default if the list has more than 100 items, or if the first item of the list is a list, in which case it makes a very different-looking two-dimensional picture:

In this format, the column of red items has been replaced by a spreadsheet-looking display. For short, wide lists, this display makes the content of the list very clear. A vertical display, with much of the space taken up by the "machinery" at the bottom of each sublist, would make it hard to show all the text at once. (The pedagogic cost is that the structure is no longer explicit; we can't tell just by looking that this is a list of row-lists, rather than a list of column-lists or a primitive two-dimensional array type. But you can choose list view to see the structure.)

Beyond such simple cases, in which every item of the main list is a list of the same length, it's important to keep in mind that the design of table view has to satisfy two goals, not always in agreement: (1) a visually compelling display of two-dimensional arrays, and (2) highly efficient display generation, so that Snap! can handle very large lists, since "big data" is an important topic of study. To meet the first goal perfectly in the case of "ragged right" arrays in which sublists can have different lengths, Snap! would scan the entire list to find the maximum width before displaying anything, but that would violate the second goal.

Snap! uses the simplest possible compromise between the two goals: It examines only the first ten items of the list to decide on the format. If none of those are a list, or they're all lists of one item, and the overall length is no more than 100, list view is used. If the any of first ten items is a list, then table view is used, and the number of columns in the table is equal to the largest number of items among the first ten items (sublists) of the main list.

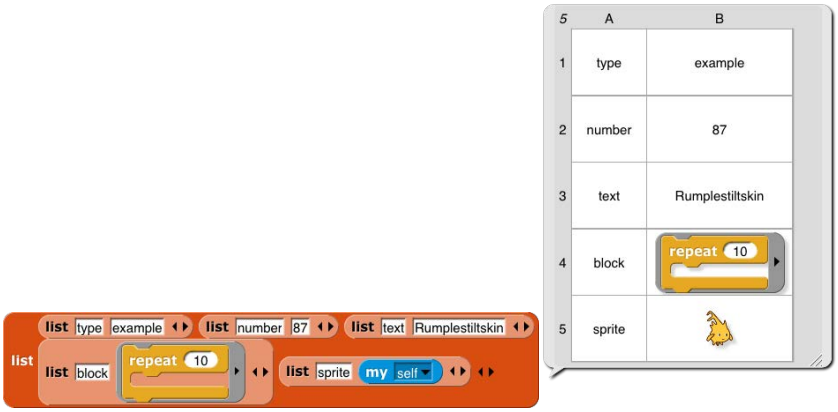
Table views open with standard values for the width and height of a cell, regardless of the actual data. You can change these values by dragging the column letters or row numbers. Each column has its own width, but changing the height of a row changes the height for all rows. (This distinction is based not on the semantics of rows vs. columns, but on the fact that a constant row height makes scrolling through a large list more efficient.) Shift-dragging a column label will change the width of all columns.

If you tried out the adjustments in the previous paragraph, you may have noticed that a column letter turns into a number when you hover over it. Labeling rows and columns differently makes cell references such as "cell 4B" unambiguous; you don't have to have a convention about whether to say the row first or the column first. ("Cell B4" is the same as "cell 4B.") On the other hand, to extract a value from column B in your

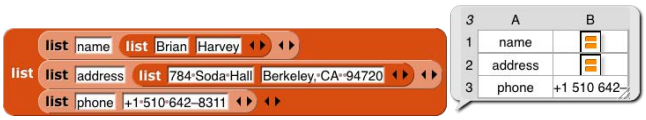
program, you have to say **item 2 of**, not **item B of**. So it's useful to be able to find out a column number by hovering over its letter.

Any value that can appear in a program can be displayed in a table cell:

This display shows that the standard cell dimensions may not be enough for large value images. By expanding the entire speech balloon and then the second column and all the rows, we can make the result fit:

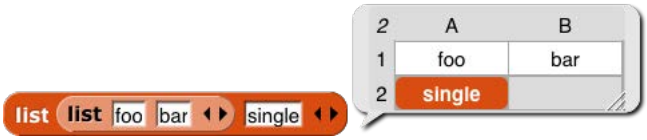


But we make an exception for cases in which the value in a cell is a list (so that the entire table is three-dimensional). Because lists are visually very big, we don't try to fit the entire value in a cell:



Even if you expand the size of the cells, Snap! will not display sublists of sublists in table view. There are two ways to see these inner sublists: You can switch to list view, or you can double-click on a list icon in the table to open a dialog box showing just that sub-sub-list in table view.

One last detail: If the first item of a list is a list (so table view is used), but a later item *isn't* a list, that later item will be displayed on a red background, like an item of a single-column list:



So, in particular, if only the first item is a list, the display will look almost like a one-column display.

Comma-Separated Values

Spreadsheet and database programs generally offer the option to export their data as CSV (comma-separated values lists. You can import these files into Snap! and turn them into tables (lists of lists), and you can export tables in CSV format. Snap! recognizes as CSV file by the extension **.csv** in its filename.

A CSV file has one line per table row, with the fields separated by commas within a row:

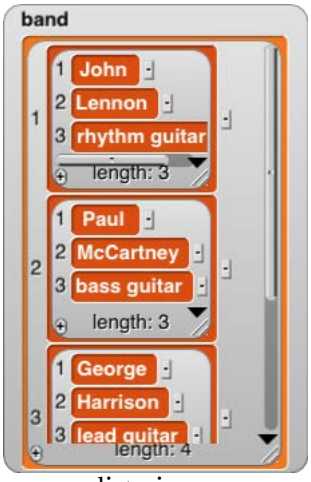
```
John,Lennon,rhythm guitar
Paul,McCartney,bass guitar
George,Harrison,lead guitar
Ringo,Starr,drums
```

Here's what the corresponding table looks like:

band

4	A	B	C
1	John	Lennon	rhythm guitar
2	Paul	McCartney	bass guitar
3	George	Harrison	lead guitar
4	Ringo	Starr	drums

table view



list view

Here's how to read a spreadsheet into Snap!:

1. Make a variable with a watcher on stage: 

2. Right-click on the watcher and choose the "**import**" option. (If the variable's value is already a list, be sure to click on the outside border of the watcher; there is a different menu if you click on the list itself.) Select the file with your csv data.

3. There is no 3; that's it! Snap! will notice that the name of the file you're importing is something.**csv** and will turn the text into a list of lists automatically.

Or, even easier, just drag and drop the file from your desktop onto the Snap! window, and Snap! will automatically create a variable named after the file and import the data into it.

If you actually want to import the raw CSV data into a variable, either change the file extension to **.txt** before loading it, or choose "**raw data**" instead of "**import**" in the watcher menu.

If you want to export a list, put a variable watcher containing the list on the stage, right-click its border, and choose "**Export**." (Don't right-click an item instead of the border; that gives a different menu.)

Multi-dimensional lists and JSON

CSV format is easy to read, but works only for one- or two-dimensional lists. If you have a list of lists of lists, Snap! will instead export your list as a JSON (JavaScript Object Notation) file. I modified my list:

and then exported again, getting this file:

```
[["John","Lennon","rhythm guitar"],[["James","Paul"],"McCartney","bass
guitar"],["George","Harrison","lead guitar"],["Ringo","Starr","drums"]]
```

You can also import lists, including tables, from a **.json** file. (And you can import plain text from a **.txt** file.) Drag and drop works for these formats also.

V. Typed Inputs

A. Scratch's Type Notation

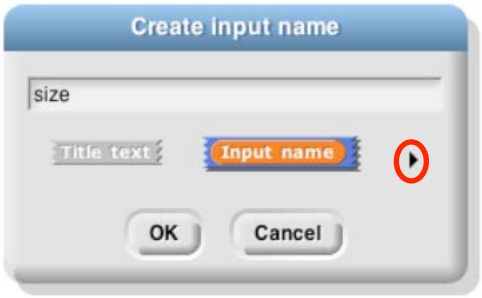
Scratch block inputs come in two types: Text-or-number type and Number type. The former is indicated by a rectangular box, the latter by a rounded box: 

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

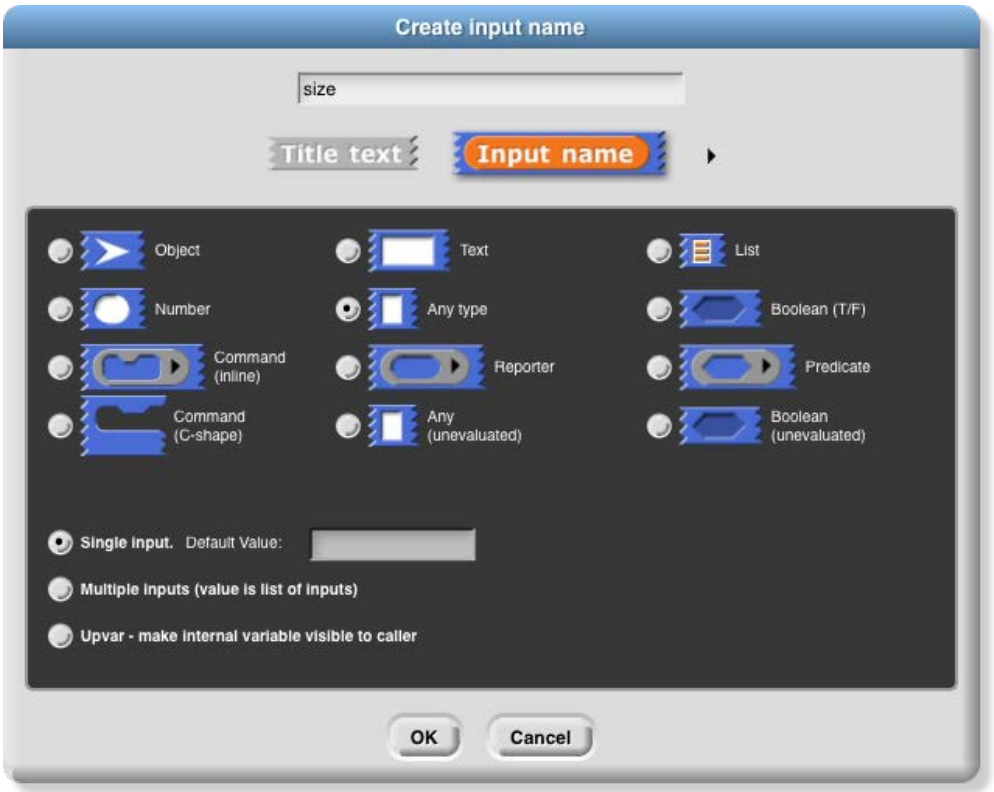
The Snap! types are an expanded collection including Procedure, List, and Object types. Note that, with the exception of Procedure types, all of the input type shapes are just reminders to the user of what the block expects; they are not enforced by the language.

B. The Snap! Input Type Dialog

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



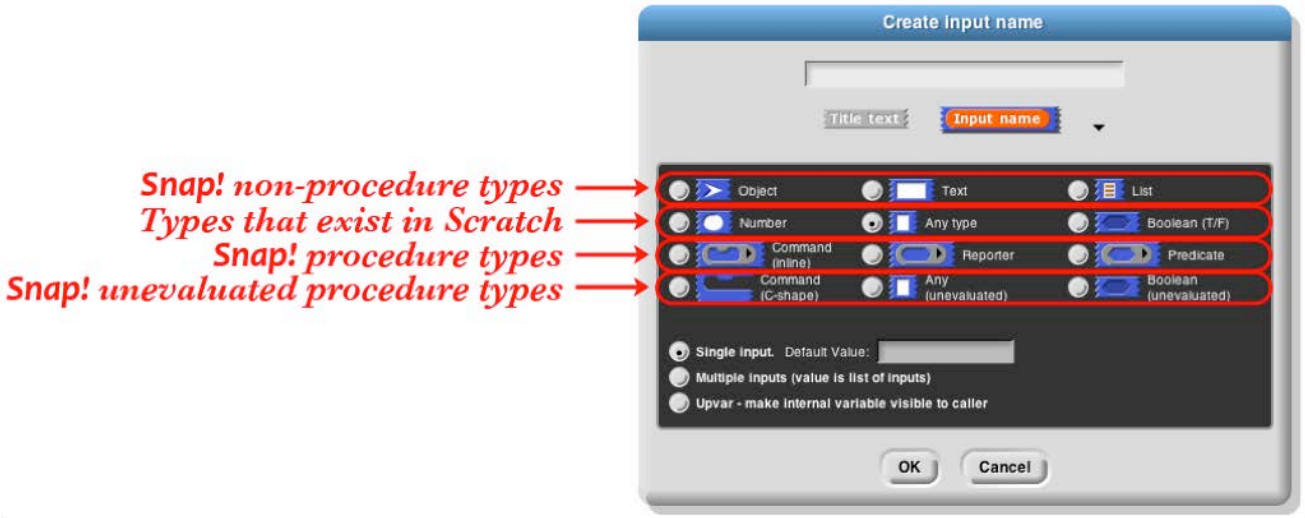
Clicking that arrowhead opens the "long" input name dialog:



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. If the **size** input in your block should be an oval-shaped numeric slot rather than a generic rectangle, click "**Number**."

The arrangement of the input types is systematic. As the pictures on this and 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 Snap! types other than procedures: Object, Text, and List. The last two rows are the types related to procedures, discussed more fully below.

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

The Object type is for sprites, costumes, sounds, and similar data types.

The Text type is really just a variant form of the Any type, using a shape that suggests a text input.[1]

Procedure Types

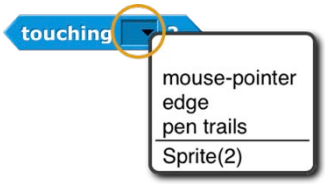
Although the procedure types are discussed more fully later, they are the key to understanding the column arrangement in the input types. Like Scratch, Snap! has three block shapes: jigsaw-piece for command blocks, oval for reporters, and hexagonal for predicates. (A *predicate* is a reporter that always reports **true** or **false**.) In Snap! 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.

[1] 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 some of us 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 just above Any in the input type selection box.)

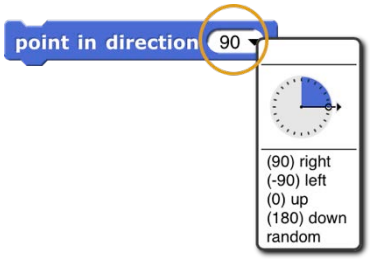
The unevaluated procedure types in the fourth row are explained in Section VII.E below. In one handwavy sentence, they combine the *meaning* of the procedure types with the *appearance* of the reported value types two rows higher. (Of course, this isn't quite right for the C-shaped command input type,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

Pulldown inputs

Certain primitive blocks have *pulldown* inputs, either *read-only*, like the input to the **touching** block:



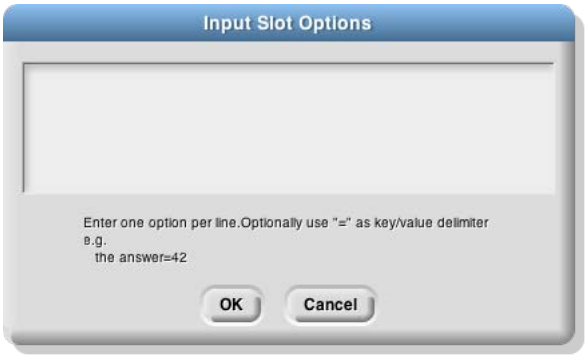
"(indicated by the input slot being the same (cyan, in this case) color as the body of the block), or *writeable*, like the input to the **point in direction** block:



(indicated by the white input slot), which means that the user can enter an arbitrary input instead of using the pulldown menu.

Custom blocks can also have such inputs. This is an experimental feature, and the user interface is likely to change! To make a pulldown input, open the long form input dialog and control-click/right-click in the dark grey area. You will see this menu:

Click the **read-only** checkbox if you want a read-only pulldown input. Then control-click/right-click again and choose **options…** to get this dialog box:

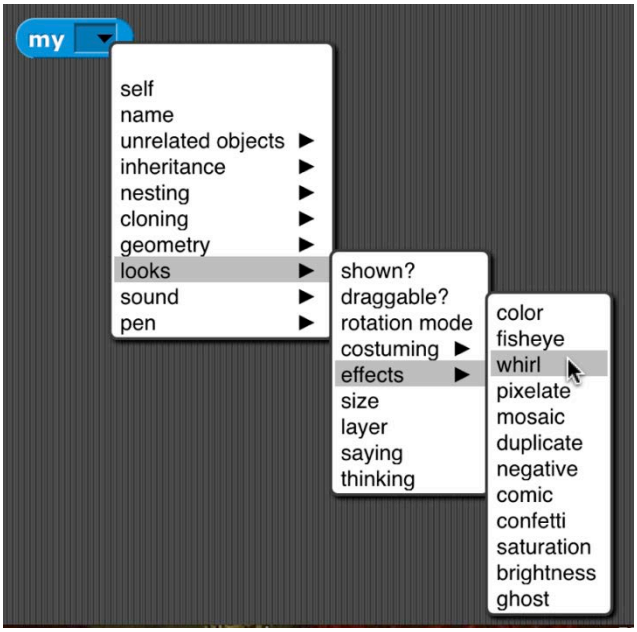


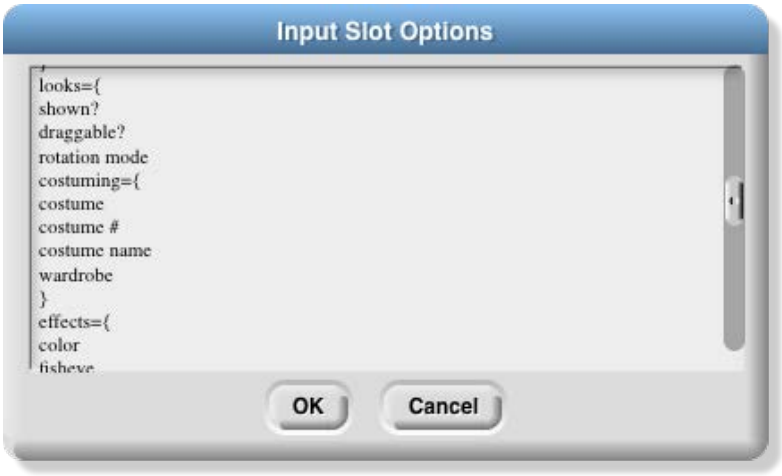
Each line in the text box represents one menu item. If the line does not contain any of the characters **=~{}** then the text is both what's shown in the menu and the value of the input if that entry is chosen.

If the line contains an equal sign **=**, then the text to the left of the equal sign is shown in the menu, and the text to the right is what appears in the input slot if that entry is chosen, and is also the value of the input as seen by the procedure.

If the line consists of a tilde **~**, then it represents a separator (a horizontal line) in the menu, used to divide long menus into visible categories. There should be nothing else on the line. This separator is not choosable, so there is no input value corresponding to it.

If the line ends with the two characters equal sign and open brace **={**, then it represents a *submenu.* The text before the equal sign is a name for the submenu, and will be displayed in the menu with an arrowhead ▶ at the end of the line. This line is not clickable, but hovering the mouse over it displays the submenu next to the original menu. A line containing a close brace **}** ends the submenu; nothing else should be on that line. Submenus may be nested to arbitrary depth.





It is also possible to get the special menus used in some primitive blocks, by choosing from the **menu** submenu: broadcast messages, sprites and stage, costumes, sounds, variables that can be **set** in this scope, the **play note** piano keyboard, or the **point in direction** 360° dial. Finally, you can make the input box accept more than one line of text (that is, text including a newline character) from the **special** submenu, either "**multi-line**" for regular text or "***code***" for `monospace-font` computer code.

Input variants

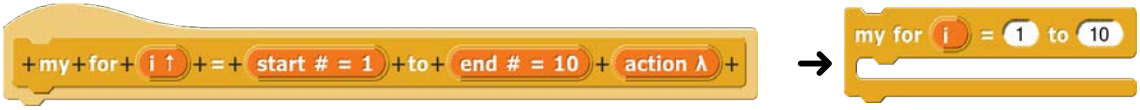
We now turn to the three mutually exclusive options that come below the type array.

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, Text, or Boolean, 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, Snap! introduces the arrowhead notation (◂ ▸) that expands and contracts the block, adding and removing input slots. Custom blocks made by the Snap! 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 or *variadic* input.

The third category, "**Upvar - make internal variable visible to caller**," isn't really an input at all, but rather a sort of output from the block to its user. It appears as an orange variable oval in the block, rather than as an input slot. Here's an example; the uparrow (↑) in the prototype indicates this kind of internal variable name:



The variable **i** (in the block on the right above) 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 calling 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.

Note about the example: **for** is a primitive block, but it doesn't need to be. You're about to see how it can be written in Snap!. Just give it a different name to avoid confusion, such as **my for** as above.

Prototype Hints

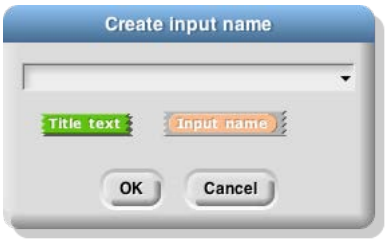
We have noted three notations that can appear in an input slot in the prototype to remind you of what kind of input this is. Here is the complete list of such notations:

=	default value	…	multiple input	↑	upvar	#	number
λ	procedure types	⋮	list	?	Boolean	⩾	object

Title Text and Symbols

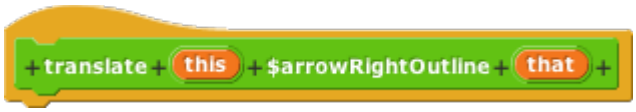
Some primitive blocks have symbols as part of the block name: 

Custom blocks can use symbols too. In the Block Editor, click the plus sign in the prototype at the point where you want to insert the symbol. Then click the **title text** picture below the text box that's expecting an input slot name. The dialog will then change to look like this:



The important part to notice is the arrowhead that has appeared at the right end of the text box. Click it to see the menu at the left.

Choose one of the symbols. The result will look kind of ugly in the prototype:



but the actual block will have the symbol you want:

The available symbols are, pretty much, the ones that are used in Snap! icons.

But I'd like the arrow symbol bigger, and yellow, so I edit its name:

This says to make the symbol **1.5** times as big as the letters in the block text, using a color with red-green-blue values of **255-255-150** (each between 0 and 255). Here's the result:

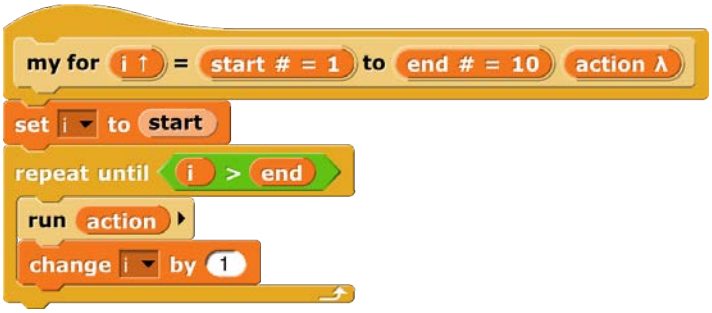
The size and color controls can also be used with text: **$foo-8-255-120-0** will make a huge orange "foo."

Note the last entry in the symbol menu: "new line." This can be used in a block with many inputs to control where the text continues on another line, instead of letting Snap! choose the line break itself.

VI. Procedures as Data

A. Call and Run

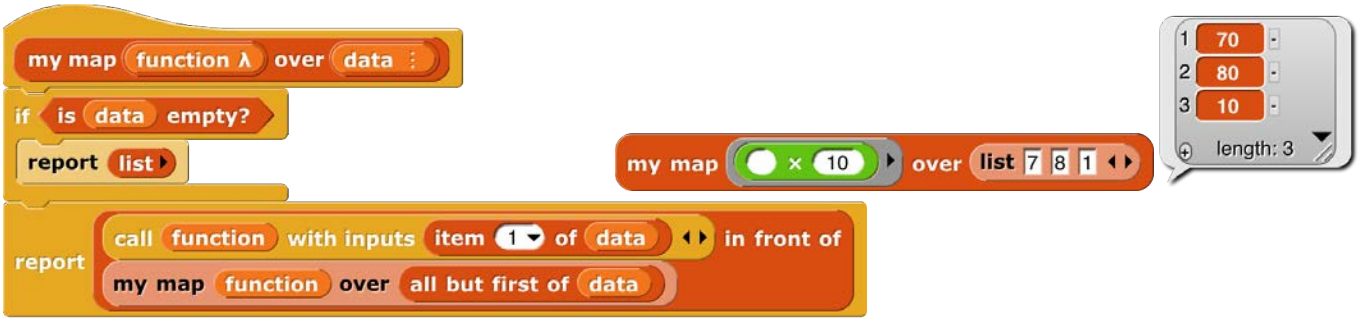
In the **for** block example above, the input named **action** has been declared as type "Command (C-shaped)"; that's why the finished block is C-shaped. But how does the block actually tell Snap! to carry out the commands inside the C-slot? Here is a simple version of the block script:



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 for each repetition instead of by 1.

The important part of this script is the **run** block near the end. This is a Snap! built-in command block that takes a Command-type value (a script) as its input, and carries out its instructions. (In this example, the value of the input action is the script that the user puts in the C-slot of the **my for** block.) There is a similar **call** reporter block for invoking a Reporter or Predicate block. The **call** and **run** blocks are at the heart of Snap!'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 in a **map** block (see page 38):



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. In the **my map** definition, the input **function** is declared to be type Reporter, and **data** is of type List.

Call/Run with inputs

The **call** block (like 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.

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:

If the number of inputs provided is neither one nor the number of empty slots, then there is no automatic filling of empty slots. (Instead you must use explicit parameters in the ring, as discussed in subsection C below.)

An even more important thing to notice about these examples is the *ring* around the Reporter-type input slots in **call** and **map** above. This notation indicates that *the block itself,* not the number or other value that the block would report when called, is the input. If you want to use a block itself in a non-Reporter-type (e.g., Any-type) input slot, you can enclose it explicitly in a ring, found at the top of the Operators palette.

As a shortcut, if you right-click or control-click on a block (such as the **+** block in this example), one of the choices in the menu that appears is "**ringify**" and/or "**unringify**." The ring indicating a Reporter-type or Predicate-type input slot is essentially the same idea for reporters as the C-shaped input slot with which you're already familiar; with a C-shaped slot, it's *the script* you put in the slot that becomes the input to the C-shaped block.

There are three ring shapes. All are oval on the outside, indicating that the ring reports a value, the block or script inside it, but the inside shapes are command, reporter, or predicate, indicating what kind of block or script is expected. Sometimes you want to put something more complicated than a single reporter inside a reporter ring; if so, you can use a script, but the script must **report** a value, as in a custom reporter definition.

Variables in Ring Slots

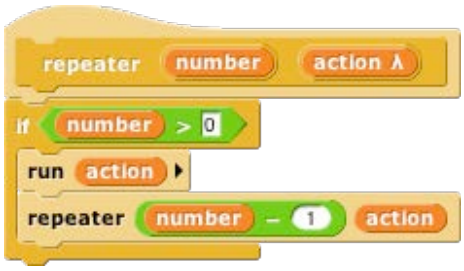
Note that the **run** block in the definition of the **my for** block (page 49) doesn't have a ring around its input variable **action**. When you drag a variable into a ringed input slot, you generally *do* want to use *the value of* the variable, which will be the block or script you're trying to run or call, rather than the orange variable reporter itself. So Snap! automatically removes the ring in this case. If you ever do want to use the variable *block itself,* rather than the value of the variable, as a Procedure-type input, you can drag the variable into the input slot, then control-click or right-click it and choose "**ringify**" from the menu that appears. (Similarly, if you ever want to call a function that will report a block to use as the input, such as **item 1 of** applied to a list *of blocks,* you can choose "**unringify**" from the menu. Almost all the time, though, Snap! does what you mean without help.)

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.)

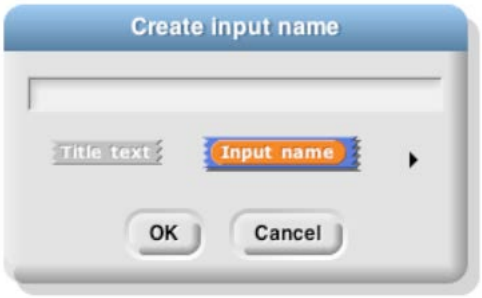
Although an Any-type input slot (what you get if you use the small input-name dialog box) will accept a procedure input, it doesn't automatically ring the input as described above. So the declaration of Procedure-type inputs makes the use of your custom higher order block much more convenient.

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

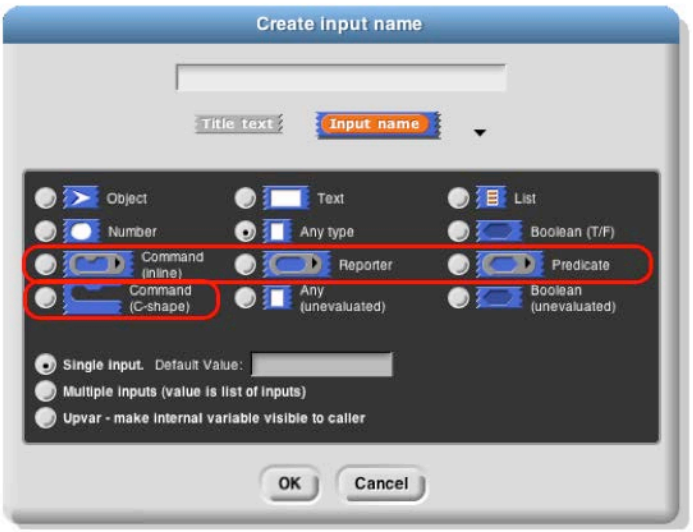


The lambda (λ) next to **action** in the prototype indicates that this is a C-shaped block, and that the script enclosed by the C when the block is used is the input named **action** in the body of the script. The only way to make sense of the variable **action** is to understand that its value is a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

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



"Then, in the long dialog, choose the appropriate Procedure type. The third row of input types has a ring 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 E below. The two Command-related input types (inline and C-shaped) 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 (and one predicate input), 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 an inline ring, 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 Snap! know where to use those inputs? If the called procedure (block or script) has empty input slots, Snap! "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 Snap! fills the empty slots from left to right:

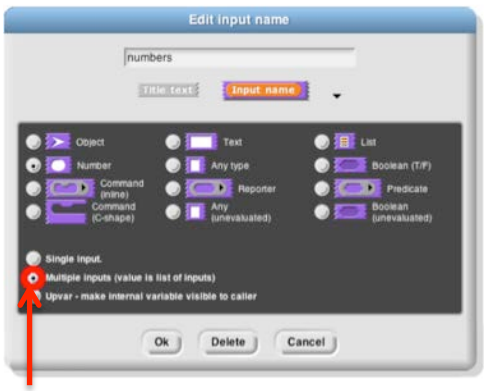
2. If exactly one input is provided, Snap! will fill any number of empty slots with it:

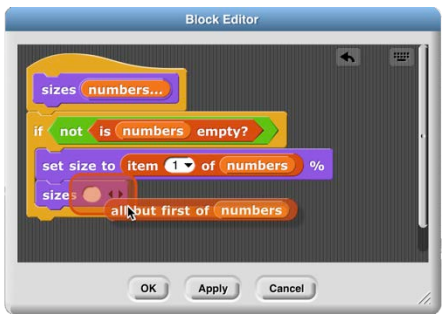
3. Otherwise, Snap! won't fill any slots, because the user's intention is unclear.

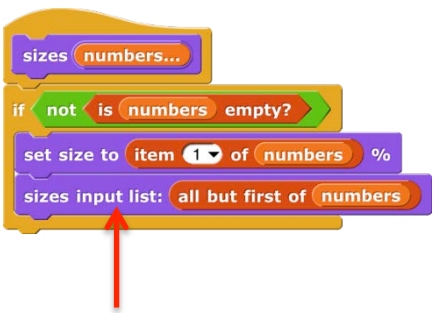
If the user wants to override these rules, the solution is to use a ring with explicit input names that can be put into the given block or script to indicate how inputs are to be used. This will be discussed more fully below.

Recursive Calls to Multiple-Input Blocks

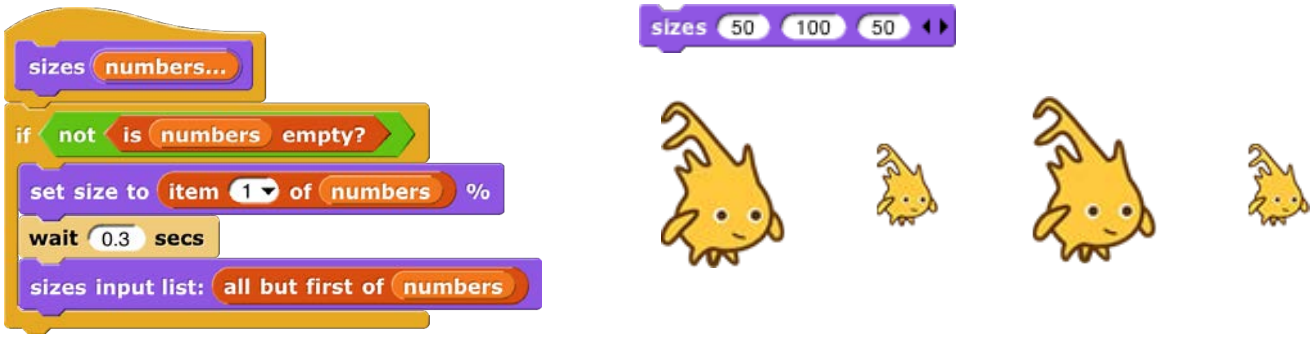
A relatively rare situation not yet considered here is the case of a recursive block that has a variable number of inputs. Let's say the user of your project calls your block with five inputs one time, and 87 inputs another time. How do you write the recursive call to your block when you don't know how many inputs to give it? The answer is that you collect the inputs in a list (recall that, when you declare an input name to represent a variable number of inputs, your block sees those inputs as a list of values in the first place), and then, in the recursive call, you drop that input list *onto the arrowheads* that indicate a variable-input slot, rather than onto the input slot:







Note that the halo you see while dragging onto the arrowheads is red instead of white, and covers the input slot as well as the arrowheads. And when you drop the expression onto the arrowheads, the words "**input list:**" are added to the block text and the arrowheads disappear (in this invocation only) to remind you that the list represents all of the multiple inputs, not just a single input. 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. 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. But **sizes** doesn't take a list as input; it takes numbers as inputs! So this would be wrong:

C. Formal Parameters

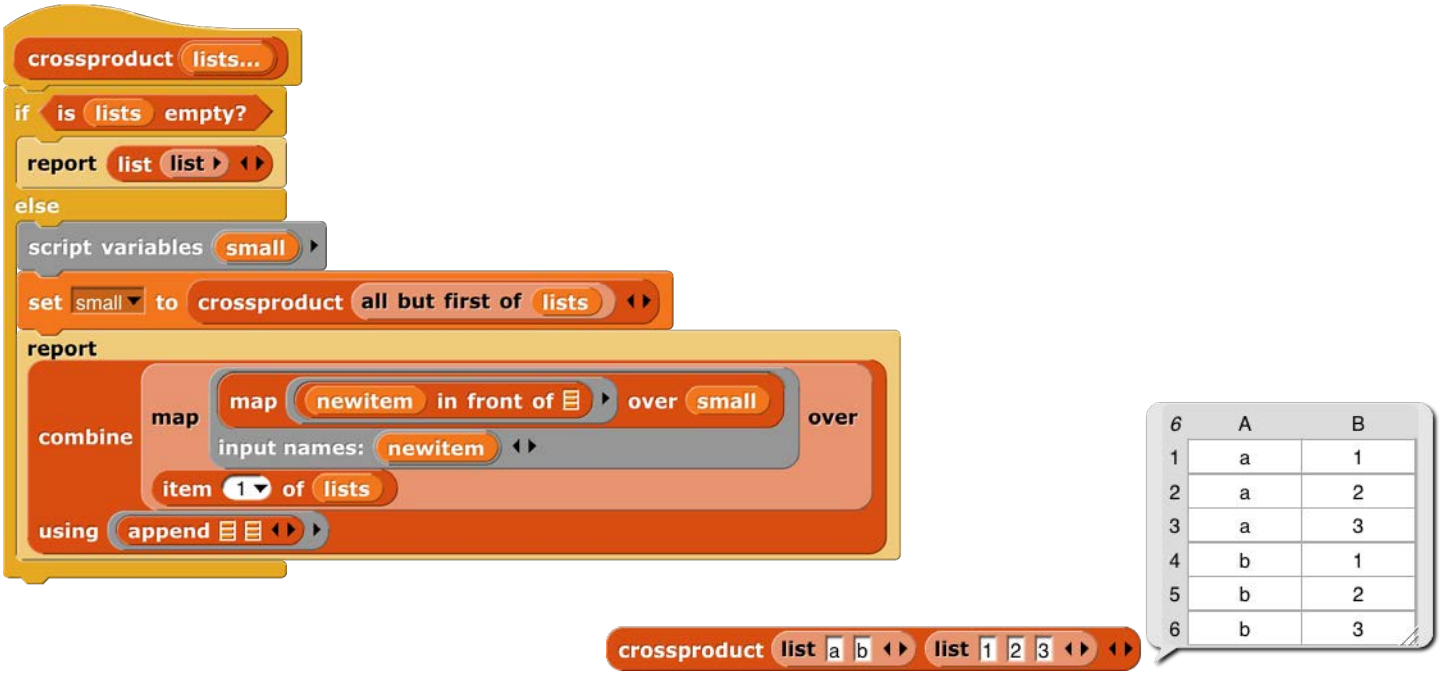
The rings around Procedure-type inputs have an arrowhead at the right. Clicking the arrowhead allows you to give the inputs to a block or script explicit names, instead of using empty input slots as we've done until now.

The names **#1**, **#2**, etc. are provided by default, but you can change a name by clicking on its orange oval in the **input names** list. Be careful not to *drag* the oval when clicking; that's how you use the input inside the ring. The names of the input variables are called the *formal parameters* of the encapsulated procedure.

Here's a simple but contrived example using explicit names to control which input goes where inside the ring:

Here we just want to put one of the inputs into two different slots. If we left all three slots empty, Snap! would not fill any of them, because the number of inputs provided (2) would not match the number of empty slots (3).

Here is a more realistic, much more advanced example:

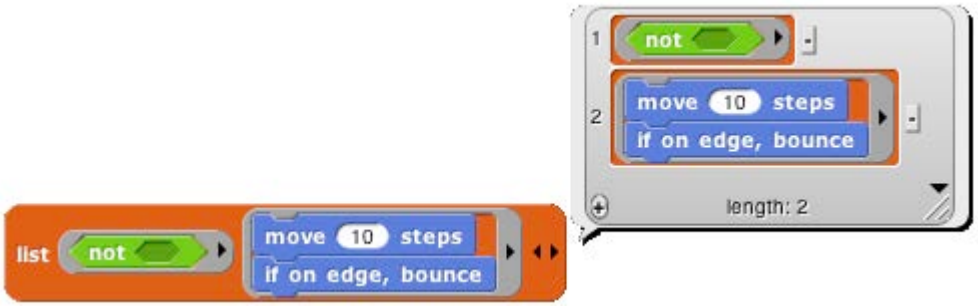


This is the definition of a block that takes any number of lists, and reports the list of all possible combinations of one item from each list. The important part for this discussion is that near the bottom there are two *nested* calls to **map**, the higher order function that applies an input function to each item of an input list. In the inner block, the function being mapped is **in front of**, 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 **in front of** 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 **in front of** block.

By the way, once the called block provides names for its inputs, Snap! 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 Snap! from filling a slot that should really remain empty.

D. Procedures as Data

Here's an example of a situation in which a procedure must be explicitly marked as data by pulling a ring from the Operators palette and putting the procedure (block or script) inside it:



Here, we are making a list of procedures. But the **list** block accepts inputs of any type, so its input slots are not ringed. We must say explicitly that we want the block *itself* as the input, rather than whatever value would result from evaluating the block.

Besides the **list** block in the example above, other blocks into which you may want to put procedures are **set** (to 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):

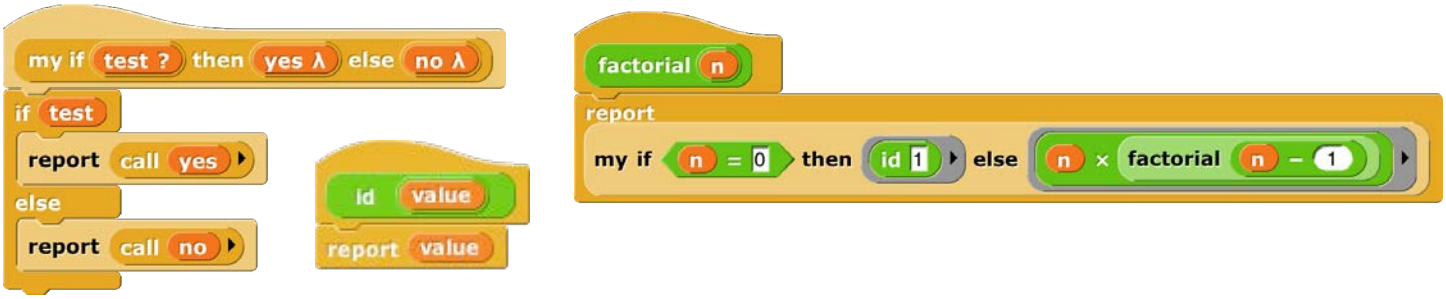
E. Special Forms

The primitive **if else** block 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. Snap! has one, but we could write our own:

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 block is called, all of its inputs are computed (evaluated) before the block itself runs. The block itself knows only 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, those inputs will be enclosed in a ring 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 rings, and about a trick to get a constant value in a ringed slot. (The **id** block implements the identity function,

which reports its input. We need it because rings take only 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 an "invisible ring" 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 rings, and the **factorial** block will look like our first attempt.

In a special form's prototype, the unevaluated input slot(s) are indicated by a lambda (λ) next to the input name, just as if they were declared as Procedure type. They *are* Procedure type, really; they're just disguised to the user of the block.

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 **my if then else**. But any experienced Scratch programmer can *use* **my if then else** without thinking at all about how it works internally.

Special Forms in Scratch

Special forms are actually not a new invention in Snap*!*. 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 C-shaped blocks, it can afford to handwave away the distinction between evaluated and unevaluated Booleans, but Snap*!* 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 a ring 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 Scratchers don't think of the contents of a C-slot as an input at all.) This is why it makes sense that "C-shaped" is on the fourth row of types in the long form input dialog, with other unevaluated types.

VII. 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). 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. Our approach is therefore less restrictive than that of some other OOP languages; we give objects easy access to each others' data and methods.

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 identical to those of other objects, 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

Like Scratch, Snap! 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 two ways in which Scratch sprites are less versatile than the objects of an OOP language. First, Scratch message passing is weak 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, and more basic, in the OOP paradigm objects are *data;* they can be the value of a variable, an element of a list, and so on, but that's not the case for Scratch sprites.

Snap! sprites are first class data. They can be created and deleted by a script, stored in a variable or list, and sent messages individually. The children of a sprite can inherit sprite-local variables, methods (sprite-local procedures), and other attributes (e.g., **x position**).

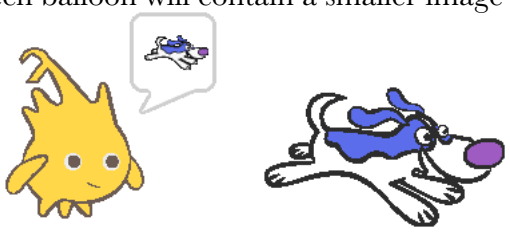
The fundamental means by which programs get access to sprites is the **my** reporter block. It has a dropdown-menu input slot that, when clicked, gives access to all the sprites, plus the stage. 

reports a single sprite, the one asking the question. 

reports a list of all sprites other than the one asking the question. 

reports a list of all sprites that are *near* the one asking—the ones that are candidates for having collided with this one, for example. The **my** block has many other options, discussed below. If you know the name of a particular sprite, the **object** reporter will report the sprite itself.

An object or list of objects reported by **my** or **object** can be used as input to any block that accepts any input type, such as **set**'s second input. If you **say** an object, the resulting speech balloon will contain a smaller image of the object's costume or (for the stage) background.

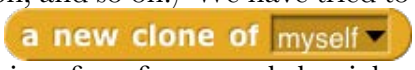


B. Permanent and Temporary Clones

The 

block is used to create and report an instance (a clone) of any sprite. (There is also a command version, for historical reasons.) There are two different kinds of situations in which clones are used. One is that you've made an example sprite and, when you start the project, you want a fairly large number of essentially identical sprites that behave like the example. (Hereafter we'll call the example sprite the "parent" and the others the "children.") Once the game or animation is over, you don't need the copies any more. (As we'll see, "copies" is the wrong word because the parent and the children *share* a lot of properties. That's why we use the word "clones" to describe the children rather than "copies.") These are *temporary* clones. They are automatically deleted when the user presses either the green flag or the red stop sign. In Scratch 2.0 and later, all clones are temporary.

The other kind of situation is what happens when you want specializations of sprites. For example, let's say you have a sprite named Dog. It has certain behaviors, such as running up to a person who comes near it. Now you decide that the family in your story really likes dogs, so they adopt a lot of them. Some are cocker spaniels, who wag their tails when they see you. Others are rottweilers, who growl at you when they see you. So you make a clone of Dog, perhaps rename it Cocker Spaniel, and give it a new costume and a script for what to do when someone gets near. You make another clone of Dog, perhaps rename it Rottweiler, and give it a new costume, etc. Then you make three clones of Cocker Spaniel (so there are four altogether) and two clones of Rottweiler. Maybe you hide the Dog sprite after all this, since it's no breed in particular. Each dog has its own position, special behaviors, and so on. You want to save all of these dogs in the project. These are *permanent* clones. In BYOB 3.1, the predecessor to Snap!, all clones are permanent.

One advantage of temporary clones is that they don't slow down Snap! even when you have a lot of them. (If you're curious, one reason is that permanent clones appear in the sprite corral, where their pictures have to be updated to reflect the clone's current costume, direction, and so on.) We have tried to anticipate your needs, as follows: When you make a clone in a script, using the 

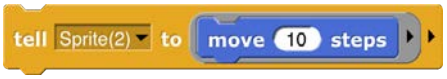
block, it is "born" temporary. But when you make a clone from the user interface, for example by right-clicking on a sprite and choosing "clone," it is born permanent. The reason this makes sense is that you don't create 100 *kinds* of dogs automatically. Each kind has many different characteristics, programmed by hand. But when your project is running, it might create 100 rottweilers, and those will be identical unless you change them in the program.

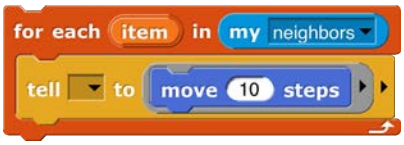
You can change a temporary sprite to permanent by right-clicking it and choosing "edit." (It's called "edit" rather than, say, "permanent" because it also shifts the scripting area to reflect that sprite, as if you'd pressed its button in the sprite corral.) You can change a permanent sprite to temporary by right-clicking it and choosing "release."

C. 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.

The way to send a message to a sprite (or the stage) is with the **tell** block (for command messages) or the **ask** block (for reporter messages).





A small point to note in the examples above: most of the primitive blocks' dropdown menus include an empty entry at the top, which can be selected for use in higher order procedures like the **for each** and **map** examples. Each of the sprites in **my neighbors** or **my other sprites** is used to fill the blank space in turn.

By the way, if you want a list of *all* the sprites, including this sprite, you can use either of these:

Tell and **ask** wait until the other sprite has carried out its method before this sprite's script continues. (That has to be the case for **ask**, since we want to do something with the value it reports.) So **tell** is analogous to **broadcast and wait**. Sometimes the other sprite's method may take a long time, or may even be a **forever** loop, so you want the originating script to continue without waiting. For this purpose we have the **launch** block:

Launch is analogous to **broadcast** without the "wait."

Snap! 4.1, following BYOB 3.1, used an extension of the **of** block to provide access to other sprites' methods. That interface was designed back when we were trying hard to avoid adding new primitive blocks; it allowed us to write **ask** and **tell** as tool procedures in Snap! itself. That technique still works, but is deprecated, because nobody understood it, and now we have the more straightforward primitives.

Polymorphism

Suppose you have a Dog sprite with two clones CockerSpaniel and PitBull. In the Dog sprite you define this method ("For this sprite only" block):

Note the *loca*tion (map-pin) symbol before the block's name. The symbol is not part of the block title; it's a visual reminder that this is a sprite-*local* block. Sprite-local variables are similarly marked.

But you don't define **greet as friend** or **greet as enemy** in Dog. Each kind of dog has a different behavior. Here's what a CockerSpaniel does:

And here's what a PitBull does:

Greet is defined in the Dog sprite. If Fido is a particular cocker spaniel, and you ask Fido to **greet** someone, Fido inherits the **greet** method from Dog, but Dog itself couldn't actually run that method, because Dog doesn't have **greet as friend** or **greet as enemy**. And perhaps only individual dogs such as Fido have **friend?** methods. Even though the **greet** method is defined in the Dog sprite, when it's running it remembers what specific dog sprite called it, so it knows which **greet as friend** to use. Dog's **greet** block is called a *polymorphic* method, because it means different things to different dogs, even though they all share the same script.

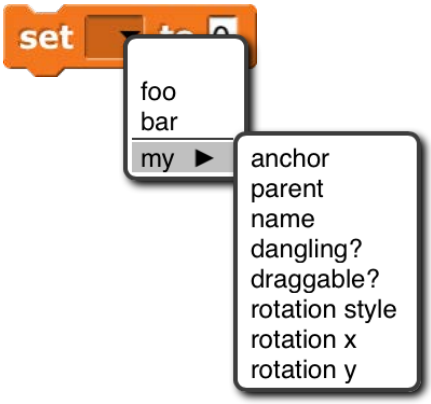
D. Local State in Sprites: Variables and Attributes

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, however, 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** 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 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** 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 direction example above) that modifies a variable or attribute is called a *setter.*

In Snap! we allow virtually all attributes to be examined. But instead of adding dozens of reporters, we use a more uniform interface for attributes: The **my** block's menu (in Sensing; see page 62) includes many of the attributes of a sprite. It serves as a general getter for those attributes, e.g., **my [anchor]** to find the sprite, if any, to which this sprite is attached in a nesting arrangement (see page 10). Similarly, the same **set** block used to set variable values allows setting some sprite attributes.

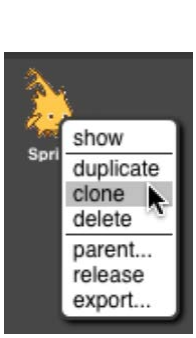


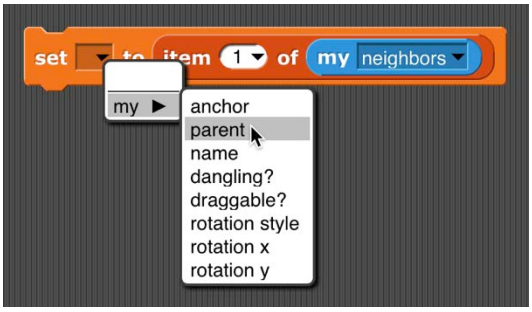
E. Prototyping: Parents and Children

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. Snap! 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); you can actually watch it and interact with it on the stage; 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/instance hierarchy 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.[1]

There are three ways to make a child sprite. 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. There is an **a new clone of** block in the Control palette that creates and reports a child sprite. And sprites have a "parent" attribute that can be set, like any attribute, thereby *changing* the parent of an existing sprite.





F. 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 only its methods, wardrobe, and jukebox with its parent. All other properties are copied to the clone, but not shared. (One exception is that a new *permanent* clone is given a random position. Another is that *temporary* clones share the scripts in their parent's scripting area. A third is that sprite-local variables that the parent creates *after* cloning are shared with its children.) 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 uses the **inherit** command block. The pulldown menu in the block lists all the things this sprite can inherit from its parent (which might be nothing, if this sprite has no parent) and is not already inheriting. But that would prevent **tell**ing a child to inherit, so if the **inherit** block is inside a ring, its pulldown menu includes all the things a child could inherit from this sprite. Right-clicking on the scripting area of a permanent clone gives a menu option to share the entire collection of scripts from its parent, as a temporary clone does.

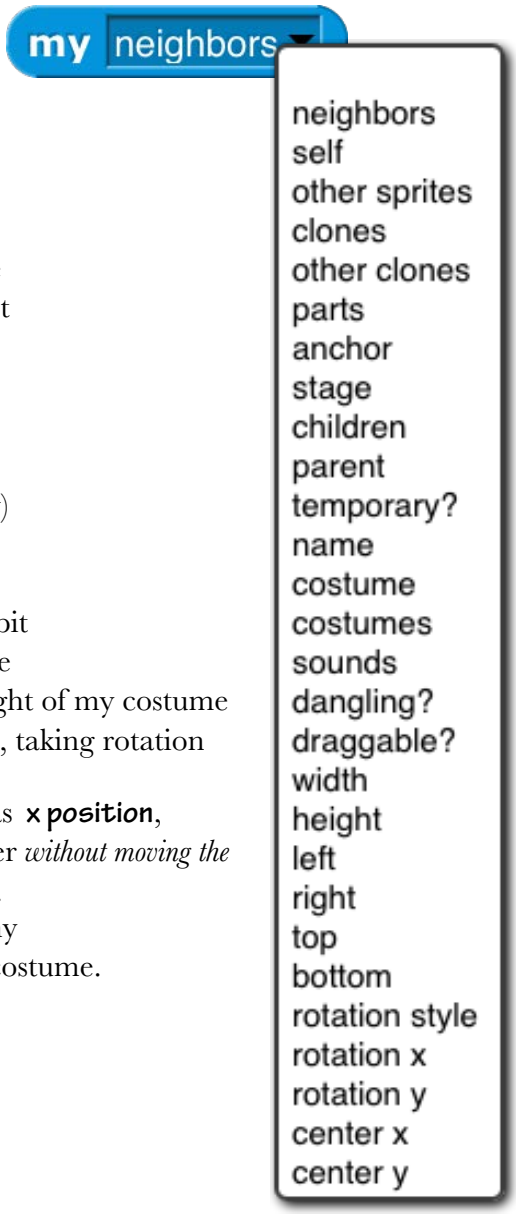
[1] Some languages popular in the "real world" today, such as JavaScript, claim to use prototyping, but their object system is much more complicated than what we are describing (we're guessing it's because they were designed by people too familiar with class/instance programming); that has, in some circles, given prototyping a bad name. Our prototyping design comes from Object Logo, and before that, from Henry Lieberman. [Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, OR, September, 1986. Also in *Object-Oriented Computing,* Gerald Peterson, Ed., IEEE Computer Society Press, 1987.]

The rules are full of details, but the basic idea is simple: Parents can change their children, but children can't directly change their parents. That's what you'd expect from the word "inherit": the influence just goes in one direction. When a child changes some property, it's declaring independence from its parent (with respect to that one property). What if you really want the child to be able to make a change in the parent (and therefore in itself and all its siblings)? Remember that in this system any object can **tell** any other object to do something:

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 **my (self)**, it means the child to which the message was originally sent, not the parent to which the message was delegated.

G. List of attributes

At the right is a picture of the dropdown menu of attributes in the **my** block. Several of these are not real attributes, but things related to attributes:



- **self**: this sprite
- **neighbors**: a list of *nearby* sprites
- **other sprites**: a list of all sprites except myself
- **stage**: the stage, which is first-class, like a sprite
- **clones**: a list of my *temporary* clones
- **other clones**: a list of my *temporary* siblings
- **parts**: a list of sprites whose **anchor** attribute is this sprite
- **children**: a list of all my clones, temporary and permanent

The others are individual attributes:

- **anchor**: the sprite of which I am a (nested) part
- **parent**: the sprite of which I am a clone
- **temporary?**: am I a temporary clone?
- **name**: my name (same as parent's name if I'm temporary)
- **costumes**: a list of the sprite's costumes
- **sounds**: a list of the sprite's sounds
- **dangling?**: True if I am a part and not in synchronous orbit
- **draggable?**: True if the user can move me with the mouse
- **width**, **height**, **left**, **right**, **top**, **bottom**: The width or height of my costume *as seen right now,* or the left, etc., edge of my bounding box, taking rotation into account.
- **rotation x**, **rotation y**: when reading with **my**, the same as **x position**, **y position**. When **set**, changes the sprite's rotation center *without moving the sprite,* like dragging the rotation center in the paint editor.
- **center x**, **center y**: the x and y position of the center of my bounding box, rounded off–the geometric center of the costume.

H. First Class Costumes and Sounds

Costumes and sounds don't have methods, as sprites do; you can't ask them to do things. But they *are* first class: you can make a list of them, put them in variables, use them as input to a procedure, and so on. **My [costumes]** and **my [sounds]** report lists of them.

Media Computation with Costumes

The components of a costume are its name, width, height, and pixels. The block 

gives access to these components using its left menu. From its right menu you can choose the current costume, the Turtle costume, or any costume in the sprite's wardrobe. Since costumes are first class, you can also drop an expression whose value is a costume on that second input slot. (Due to a misfeature, even though you can select Turtle in the right menu, the block reports 0 for its width and height, and an empty string for the other components.) The costume's width and height are in its standard orientation, regardless of the sprite's current direction. (This is different from the *sprite's* width and height, reported by the **my** block.)

But the really interesting part of a costume is its bitmap, a list of *pixels*. (A pixel, short for "picture element," represents one dot on your display.) Each pixel is itself a list of four items, the red, green, and blue components of its color (in the range 0-255) and what is standardly called its "transparency" but should be called its opacity, also in the range 0-255, in which 0 means that the pixel is invisible and 255 means that it's fully opaque: you can't see anything from a rearward layer at that point on the stage. (Costume pixels typically have an opacity of 0 only for points inside the bounding box of the costume but not actually part of the costume; points in the interior of a costume typically have an opacity of 255. Intermediate values appear mainly at the edge of a costume, or at sharp boundaries between colors inside the costume, where they are used to reduce "jaggies": the stairstep-like shape of a diagonal line displayed on an array of discrete rectangular screen coordinates. Note that the opacity of a *sprite* pixel is determined by combining the costume's opacity with the sprite's ghost effect. (The latter really is a measure of transparency: 0 means opaque and 100 means invisible.)

The bitmap is a one-dimensional list of pixels, not an array of *height* rows of *width* pixels each. That's why the pixel list has to be combined with the dimensions to produce a costume. This choice partly reflects the way bitmaps are stored in the computer's hardware and operating system, but also makes it easy to produce transformations of a costume with **map**:

In this simplest possible transformation, the red value of all the pixels have been changed to a constant 150. Colors that were red in the original (such as the logo printed on the t-shirt) become closer to black (the other color components being near zero); the blue jeans become purple (blue plus red); perhaps counterintuitively, the white t-shirt, which has the maximum value for all three color components, loses some of its red and becomes cyan, the color opposite red on the color wheel. In reading the code, note that the function that is the first input to **map** is applied to a single pixel, whose first item is its red component. Also note that this process works only on bitmap costumes; if you call **pixels of** on a vector costume (one with "svg" in the corner of its picture), it will be converted to pixels first.

One important point to see here is that a bitmap (list of pixels) is not, by itself, a costume. The **new costume** block creates a costume by combining a bitmap, a width, and a height. But, as in the example above, **switch to costume** will accept a bitmap as input and will automatically use the width and height of the current costume. Note that there's no **name** input; costumes computed in this way are all named **costume**. Note also that the use of **switch to costume** does *not* add the computed costume to the sprite's wardrobe; to do that, say

Here's a more interesting example of color manipulation:

Each color value is constrained to be 0, 80, 160, or 240. This gives the picture a more cartoonish look. Alternatively, you can do the computation taking advantage of hyperblocks:

Here's one way to exchange red and green values:

It's the list 

that determines the rearrangement of colors: green→red, red→green, and the other two unchanged. That **list** is inside another **list** because otherwise it would be selecting *rows* of the pixel array, and we want to select columns. We use **pixels of costume current** rather than **costume apple** because the latter is always a red apple, so this little program would get stuck turning it green, instead of alternating colors.

The **stretch** block takes a costume as its first input, either by selecting a costume from the menu or by dropping a costume-valued expression such as 

onto it. The other two inputs are percents of the original width and height, as advertised, so you can make fun house mirror versions of costumes:

The resulting costumes can be used with **switch to costume** and so on.

Finally, you can use pictures from your computer's camera in your projects using these blocks:

Using the **video on** block turns on the camera and displays what it sees on the stage, regardless of the inputs given. The camera remains on until you click the red stop button, your program runs the **stop all** block, or you turn it off explicitly with the 

block. The video image on the stage is partly ghosted, to an extent determined by the **set video transparency** block, whose input really is transparency and not opacity.

(Small numbers make the video more visible.) By default, the video image is mirrored, like the selfie camera on your cell phone: When you raise your left hand, your image raises its right hand. You can control this mirroring with the 

block.

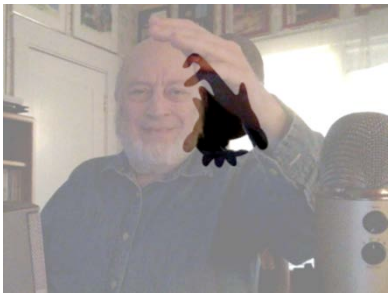
The ***video snap on*** block then takes a still picture from the camera, and trims it to fit on the selected sprite. (***Video snap on stage*** means to use the entire stage-sized rectangle.) For example, here's a camera snapshot trimmed to fit Alonzo:

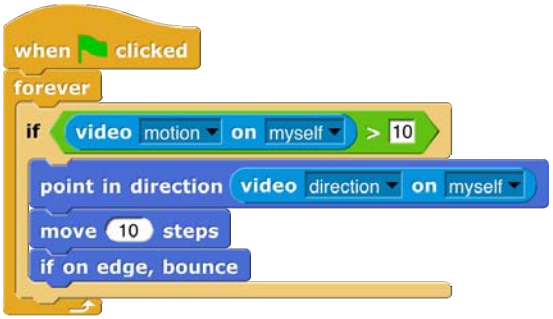
The "Video Capture" project in the Examples collection repeatedly takes such trimmed snapshots and has the Alonzo sprite use the current snapshot as its costume, so it looks like this:

(The picture above was actually taken with transparency set to 50, to make the background more visible for printing.) Because the sprite is always still in the place where the snapshot was taken, its costume exactly fits in with the rest of the full-stage video. If you were to add a ***move 100 steps*** block after the ***switch to costume***, you'd see something like this:

This time, the sprite's costume was captured at one position, and then the sprite is shown at a different position. (You probably wouldn't want to do this, but perhaps it's helpful for explanatory purposes.)

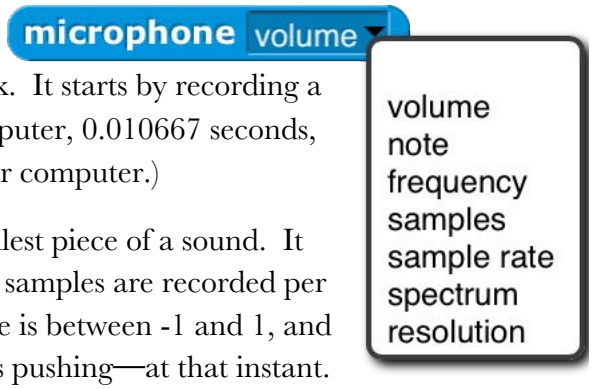
What you *would* want to do is push the sprite around the stage:





(Really these should be Jens's picture; it's his project. But he's vacationing. ☺) ***Video motion*** compares two snapshots a moment apart, looking only at the part within the given trim (here ***myself***, meaning the current sprite, not the person looking into the camera), to detect a difference between them. It reports a number, measuring the number of pixels through which some part of the picture has moved. ***Video direction*** also compares two snapshots to detect motion, but what it reports is the direction (in the ***point in direction*** sense) of the motion. So the script above moves the sprite in the direction in which it's being pushed, but only if a significant amount of motion is found; otherwise the sprite would jiggle around too much. And yes, you can run the second script without the first to push a balloon around the stage.

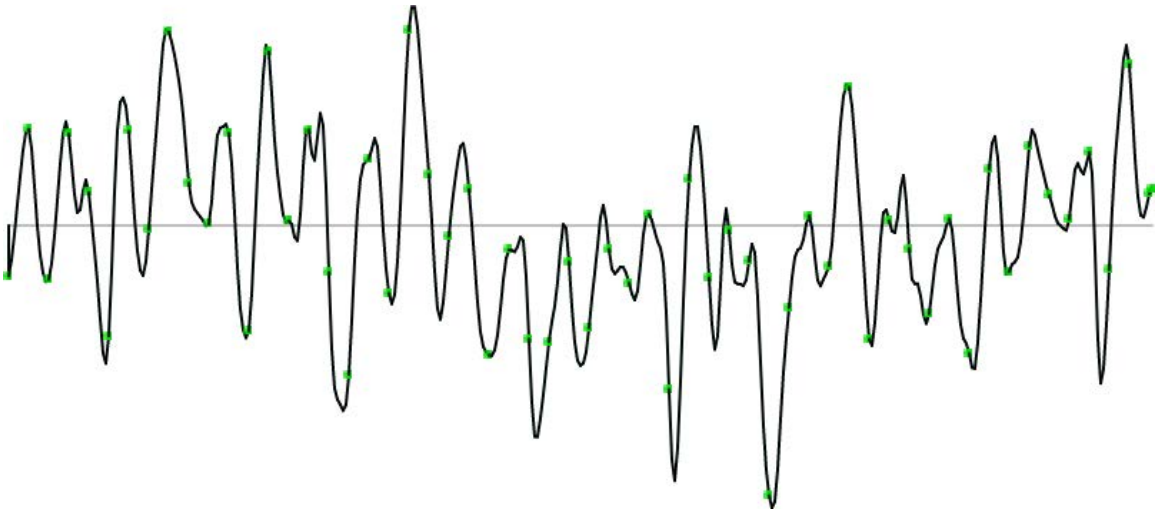
Media Computation with Sounds



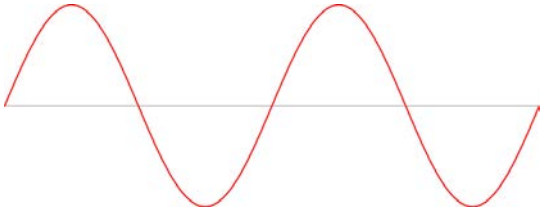
The starting point for computation with sound is the ***microphone*** block. It starts by recording a brief burst of sound from your microphone. (How brief? On my computer, 0.010667 seconds, but you'll see shortly how to find out or control the sample size on your computer.)

Just as the *pixel* is the smallest piece of a picture, the *sample* is the smallest piece of a sound. It says here: 

that on my computer, 48,000 samples are recorded per second, so each sample is 1/48,000 of a second. The value of a sample is between -1 and 1, and represents the sound pressure on the microphone—how hard the air is pushing—at that instant. (You can skip the next page or so if you know about Fourier analysis.) Here's a picture of 400 samples:



In this graph, the *x* axis represents the time at which each sample was measured; the *y* axis measures the value of the sample at that time. The first obvious thing about this graph is that it has a lot of ups and downs. The most basic up-and-down function is the *sine wave:*



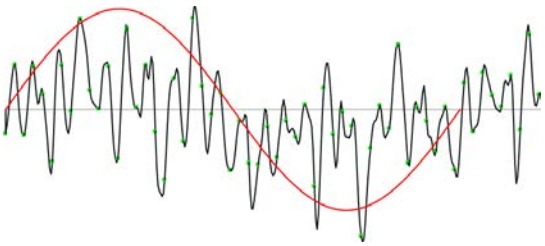
Every periodic function (more or less, any sample that sounds like music rather than sounding like static) is composed of a sum of sine waves of different frequencies.

Look back at the graph of our sampled sound. There is a green dot every seven samples. There's nothing magic about the number seven; I tried different values until I found one that looked right. What "right" means is that, for the first few dots at least, they coincide almost perfectly with the high points and low points of the graph. Near the middle (horizontally) of the graph, the green dots don't seem anywhere near the high and low points, but if you find the very lowest point of the graph, about 2/3 of the way along, the dots start lining up almost perfectly again.

The red graph above shows two *cycles* of a sine wave. One cycle goes up, then down, then up again. The amount of time taken for one cycle is the *period* of the sine function. If the green dots match both ups and downs in the captured sound, then two dots—14 samples, or 14/48000 of a second—represent the period. The first cycle and a half of the graph looks like it could be a pure sine wave, but after that, the tops and bottoms don't line up, and there are peculiar little jiggles, such as the one before the fifth green dot. This happens because sine waves of different periods are added together.

It turns out to be more useful to measure the reciprocal of the period, in our case, 48000/14 or about 3429 *cycles per second.* Another name for "cycles per second" is "Hertz," abbreviated Hz, so our sound has a component at 3249 Hz. As a musical note, that's about an A (a little flat), four octaves above middle C. (Don't worry too much about the note being a little off; remember that the 14-sample period was just eyeballed and is unlikely to be exactly right.)

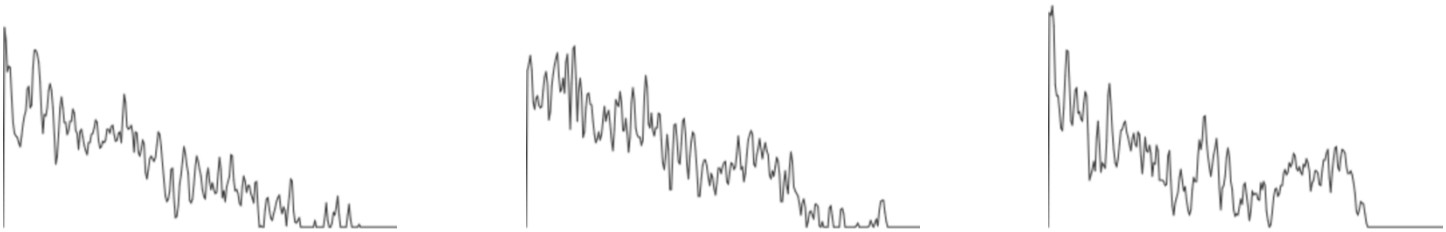
Four octaves above middle C is really high! That would be a shrill-sounding note. But remember that a complex waveform is the sum of multiple sine waves at different frequency. Here's a different up-and-down regularity:



It's not obvious, but in the left part of the graph, the signal is more above the *x* axis than below it. Toward the right, it seems to be more below than above the axis. At the very right it looks like it might be climbing again.

The period of the red sine wave is 340 samples, or 340/48000 second. That's a frequency of about 141 Hz, about D below middle C. Again, this is measuring by eyeball, but likely to be close to the right frequency.

All this eyeballing doesn't seem very scientific. Can't we just get the computer to find all the relevant frequencies? Yes, we can, using a mathematical technique called *Fourier analysis.* (Jean-Baptiste Joseph Fourier, 1768–1830, made many contributions to mathematics and physics, but is best known for working out the nature of periodic functions as a sum of sine waves.) Luckily we don't have to do the math; the **microphone** block will do it for us, if we ask for **microphone spectrum**:



These are frequency spectra from (samples of) three different songs. The most obvious thing about these graphs is that their overall slope is downward; the loudest frequency is the lowest frequency. That's typical of music.

The next thing to notice is that there's a regularity in the spacing of spikes in the graph. This is partly just an artifact; the frequency (horizontal) axis isn't continuous. There are a finite number of "buckets" (default: 512), and all the frequencies within a bucket contribute to the amplitude (vertical axis) of that bucket. The spectrum is a list of that many amplitudes. But the patterns of alternating rising and falling values are real; the frequencies that are multiples of the main note being sampled will have higher amplitude than other frequencies.

samples and **spectrum** are the two most detailed representations of a sound. But the **microphone** block has other, simpler options also:

volume	the instantaneous volume when the block is called
note	the MIDI note number (as in **play note**) of the main note heard
frequency	the frequency in Hz of the main note heard
sample rate	the number of samples being collected per second
resolution	the size of the array in which data are collected (typically 512, must be a power of 2)

The block for sounds that corresponds to **new picture** for pictures is 

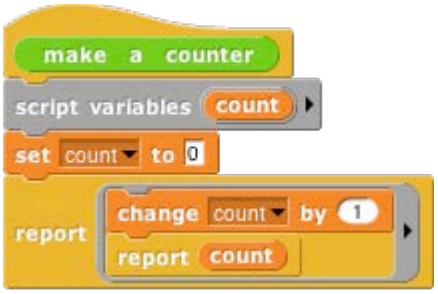
Its first input is a list of samples, and its second input specifies how many samples occupy one second.

VIII. OOP with Procedures

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 and lexical scope 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 works, and build up to full implementations of 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, the ringed script 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:

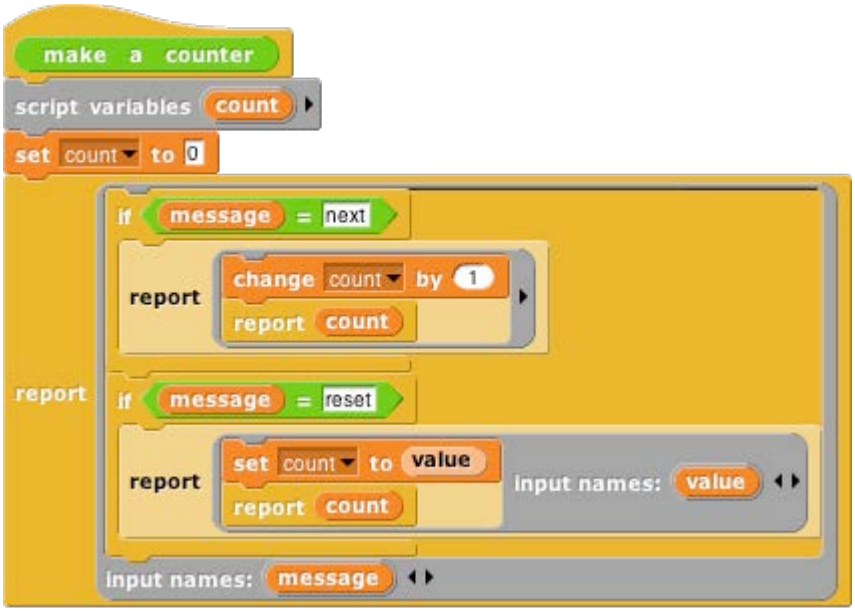
This example will repay careful study, because it isn't obvious why each instance has a separate count. From the point of view of the **make a counter** procedure, each invocation causes a new **count** variable to be created. Usually such *script variables* are temporary, going out of existence when the script ends. But this one is special, because **make a counter** returns *another script* 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 be "exported" 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 ring 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 refined version that uses message passing:



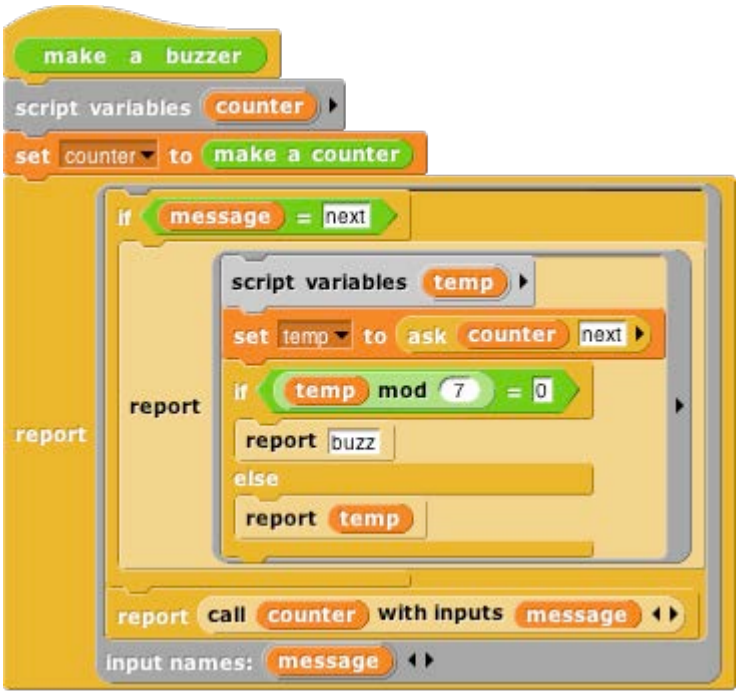
Again, the **make a counter** block represents the **counter** class, and again the script creates a local variable **count** each time it is invoked. The large outer ring represents an instance. It is a *dispatch procedure:* it takes a **message** (just a text word) as input, and it reports a method. The two smaller rings 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 Snap! puts in a list; that list is named **args** inside the block. **Ask** 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 drop a list of values onto the arrowheads of a multiple input (in the outer **call** block). Note also that this is one of the rare cases in which we must unringify the inner **call** block, whose *value when called* gives the method.

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:

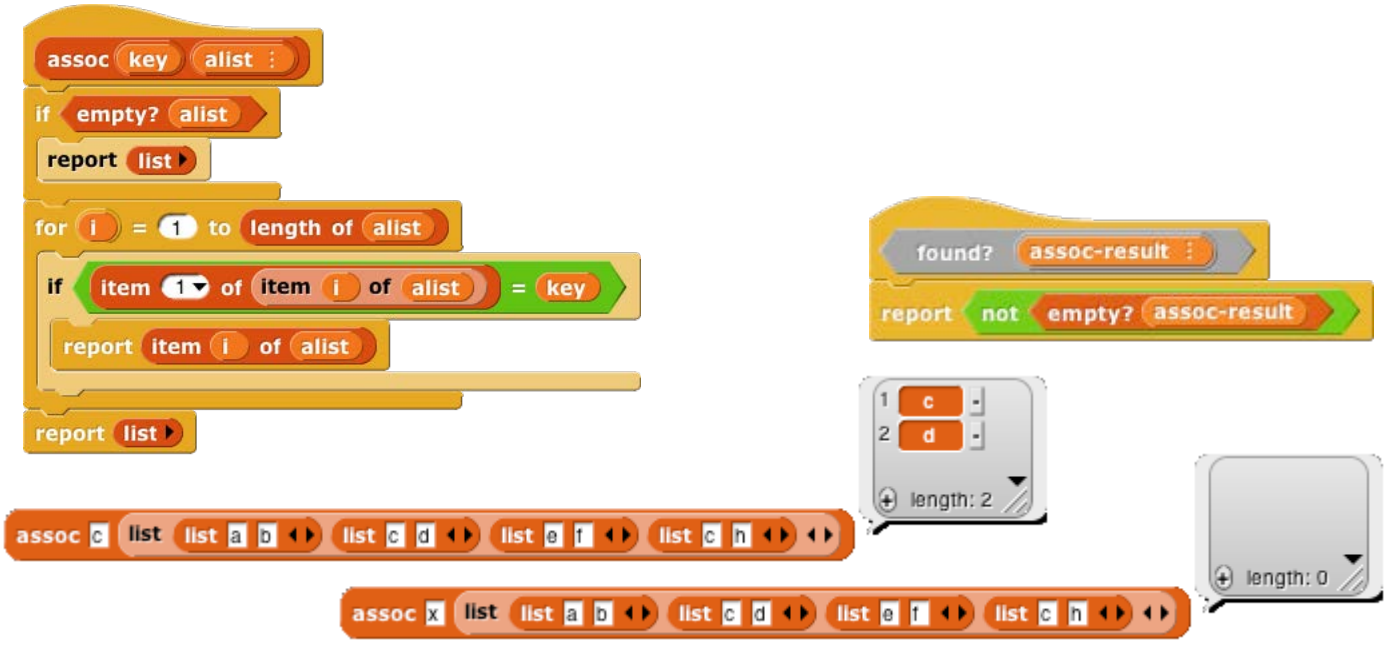


This script implements the **buzzer** class, which is a child of **counter**. Instead of having a count (a number) as a local state variable, each buzzer has a **counter** (an object) as a local state variable. The class specializes the **next** method, reporting what the counter reports unless that result is divisible by 7, in which case it reports "**buzz**." (Yeah, it should also check for a digit 7 in the number, but this code is complicated enough already.) If the message is anything other than **next**, though, such as **reset**, then the buzzer simply invokes its **counter**'s dispatch procedure. So the counter handles any message that the buzzer doesn't handle explicitly. (Note that in the non-**next** case we **call** the counter, not **ask** it something, because we want to report a method, not the value that the message reports.) So, if we **ask** a **buzzer** to **reset** to a value divisible by 7, it will end up reporting that number, not "**buzz**."

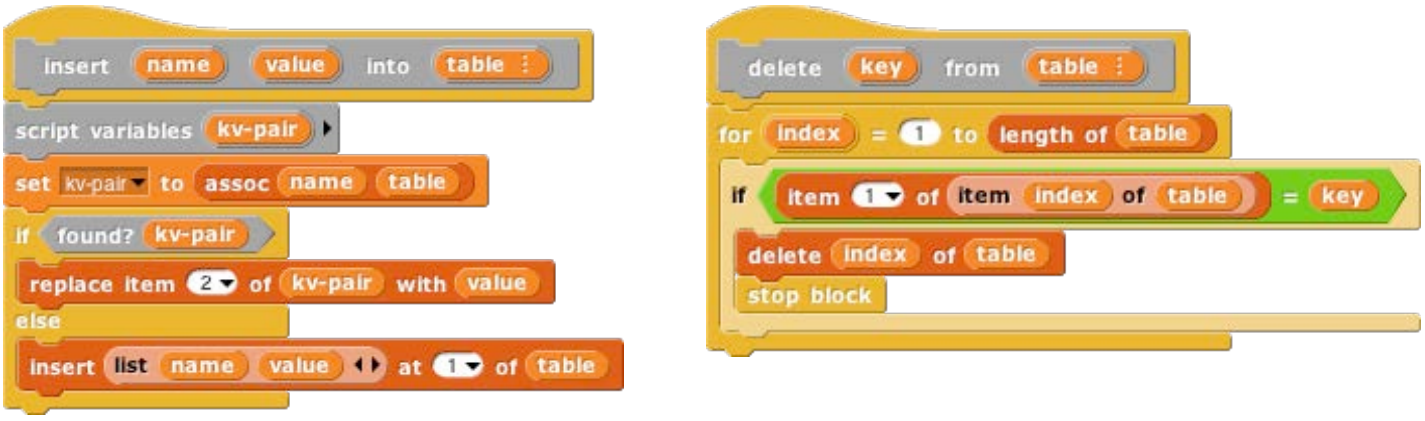
D. An Implementation of Prototyping OOP

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 okay for top-down design, but not great for experimentation. Here we sketch the implementation of a *prototyping* OOP system: You make an object, tinker with it, make clones of it, and keep tinkering. Any changes you make in the parent are inherited by its children. In effect, that first object is both the class and an instance of the class. In the implementation below, children share properties (methods and local variables) of their parent unless and until a child changes a property, at which point that child gets a private copy. (If a child wants to change something for its entire family, it must ask the parent to do it.)

Because we want to be able to create and delete properties dynamically, we won't use Snap! 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 corresponding *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:

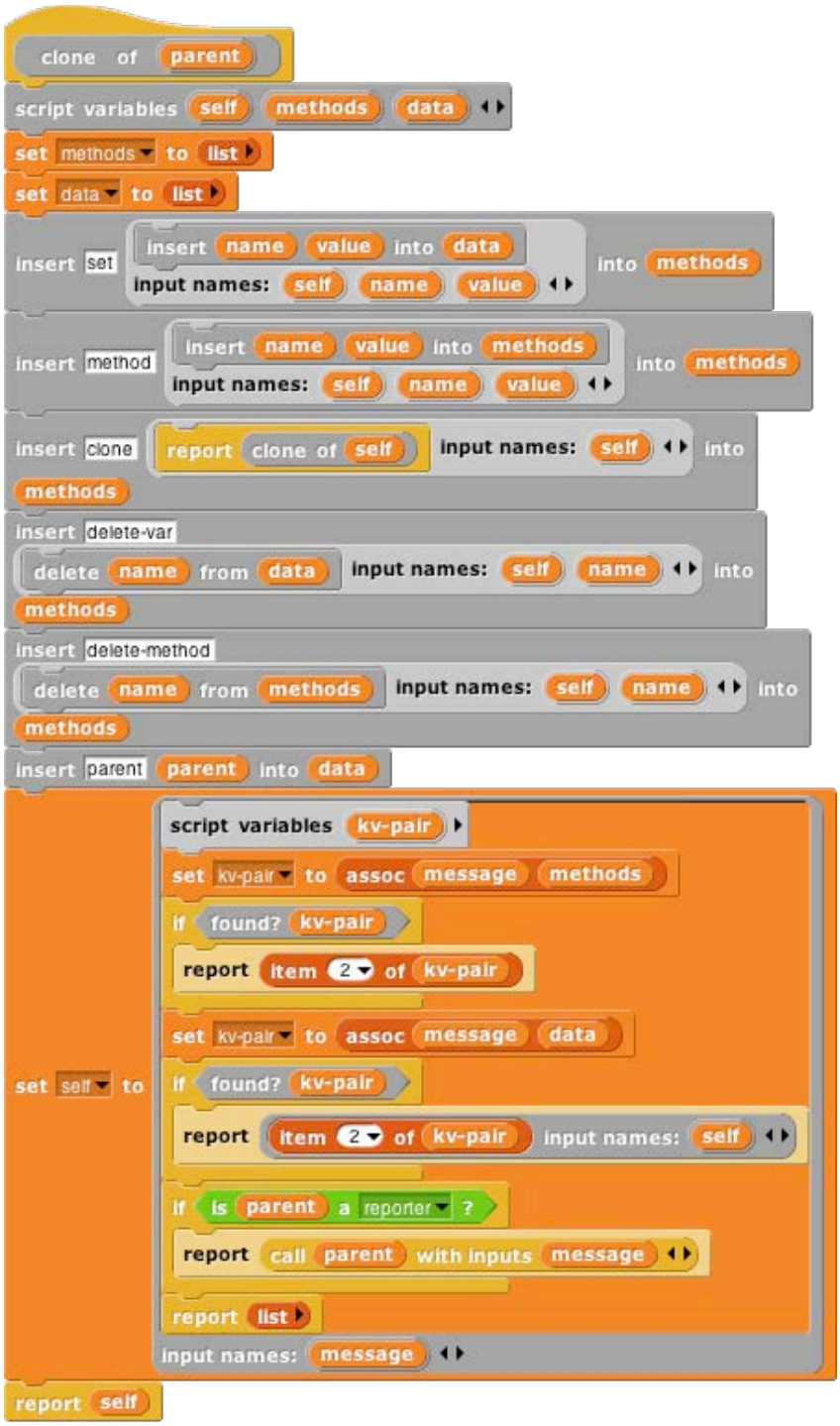


As in the class/instance version, an object is represented as a dispatch procedure that takes a message as its input and reports the corresponding method. When an object gets a message, it will first look for that keyword in its **methods** table. If it's found, the corresponding value is the method 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.

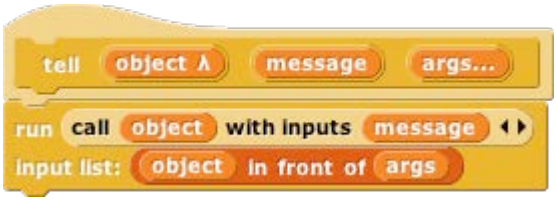
Messages can take any number of inputs, as in the class/instance system, but in the prototyping version, every method automatically gets the object to which the message was originally sent as an extra first input. We must do this so that if a method is found in the parent (or grandparent, etc.) of the 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.



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**:

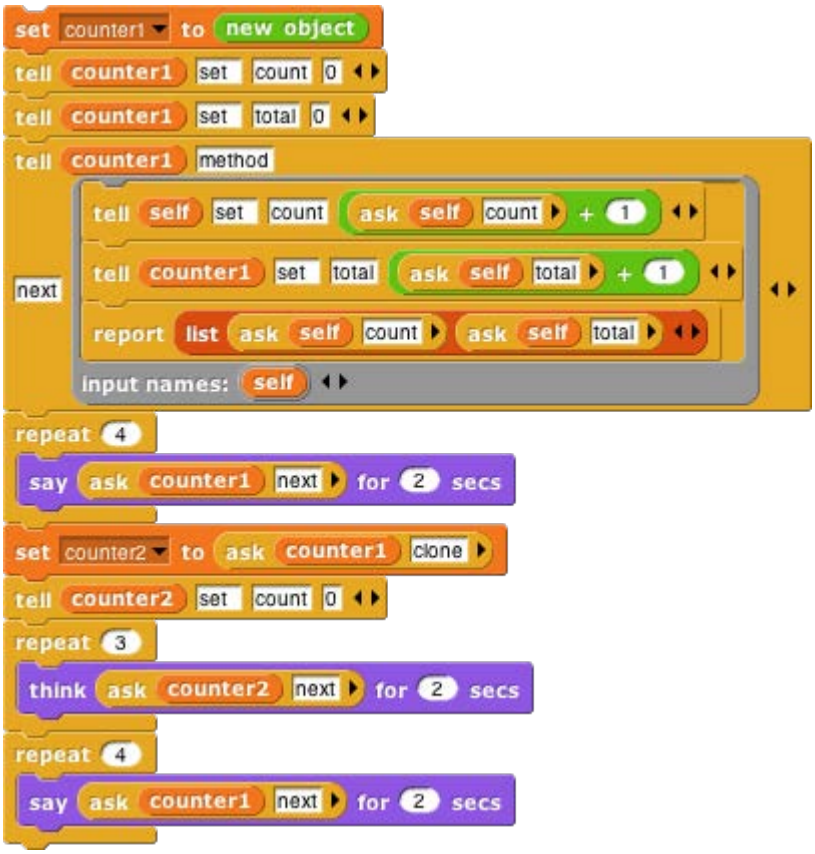
As before, we provide procedures to call an object's dispatch procedure and then call the method. But in this version, we provide the desired object as the first method input. We provide one procedure for Command methods and one for Reporter methods:



(Remember that the "**input list:**" variant of the **run** and **call** blocks is made by dragging the input expression over the arrowheads rather than over the input slot.)

The script below 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]



IX. The Outside World

The facilities discussed so far are fine for projects that take place entirely on your computer's screen. But you may want to write programs that interact with physical devices (sensors or robots) or with the World Wide Web. For these purposes Snap! provides a single primitive block:

This might not seem like enough, but in fact it can be used to build the desired capabilities.

A. The World Wide Web

The input to the **url** block is the URL (Uniform Resource Locator) of a web page. The block reports the body of the Web server's response (minus HTTP header), *without interpretation*. This means that in most cases the response is a description of the page in HTML (HyperText Markup Language) notation. Often, especially for commercial web sites, the actual information you're trying to find on the page is actually at another URL included in the reported HTML. The Web page is typically a very long text string, and so the primitive **split** block is useful to get the text in a manageable form, namely, as a list of lines:

The second input to **split** is the character to be used to separate the text string into a list of lines, or one of a set of common cases (such as **line**, which separates on carriage return and/or newline characters.

This might be a good place for a reminder that list-view watchers scroll through only 100 items at a time. The downarrow near the bottom right corner of the speech balloon in the picture presents a menu of hundred-item ranges. (This may seem unnecessary, since the scroll bar should allow for any number of items, but doing it this way makes Snap! much faster.) In table view, the entire list is included.

If you include a protocol name in the input to the **url** block (such as **http://** or **https://**), that protocol will be used. If not, the block first tries HTTPS and then, if that fails, HTTP.

A security restriction in JavaScript limits the ability of one web site to initiate communication with another site. There is an official workaround for this limitation called the CORS protocol (Cross-Origin Resource Sharing), but the target site has to allow `snap.berkeley.edu` explicitly, and of course most don't. To get around this problem, you can use third-party sites ("cors proxies") that are not limited by JavaScript and that forward your requests.

B. Hardware Devices

Another JavaScript security restriction prevents Snap! from having direct access to devices connected to your computer, such as sensors and robots. (Mobile devices such as smartphones may also have useful devices built in, such as accelerometers and GPS receivers.) The **url** block is also used to interface Snap! with these external capabilities.

The idea is that you run a separate program that both interfaces with the device and provides a local HTTP server that Snap! can use to make requests to the device. *Unlike* Snap! *itself, these programs have access to anything on your computer, so you have to trust the author of the software!* Our web site, `snap.berkeley.edu`, provides links to drivers for several devices, including, at this writing, the Lego NXT, Finch, Hummingbird, and Parallax S2 robots; the Nintendo Wiimote and Leap Motion sensors, the Arduino microcomputer, and Super-Awesome Sylvia's Water Color Bot. The same server technique can be used for access to third party software libraries, such as the speech synthesis package linked on our web site.

Most of these packages require some expertise to install; the links are to source code repositories. This situation will improve with time.

C. Date and Time

The **current** block in the Sensing palette can be used to find out the current date or time. Each call to this block reports one component of the date or time, so you will probably combine several calls, like this:

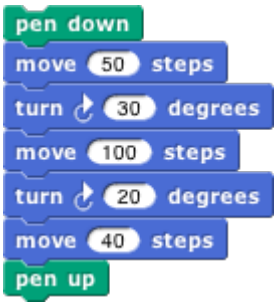
for Americans, or like this:

for Europeans.

X. Continuations

Blocks are usually used within a script. The *continuation* of a block within a particular script is the part of the computation that remains to be completed after the block does its job. A continuation can be represented as a ringed script. Continuations are always part of the interpretation of any program in any language, but usually these continuations are implicit in the data structures of the language interpreter or compiler. Making continuations explicit is an advanced but versatile programming technique that allows users to create control structures such as nonlocal exit and multithreading.

In the simplest case, the continuation of a command block may just be the part of the script after the block. For example, in the script



the continuation of the **move 100 steps** block is

But some situations are more complicated. For example, what is the continuation of **move 100 steps** in the following script?

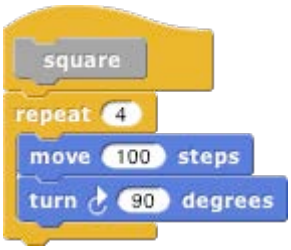
That's a trick question; the **move** block is run four times, and it has a different continuation each time. The first time, its continuation is

Note that there is no **repeat 3** block in the actual script, but the continuation has to represent the fact that there are three more times through the loop to go. The fourth time, the continuation is just

What counts is not what's physically below the block in the script, but what computational work remains to be done.

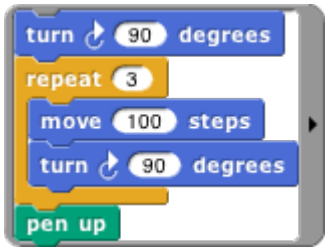
(This is a situation in which visible code may be a little misleading. We have to put a **repeat 3** block in the *picture* of the continuation, but the actual continuation is made from the evaluator's internal bookkeeping of where it's up to in a script. So it's really the original script plus some extra information. But the pictures here do correctly represent what work the process still has left to do.)

When a block is used inside a custom block, its continuation may include parts of more than one script. For example, if we make a custom **square** block



and then use that block in a script:

then the continuation of the first use of **move 100 steps** is



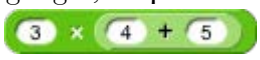
in which part comes from inside the **square** block and part comes from the call to **square**. Nevertheless, ordinarily when we *display* a continuation we show only the part within the current script.

The continuation of a command block, as we've seen, is a simple script with no input slots. But the continuation of a *reporter* block has to do something with the value reported by the block, so it takes that value as input. For example, in the script

the continuation of the **3+4** block is

Of course the name **result** in that picture is arbitrary; any name could be used, or no name at all by using the empty-slot notation for input substitution.

A. Continuation Passing Style

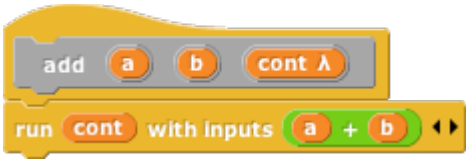
Like all programming languages, Snap! evaluates compositions of nested reporters from the inside out. For example, in the expression 

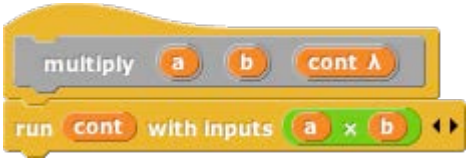
Snap! first adds 4 and 5, then multiplies 3 by that sum. This often means that the order in which the operations are done is backwards from the order in which they appear in the expression: When reading the above expression you say "times" before you say "plus." In English, instead of saying "three times four plus five," which actually makes the order of operations ambiguous, you could say, "take the sum of four and five, and then take the product of three and that sum." This sounds more awkward, but it has the virtue of putting the operations in the order in which they're actually performed.

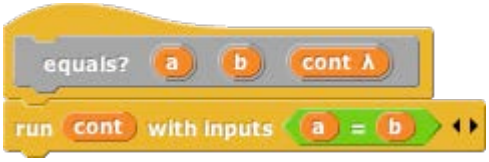
That may seem like overkill in a simple expression, but suppose you're trying to convey the expression

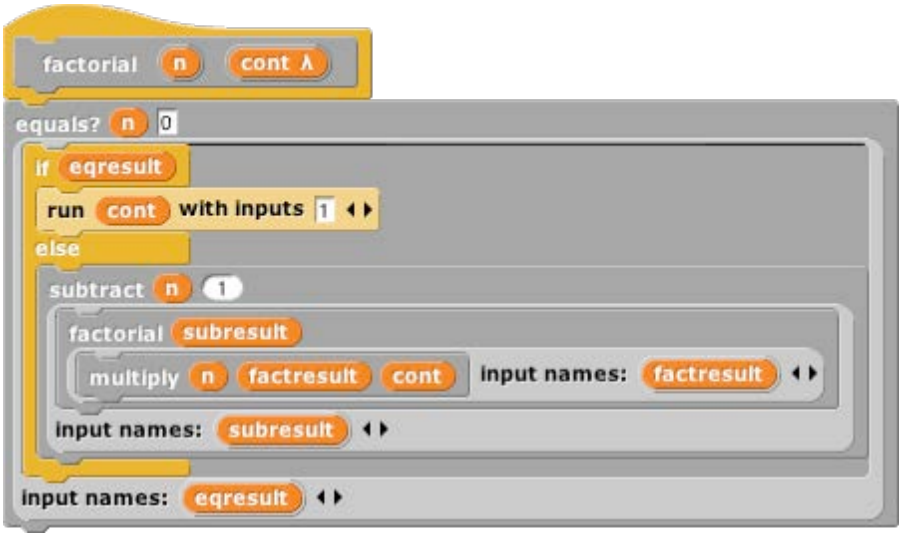
to a friend over the phone. If you say "factorial of three times factorial of two plus two plus five" you might mean any of these:

Wouldn't it be better to say, "Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result"? We can do a similar reordering of an expression if we first define versions of all the reporters that take their continuation as an explicit input. In the following picture, notice that the new blocks are *commands*, not reporters.



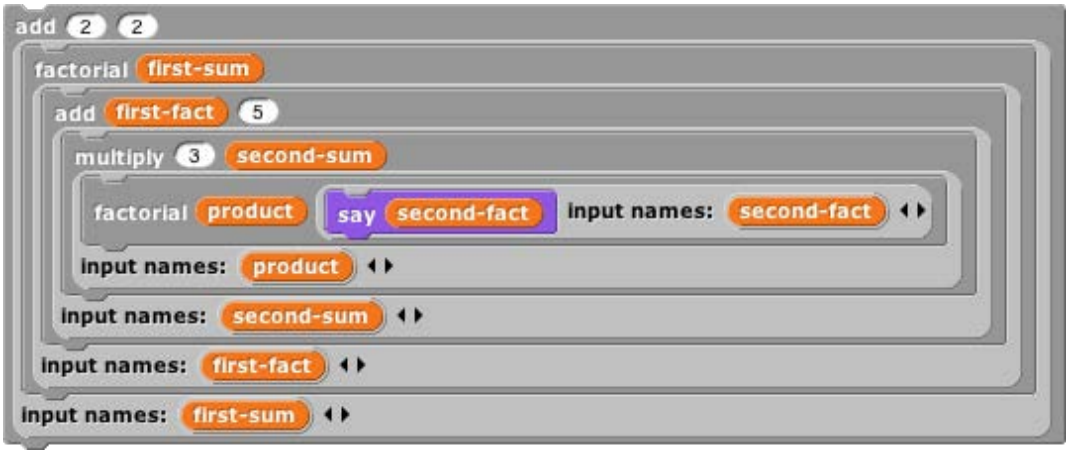






We can check that these blocks give the results we want:

The original expression can now be represented as



If you read this top to bottom, don't you get "Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result"? Just what we wanted! This way of working, in which every block is a command that takes a continuation as one of its inputs, is called *continuation-passing style (CPS)*. Okay, it looks horrible, but it has subtle virtues. One of them is that each script is just one block long (with the rest of the work buried in the continuation given to that one block), so each block doesn't have to remember what else to do—in the vocabulary of this section, the (implicit) continuation of each block is empty. Instead of the usual picture of recursion, with a bunch of little people all waiting for each other, with CPS what happens is that each little person hands off the problem to the next one and goes to the beach, so there's only one active little person at a time. In this example, we start with Alfred, an **add** specialist, who computes the value 4 and then hands off the rest of the problem to Francine, a **factorial** specialist. She computes the value 24, then hands the problem off to Anne, another **add** specialist, who computes 29. And so on, until finally Sam, a **say** specialist, says the value $2.107757298379527\times10^{132}$, which is a very large number!

Go back to the definitions of these blocks. The ones, such as **add**, that correspond to primitive reporters are simple; they just call the reporter and then call their continuation with its result. But the definition of **factorial** is more interesting. It doesn't just call our original **factorial** reporter and send the result to its continuation. CPS is used inside **factorial** too! It says, "See if my input is zero. Send the (true or false) result to **if**. If the result is **true**, then call my continuation with the value 1. Otherwise, subtract 1 from my input. Send the result of that to **factorial**, with a continuation that multiplies the smaller number's factorial by my original input. Finally, call my continuation with the product." You can use CPS to unwind even the most complicated branched recursions.

By the way, I cheated a bit above. The **if/else** block should also use CPS; it should take one true/false input and *two continuations*. It will go to one or the other continuation depending on the value of its input. But in fact the C-shaped blocks (or E-shaped, like **if/else**) are really using CPS in the first place, because they implicitly wrap rings around the sub-scripts within their branches. See if you can make an explicitly CPS **if/else** block.

B. Call/Run w/Continuation

To use explicit continuation passing style, we had to define special versions of all the reporters, ***add*** and so on. Snap! provides a primitive mechanism for capturing continuations when we need to, without using continuation passing throughout a project.

Here's the classic example. We want to write a recursive block that takes a list of numbers as input, and reports the product of all the numbers:

But we can improve the efficiency of this block, in the case of a list that includes a zero; as soon as we see the zero, we know that the entire product is zero.

But this is not as efficient as it might seem. Consider, as an example, the list 1,2,3,0,4,5. We find the zero on the third recursive call (the fourth call altogether), as the first item of the sublist 0,4,5. What is the continuation of the ***report 0*** block? It's

Even though we already know that ***result*** is zero, we're going to do three unnecessary multiplications while unwinding the recursive calls.

We can improve upon this by capturing the continuation of the top-level call to ***product***:

The 

block takes as its input a one-input script, as shown in the **product** example. It calls that script with *the continuation of the* ***call-with-continuation*** *block itself* as its input. In this case, that continuation is

reporting to whichever script called **product**. If the input list doesn't include a zero, then nothing is ever done with that continuation, and this version works just like the original **product**. But if the input list is 1,2,3,0,4,5, then three recursive calls are made, the zero is seen, and **product-helper** *runs the continuation,* with an input of 0. The continuation immediately reports that 0 to the caller of **product**, *without* unwinding all the recursive calls and without the unnecessary multiplications.

I could have written **product** a little more simply using a Reporter ring instead of a Command ring:

but it's customary to use a script to represent the input to **call w/continuation** because very often that input takes the form

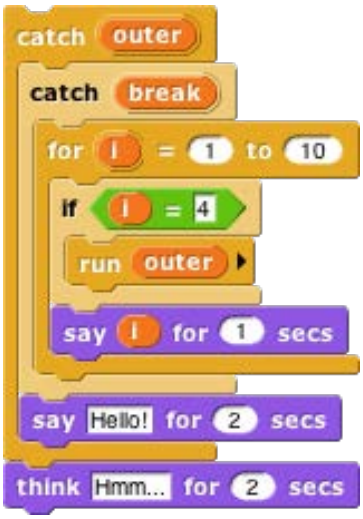
so that the continuation is saved permanently and can be called from anywhere in the project. That's why the input slot in **call w/continuation** has a Command ring rather than a Reporter ring.

First class continuations are an experimental feature in Snap*!* and there are many known limitations in it. One is that the display of reporter continuations shows only the single block in which the **call w/continuation** is an input.

Nonlocal exit

Many programming languages have a **break** command that can be used inside a looping construct such as **repeat** to end the repetition early. Using first class continuations, we can generalize this mechanism to allow nonlocal exit even within a block called from inside a loop, or through several levels of nested loops:

The upvar **break** has as its value a continuation that can be called from anywhere in the program to jump immediately to whatever comes after the **catch** block in its script. Here's an example with two nested invocations of **catch**, with the upvar renamed in the outer one:



As shown, this will say 1, then 2, then 3, then exit both nested **catch**es and think "Hmm." If in the **run** block the variable **break** is used instead of **outer**, then the script will say 1, 2, 3, and "Hello!" before thinking "Hmm."

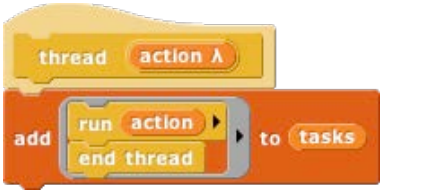
There are corresponding **catch** and **throw** blocks for reporters. The **catch** block is a reporter that takes an expression as input instead of a C-shaped slot. But the **throw** block is a command; it doesn't report a value to its own continuation, but instead reports a value (which it takes as an additional input, in addition to the **catch** tag) to *the corresponding* **catch** *block*'s continuation:

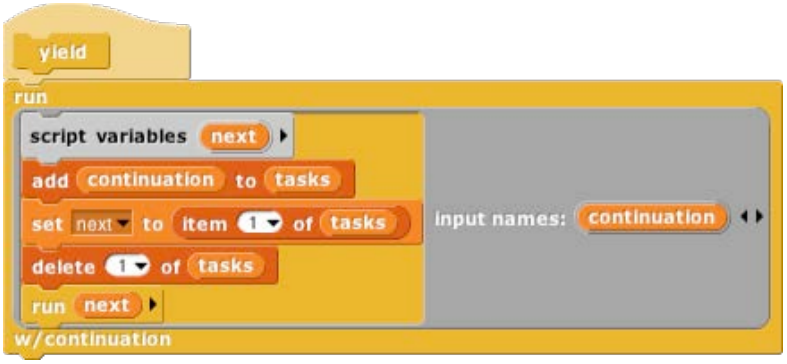
Without the **throw**, the inner **call** reports 5, the **+** block reports 8, so the **catch** block reports 8, and the **×** block reports 80. With the **throw**, the inner **call** doesn't report at all, and neither does the **+** block. The **throw** block's input of 20 becomes the value reported by the **catch** block, and the **×** block multiplies 10 and 20.

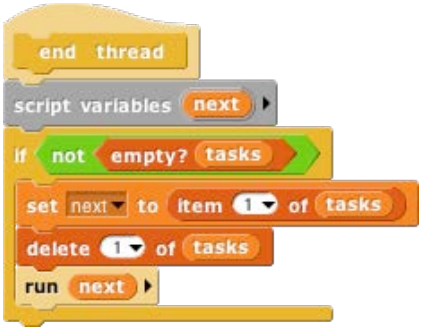
Creating a Thread System

Snap! can be running several scripts at once, within a single sprite and across many sprites. If you only have one computer, how can it do many things at once? The answer is that only one is actually running at any moment, but Snap! switches its attention from one script to another frequently. At the bottom of every looping block (**repeat**, **repeat until**, **forever**), there is an implicit "yield" command, which remembers where the current script is up to, and switches to some other script, each in turn. At the end of every script is an implicit "end thread" command (a *thread* is the technical term for the process of running a script), which switches to another script without remembering the old one.

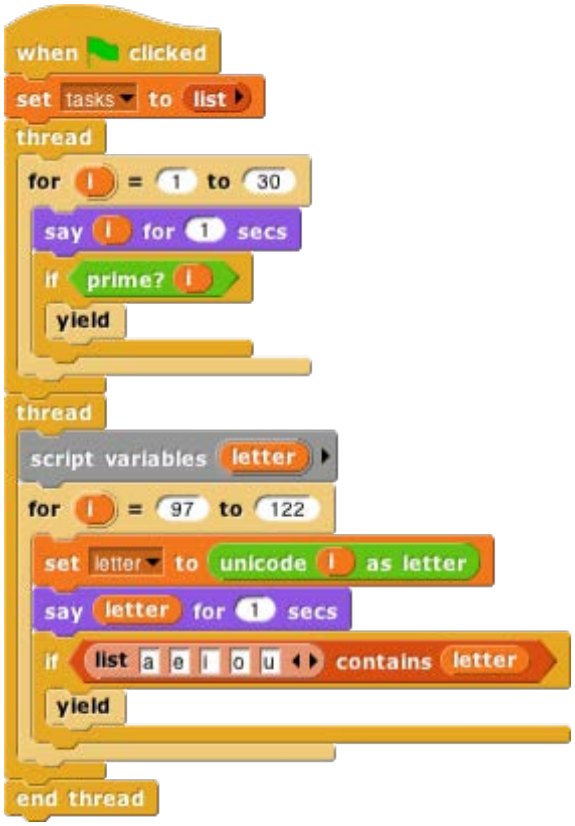
Since this all happens automatically, there is generally no need for the user to think about threads. But, just to show that this, too, is not magic, here is an implementation of a simple thread system. It uses a global variable named **tasks** that initially contains an empty list. Each use of the C-shaped **thread** block adds a continuation (the ringed script) to the list. The **yield** block uses **run w/continuation** to create a continuation for a partly done thread, adds it to the task list, and then runs the first waiting task. The **end thread** block (which is automatically added at the end of every thread's script by the **thread** block) just runs the next waiting task.







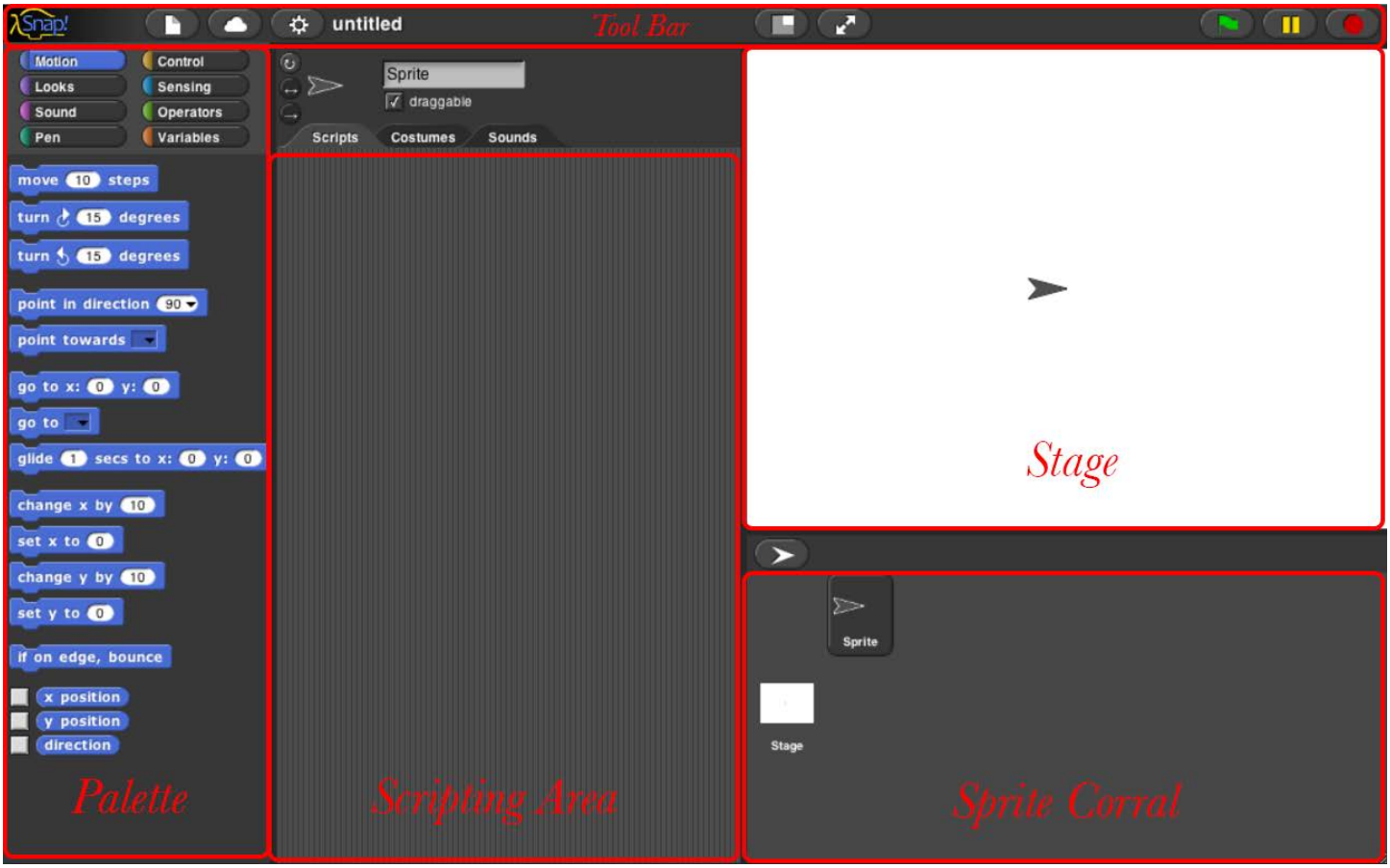
Here is a sample script using the thread system. One thread **say**s numbers; the other **say**s letters. The number thread yields after every prime number, while the letter thread yields after every vowel. So the sequence of speech balloons is 1,2,a,3,b,c,d,e,4,5,f,g,h,i,6,7,j,k,l,m,n,o,8,9,10,11, p,q,r,s,t,u,12,13,v,w,x,y,z,14,15,16,17,18,…30.



If we wanted this to behave exactly like Snap!'s own threads, we'd define new versions of **repeat** and so on that run **yield** after each repetition.

XI. User Interface Elements

In this section we describe in detail the various buttons, menus, and other clickable elements of the Snap! user interface. Here again is the map of the Snap! window:

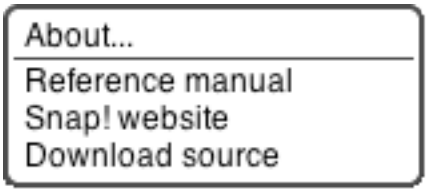


A. Tool Bar Features

Holding down the Shift key while clicking on any of the menu buttons gives access to an extended menu with options, shown in red, that are experimental or for use by the developers. We're not listing those extra options here because they change frequently and you shouldn't rely on them. But they're not secrets.

The Snap! Logo Menu

The Snap! logo at the left end of the tool bar is clickable. It shows a menu of options about Snap! itself:



The About option displays information about Snap! itself, including version numbers for the source modules, the implementors, and the license (AGPL: you can do anything with it except create proprietary versions, basically).

The **Reference manual** option downloads a copy of the latest revision of this manual in PDF format.

The **Snap! website** option opens a browser window pointing to `snap.berkeley.edu`, the web site for Snap!.

The **Download source** option opens a browser window displaying the Github repository of the source files for Snap!. At the bottom of the page are links to download the latest official release. Or you can navigate around the site to find the current development version. You can read the code to learn how Snap! is implemented, host a copy on your own computer (this is one way to keep working while on an airplane), or make a modified version with customized features. (However, access to cloud accounts is limited to the official version hosted at Berkeley.)

The File Menu

The file icon shows a menu mostly about saving and loading projects:

The **Project notes** option opens a window in which you can type notes about the project: How to use it, what it does, whose project you modified to create it, if any, what other sources of ideas you used, or any other information about the project. This text is saved with the project, and is useful if you share it with other users.

The **New** option starts a new, empty project. Any project you were working on before disappears, so you are asked to confirm that this is really what you want. (It disappears only from the current working Snap! window; you should save the current project, if you want to keep it, before using **New**.)

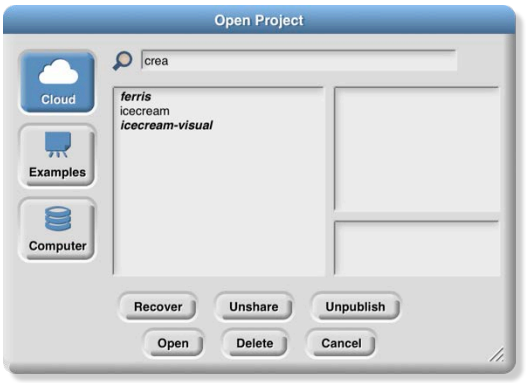
Note the **^N** at the end of the line. This indicates that you can type control-N as a shortcut for this menu item. Alas, this is not the case in every browser. Some Mac browsers require command-N (**⌘N**) instead, while others open a new browser window instead of a new project. You'll have to experiment. In general, the keyboard shortcuts in Snap! are the standard ones you expect in other software.

The ***Open…*** option shows a project open dialog box in which you can choose a project to open:

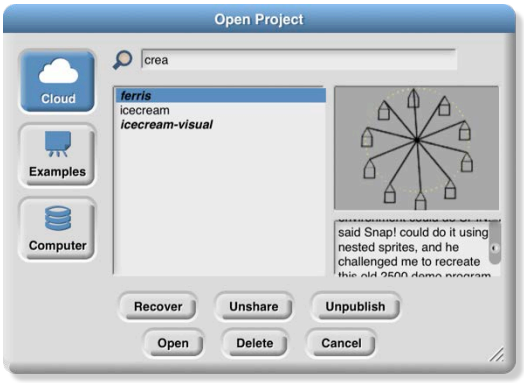
In this dialog, the three large buttons at the left select a source of projects: ***Cloud*** means your Snap! account's cloud storage. **Examples** means a collection of sample projects we provide. ***Computer*** is for projects saved on your own computer; when you click it, this dialog is replaced with your computer's system dialog for opening

files. The text box to the right of those buttons is an alphabetical listing of projects from that source; selecting a project by clicking shows its thumbnail (a picture of the stage when it was saved) and its project notes at the right.

The search bar at the top can be used to find a project by name or text in the project notes. So in this example:



I was looking for my ice cream projects and typed "crea" in the search bar, then wondered why "ferris" matched. But then when I clicked on ferris I saw this:



My search matched the word "re*crea*te" in the project notes.

The six buttons at the bottom select an action to perform on the selected project. In the top row, **Recover** looks in your cloud account for older versions of the chosen project. ***If your project is damaged, don't keep saving broken versions! Use Recover first thing.*** You will see a list of saved versions; choose one to open it. Typically, you'll see the most recent version before the last save, and the newest version saved before today. Then come buttons **Share/Unshare** and **Publish/Unpublish**. The labelling of the buttons depends on your project's publication status. If a project is neither shared nor published (the ones in lightface type in the project list), it is private and nobody can see it except you, its owner. If it is shared (**boldface** in the project list), then when you open it you'll see a URL like this one:

https://snap.berkeley.edu/snapsource/snap.html#present:Username=bh&ProjectName=count%20change

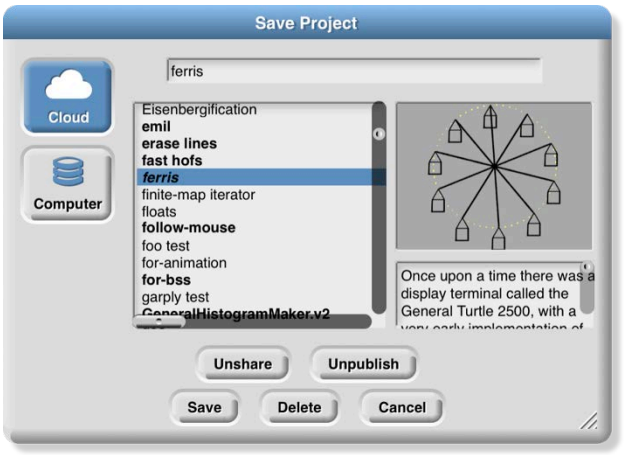
but with your username and project name. ("**%20**" in the project name represents a space, which can't be part of a URL.) Anyone who knows this URL can see your project. Finally, if your project is published (***bold italic*** in the list), then your project is shown on the **Snap!** web site for all the world to see. (In all of these cases, you are the only one who can *write* to (save) your project. If another user saves it, a separate copy will be saved in that user's account. Projects remember the history of who created the original version and any other "remix" versions along the way.

In the second row, the first button, ***Open***, loads the project into **Snap!** and closes the dialog box. The next button (if **Cloud** is the source) is **Delete**, and if clicked it deletes the selected project. Finally, the **Cancel** button

closes the dialog box without opening a project. (It does not undo any sharing, unsharing, or deletion you've done.)

Back to the File menu, the **Save** menu option saves the project to the same source and same name that was used when opening the project. (If you opened another user's shared project or an example project, the project will be saved to your own cloud account. You must be logged in to save to the cloud.)

The **Save as…** menu option opens a dialog box in which you can specify where to save the project:



This is much like the **Open** dialog, except for the horizontal text box at the top, into which you type a name for the project. You can also publish, unpublish, share, unshare, and delete projects from here. There is no **Recover** button.

The **Import…** menu option is for bringing some external resource into the current project, or it can load an entirely separate project, from your local disk. You can import costumes (any picture format that your browser supports), sounds (again, any format supported by your browser), and block libraries or sprites (XML format, previously exported from Snap! itself). Imported costumes and sounds will belong to the currently selected sprite; imported blocks are global (for all sprites). Using the **Import** option is equivalent to dragging the file from your desktop onto the Snap! window.

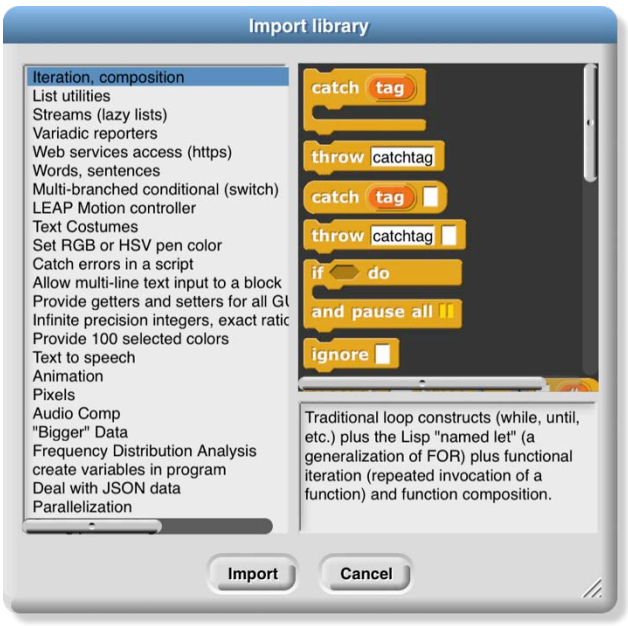
Depending on your browser, the **Export project…** option either directly saves to your disk or opens a new browser tab containing your complete project in XML notation (a plain text format). You can then use the browser's Save feature to save the project as an XML file, which should be named `something.xml` so that Snap! will recognize it as a project when you later drag it onto a Snap! window. This is an alternative to saving the project to your cloud account: keeping it on your own computer. It is equivalent to choosing **Computer** from the Save dialog described earlier.

The **Export blocks…** option is used to create a block library. It presents a list of all the global (for all sprites) blocks in your project, and lets you select which to export. It then opens a browser tab with those blocks in XML format, or stores directly to your local disk, as with the **Export project** option. Block libraries can be imported with the **Import** option or by dragging the file onto the Snap! window. This option is shown only if you have defined custom blocks.

The **Unused blocks…** option presents a listing of all the global blocks in your project that aren't used anywhere, and offers to delete them. As with Export blocks, you can choose a subset to delete with checkboxes. This option is shown only if you have defined custom blocks.

The **Export summary…** option creates a web page, in HTML, with all of the information about your project: its name, its project notes, a picture of what's on its stage, definitions of global blocks, and then per-sprite information: name, wardrobe (list of costumes), and local variables and block definitions. The page can be converted to PDF by the browser; it's intended to meet the documentation requirements of the Advanced Placement Computer Science Principles create task.

The **Libraries…** option presents a menu of useful, optional block libraries:



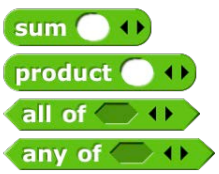
When you click on the one-line description of a library, you are shown the actual blocks in the library and a longer explanation of its purpose. You can browse the libraries to find one that will satisfy your needs.

The libraries and their contents may change, but as of this writing the list library has these blocks:

Reverse reports a list with the items of the input list in reverse order. **Remove duplicates from** reports a list in which no two items are equal. The **sort** block takes a list and a two-input comparison predicate, such as **<**, and reports a list with the items sorted according to that comparison. The **assoc** block is for looking up a key in an *association list:* a list of two-item lists. In each two-item list, the first is a *key* and the second is a *value.* The inputs are a key and an association list; the block reports the first key-value pair whose key is equal to the input key.

The other three blocks are versions of the list tools that provide a **#** variable containing the position in the input list of the currently considered item. (This is partly obsoleted by the three-input HOF feature, page 39.) This version of **map** also allows multiple list inputs, in which case the mapping function must take as many inputs as there are lists; it will be called with all the first items, all the second items, and so on.

The variadic library has these blocks:



These are versions of the associative operators **+**. **×**, **and**, and **or** that take any number of inputs instead of exactly two inputs. As with any variadic input, you can also drop a list of values onto the arrowheads instead of providing the inputs one at a time.

The iteration library has these blocks:

Catch and **throw** provide a nonlocal exit facility. You can drag the **tag** from a **catch** block to a **throw** inside its C-slot, and the **throw** will then jump directly out to the matching catch without doing anything in between.

If *do and pause all* is for setting a breakpoint while debugging code. The idea is to put **show variable** blocks for local variables in the C-slot; the watchers will be deleted when the user continues from the pause.

Ignore is used when you need to call a reporter but you don't care about the value it reports. (For example, you are writing a script to time how long the reporter takes.)

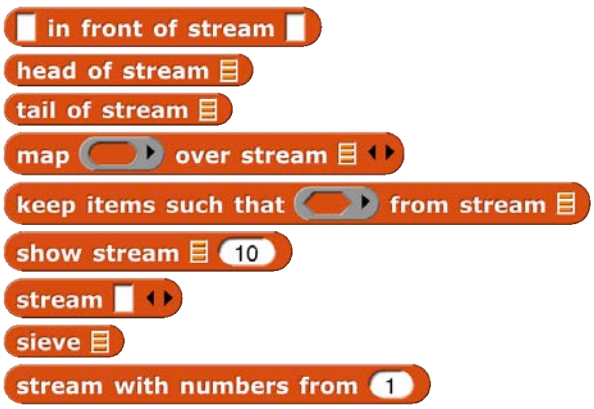
The **cascade** blocks take an initial value and call a function repeatedly on that value, $f(f(f(f\ldots(x))))$.

The **compose** block takes two functions and reports the function $f(g(x))$.

The first three **repeat** blocks are variants of the primitive **repeat until** block, giving all four combinations of whether the first test happens before or after the first repetition, and whether the condition must be true or false to continue repeating. The last **repeat** block is like the **repeat** primitive, but makes the number of repetitions so far available to the repeated script. The next two blocks are variations on **for**: the first allows an explicit step instead of using ±1, and the second allows any values, not just numbers; inside the script you say

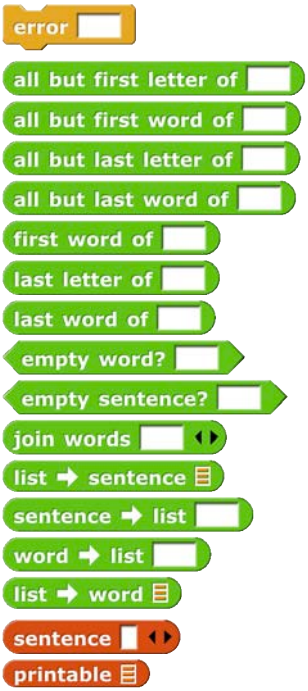
replacing the grey block in the picture with an expression to give the next desired value for the loop index.

The stream library has these blocks:



Streams are a special kind of list whose items are not computed until they are needed. This makes certain computations more efficient, and also allows the creation of lists with infinitely many items, such as a list of all the positive integers. The first five blocks are stream versions of the list blocks **in front of**, **item 1 of**, **all but first of**, **map**, and **keep**. **Show stream** takes a stream and a number as inputs, and reports an ordinary list of the first *n* items of the stream. **Sieve** is an example block that takes as input the stream of integers starting with 2 and reports the stream of all the prime numbers. **Stream** is like the primitive **list**; it makes a finite stream from explicit items. **Stream with numbers from** is like the **numbers from** block for lists, except that there is no endpoint; it reports an infinite stream of numbers.

The word and sentence library has these blocks:



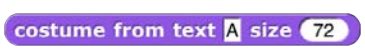
This library has the goal of recreating the Logo approach to handling text: A text isn't best viewed as a string of characters, but rather as a *sentence*, made of *words*, each of which is a string of *letters*. With a few specialized exceptions, this is why people put text into computers: The text is sentences of natural (i.e., human) language, and the emphasis is on words as constitutive of sentences. You barely notice the letters of the words, and you don't notice the spaces between them at all, unless you're proofreading. (Even then: Proofreading is *difficult*, because you see what you expect to see, what will make the text make sense, rather than the misspelling in front of your eyes.) Internally, Logo stores a sentence as a list of words, and a word as a string of letters.

Inexplicably, the designers of Scratch chose to abandon that tradition, and to focus on the representation of text as a string of characters. The one vestige of the Logo tradition from which Scratch developed is the block named **letter (1) of (world)**, rather than **character (1) of (world)**. Snap! inherits its text handling from Scratch.

In Logo, the visual representation of a sentence (a list of words) looks like a natural language sentence: a string of words with spaces between them. In Snap!, the visual representation of a list looks nothing at all like natural language. On the other hand, representing a sentence as a string means that the program must continually re-parse the text on every operation, looking for spaces, treating multiple consecutive spaces as one, and so on. Also, it's more convenient to treat a sentence as a list of words rather than a string of words because in the former case you can use the higher order functions **map**, **keep**, and **combine** on them.This library attempts to be agnostic as to the internal representation of sentences. The sentence selectors accept any combination of lists and strings; there are two sentence constructors, one to make a string (**join words**) and one to make a list (**sentence**).

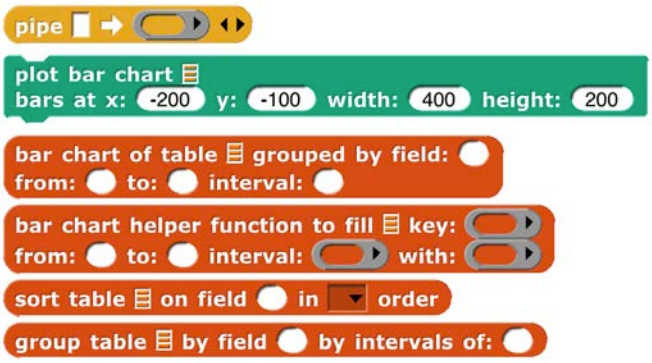
The selector names come from Logo, and should be self-explanatory. However, because in a block language you don't have to type the block name, instead of the terse **butfirst** or the cryptic **bf** we spell out "all but first of" and include "word" or "sentence" to indicate the intended domain. There's no **first letter of** block because **letter 1 of** serves that need. **Join words** (the sentence-as-string constructor) is like the primitive **join** except that it puts a space in the reported value between each of the inputs. **Sentence** (the List-colored sentence-as-list constructor) accepts any number of inputs, which can be words, sentences-as-lists, or sentences-as-strings. (If inputs are lists of lists, only one level of flattening is done.) **Sentence** reports a list of words; there will be no empty words or words containing spaces. The four blocks with right-arrows in their names convert back and forth between text strings (words or sentences) and lists. (Splitting a word into a list of letters is unusual unless you're a linguist investigating orthography.) **Printable** takes a list (including a deep list) of words as input and reports a text string in which parentheses are used to show the structure, as in Lisp/Scheme.

The text costume library has only one block:



It reports a costume that can be used with the **switch to costume** block to make a button: Snap!

The bar charts library has these blocks:



Bar chart takes a table (typically from a CSV data set) as input and reports a summary of the table grouped by the field in the specified column number. The remaining three inputs are used only if the field values are numbers, in which case they can be grouped into buckets (e.g., decades, centuries, etc.). Those inputs specify the smallest and largest values of interest and, most importantly, the width of a bucket (10 for decades, 100 for centuries). If the field isn't numeric, leave these three inputs empty or set them to zero. Each string value of the field is its own bucket, and they appear sorted alphabetically.

Bar chart reports a new table with three columns. The first column contains the bucket name or smallest number. The second column contains a nonnegative integer that says how many records in the input table fall into this bucket. The third column is a subtable containing the actual records from the original table that fall into the bucket. **Plot bar chart** takes the table reported by **bar chart** and graphs it on the stage, with axes labelled appropriately. The remaining blocks are helpers for those.

If your buckets aren't of constant width, or you want to group by some function of more than one field, load the "Frequency Distribution Analysis" library instead.

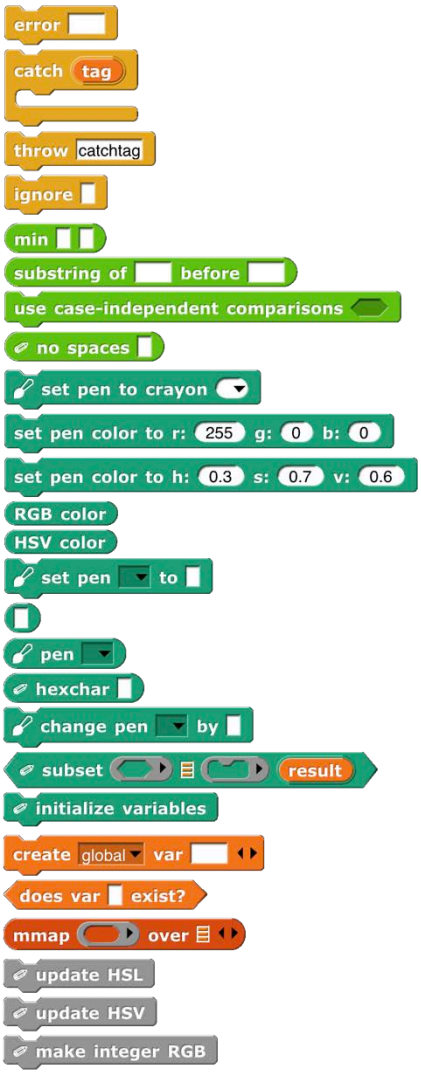
The multiple-branch conditional library has these blocks:

The **catch** and **throw** blocks duplicate ones in the iteration library, and are included because they are used to implement the others. The **cases** block sets up a multi-branch conditional, similar to **cond** in Lisp or **switch** in C-family languages. The first branch is built into the **cases** block; it consists of a Boolean test in the first hexagonal slot and an action script, in the C-slot, to be run if the test reports **true**. The remaining branches go in the variadic hexagonal input at the end; each branch consists of an **else if** block, which includes the Boolean test and the corresponding action script, except possibly for the last branch, which can use the unconditional **else** block. As in other languages, once a branch succeeds, no other branches are tested.

The web services library has these blocks:

The first block is a generalization of the primitive **url** block, allowing more control over the various options in web requests: GET, POST, PUT, and DELETE, and fine control over the content of the message sent to the server. **Current location** reports your latitude and longitude. The **key:value:** block is just a constructor for an abstract data type used with the other blocks.

The colors library has these blocks:



It is intended as a more powerful replacement for the primitive ***set pen*** block, including HSL color specification as a better alternative to the HSV that Snap! inherits from JavaScript; a "fair hue" scale that compensates for the eye's grouping a wide range of light frequencies as green while labelling mere slivers as orange or yellow; the X11/CSS standard color names; RGB in hexadecimal; a linear color scale (as in the old days) based on fair hues and including shades (darker colors) and grayscale; and incorporating the RGB library (immediately following this) and the crayon library (page 95).

Here is the normal hue scale, for reference:

Here is the fair hue scale:

(Besides equalizing the space for each spectral color, brown has been promoted to an official color. Magenta isn't a spectral color either!)

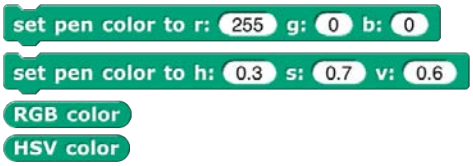
Here is the linear color scale:

(The picture is wider so that spectral colors line up with the fair hue scale.)

And here are the 100 crayons:

The four blocks whose pictures start with are meant for users: ***set pen to***, ***set pen to crayon***, ***change pen by***, and the ***pen*** reporter. The ones starting with  are internal helpers. The ones with neither decoration are imported from other libraries. See Appendix A (page 122) for more information.

The pen color library has these blocks:



The first two are commands to set the pen color; the last two report the pen color. The two supported color space representations are RGB (red-green-blue), in which each color intensity is a number between 0 and 255, and HSV (hue-saturation-value), in which each dimension is a fraction between 0 and 1. The reporters report three-item lists.

The error catching library has these blocks:

The **safely try** block allows you to handle errors that happen when your program is run within the program, instead of stopping the script with a red halo and an obscure error message. The block runs the script in its first C-slot. If it finishes without an error, nothing else happens. But if an error happens, the code in the second C-slot is run. While that second script is running, the variable error contains the text of the error message that would have been displayed if you weren't catching the error. The **error** block is sort of the opposite: it lets your program *generate* an error message, which will be displayed with a red halo unless it is caught by **safely try**. The **let** block is used in implementing the library; it's like a **script variables** followed by a **set**.

The system getter/setter library has these blocks:

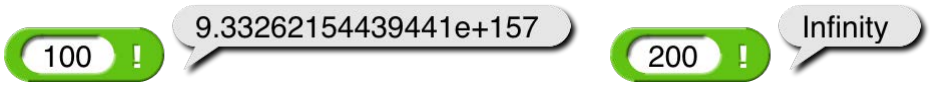
The purpose of this library is to allow program access to the settings controlled by user interface elements, such as the settings menu. The **setting** block reports a setting; the **set flag** block sets yes-or-no options that have checkboxes in the user interface, while the **set value** block controls settings with numeric or text values, such as project name.

Certain settings are ordinarily remembered on a per-user basis, such as the "zoom blocks" value. But when these settings are changed by this library, the change is in effect only while the project using the library is loaded. No permanent changes are made.

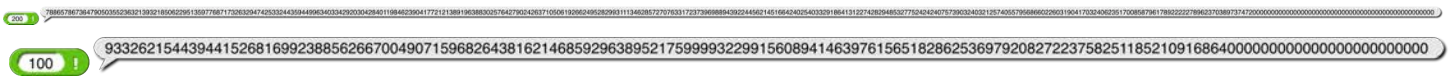
The infinite precision integer library has these blocks:

The **USE BIGNUMS** block takes a Boolean input, to turn the infinite precision feature on or off. When on, all of the arithmetic operators are redefined to accept and report integers of any number of digits (limited only by the memory of your computer) and, in fact, the entire Scheme numeric tower, with exact rationals and with complex numbers. The **Scheme number** block has a list of functions applicable to Scheme numbers, including subtype predicates such as **rational?** and **infinite?**, and selectors such as **numerator** and **real-part**.

The **!** block computes the factorial function, useful to test whether bignums are turned on. Without bignums:

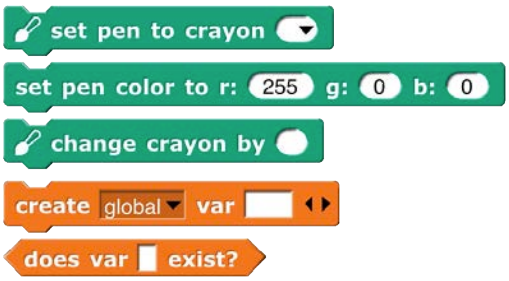


With bignums:



The 375-digit value of 200! isn't readable on this page, but if you shift-right-click on the block and choose "script pic with result," you can open the resulting picture in a browser window and scroll through it. (These values end with a bunch of zero digits. That's not roundoff error; the prime factors of 100! and 200! include many copies of 2 and 5.) The block with no name is a way to enter things like **3/4** and **4+7i** into numeric input slots by converting the slot to Any type.

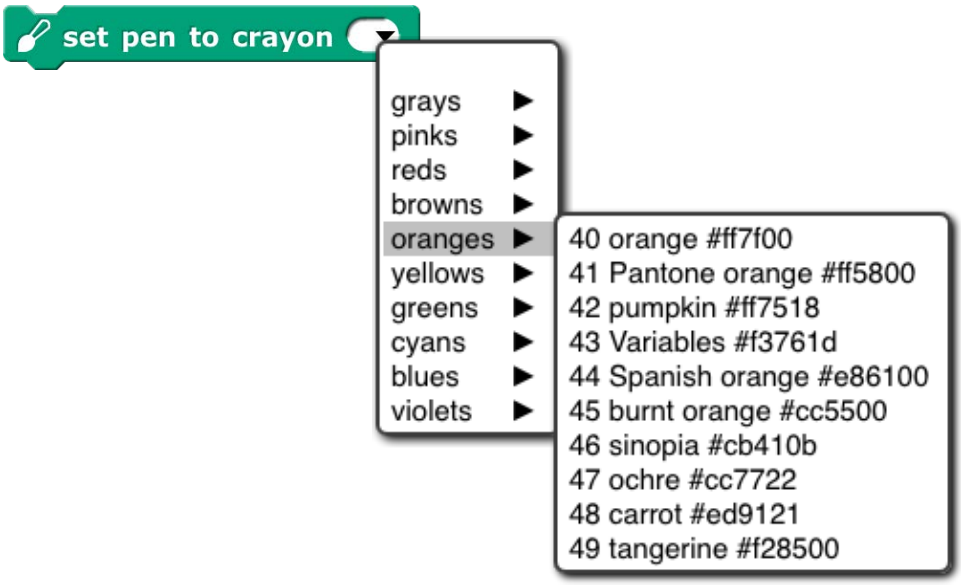
The crayon library has these blocks:



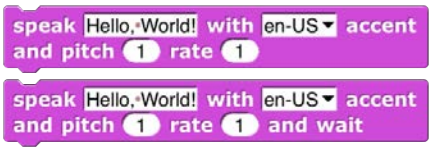
This library is an alternative to the **set pen hue** block with what is hoped to be a more interesting set of colors. For one thing, the rainbow (hue) colors are not equally spaced in color space; half the hue spectrum is some flavor of green, while brown isn't a rainbow color at all, nor are the shades of gray. The crayon library provides the equivalent of a box of 100 crayons. They are divided into color groups, so the menu in the **set pen to crayon** input slot has submenus, as shown below. The colors are chosen so that starting from crayon 0, **change crayon by 10** rotates through an interesting, basic set of ten colors:

Using **change crayon by 5** instead gives ten more colors, for a total of 20:

The **set pen to crayon** and **change crayon by** blocks are the only ones meant for users of the library; the others are helper functions used within the library itself. (Why didn't we use the colors of the 100-crayon Crayola™ box? A few reasons, one of which is that some Crayola colors aren't representable on RGB screens. Some year when you have nothing else to do, look up "color space" on Wikipedia. Also "crayon." Oh, it's deliberate that **change crayon by 5** doesn't include white, since that's the usual stage background color. White is crayon 14.) Note that crayon 43 is "Variables"; all the block colors are included. This library is now incorporated in the color library described earlier.



The speech synthesis library has these blocks:

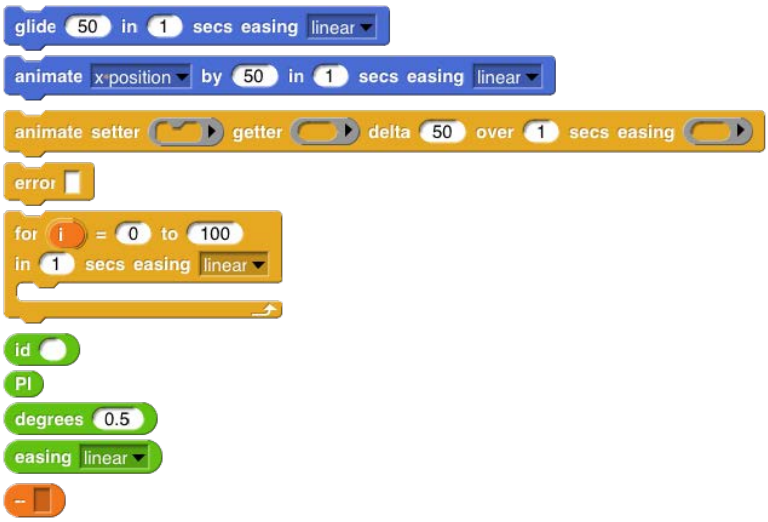


This library interfaces with a capability in up-to-date browsers, so it might not work for you. It works best if the accent matches the text!

The pixels library has one block:

Costumes are first class data in Snap!. Most of the processing of costume data is done by primitive blocks in the Looks category. (See page 63.) This library provides **snap**, which takes a picture using your computer's camera and reports it as a costume.

The animation library has these blocks:



Despite the name, this isn't only about graphics; you can animate the values of a variable, or anything else that's expressed numerically.

The central idea of this library is an *easing function*, a reporter whose domain and range are real numbers between 0 and 1 inclusive. The function represents what fraction of the "distance" (in quotes because it might be any numeric value, such as temperature in a simulation of weather) from here to there should be covered in what fraction of the time. A linear easing function means steady progression. A quadratic easing function means starting slowly and accelerating. (Note that, since it's a requirement that $f(0)=0$ and $f(1)=1$, there is only one linear easing function, $f(x)=x$, and similarly for other categories.) The 

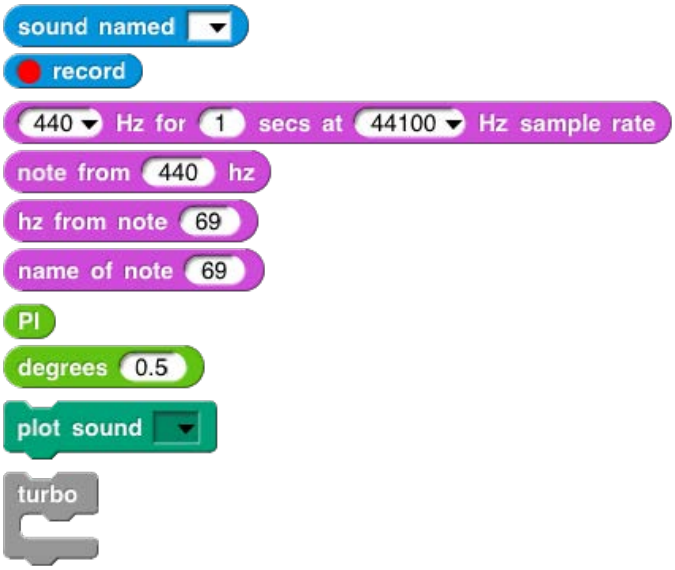
block reports some of the common easing functions.

The two Motion blocks in this library animate a sprite. ***Glide*** always animates the sprite's motion. ***Animate***'s first pulldown menu input allows you to animate horizontal or vertical motion, but will also animate the sprite's direction or size. The ***animate*** block in Control lets you animate any numeric quantity with any easing function. The getter and setter inputs are best explained by example:

is equivalent to

The other eight blocks in the library are helpers for these four.

The sound manipulation library includes these blocks:

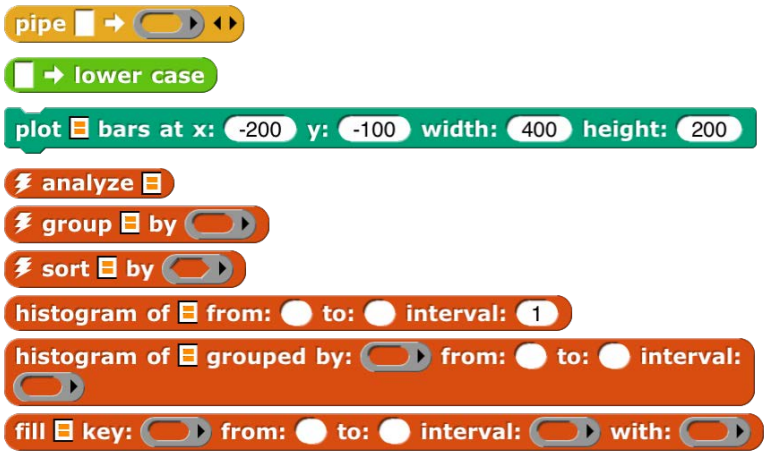


This library takes a sound, one that you record or one from our collection of sounds, and manipulates it by systematically changing the intensity of the samples in the sound and by changing the sampling rate at which the sound is reproduced. Many of the blocks are helpers for the ***plot sound*** block, used to plot the waveform of a sound. The ***play sound*** (primitive) block plays a sound. ***__ Hz for*** reports a sine wave as a list of samples.

The bigger data library includes these *experimental* blocks:

They are the same as the corresponding blocks in the list utilities library, but using a partial compilation technique to allow some use cases to run very fast, even on extremely large lists. Only primitive functions can be used in the function input slots. Save your project before experimenting with these. Blocks starting with the lightning bolt symbol in other libraries are also partially compiled for speed.

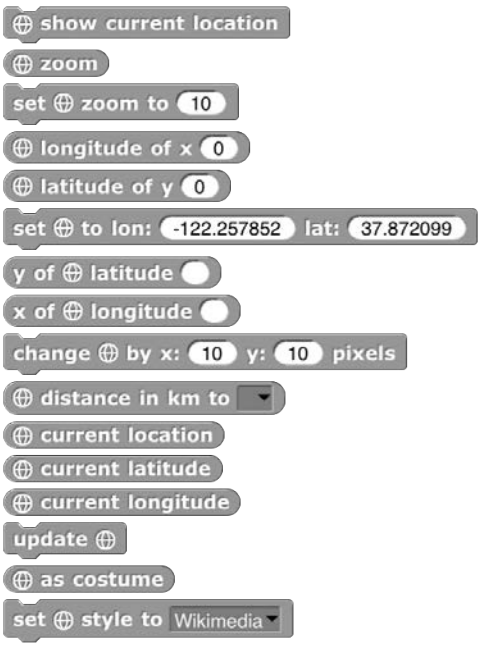
The frequency distribution analysis library has these blocks:



This is a collection of tools for analyzing large data sets and plotting histograms of how often some value is found in some column of the table holding the data.

For more information go here:
http://tinyurl.com/jens-data

The world map library has these blocks:



Using any of the command blocks puts a map on the screen, in a layer in front of the stage's background but behind the pen trails layer (which is in turn behind all the sprites). The first block asks your browser for your current physical location, for which you may be asked to give permission. The next two blocks get and set the map's zoom amount; the default zoom of 10 fits from San Francisco not quite down to Palo Alto on the screen. A zoom of 1 fits almost the entire world. A zoom of 3 fits the United States; a zoom of 5 fits Germany. The zoom can be changed in half steps, i.e., 5.5 is different from 5, but 5.25 isn't.

The next five blocks concern the conversion between stage coordinates (pixels) and Earth coordinates (latitude and longitude). The **change by x: y:** block shifts the map relative to the stage. The **distance to** block measures the map distance (in meters) between two sprites. The three reporters with **current** in their names find *your* actual location, again supposing that geolocation is enabled on your device. **Update** redraws the map; **as costume** reports the visible section of the map as a costume. **Set style** allows things like satellite pictures.

The variables library has these blocks:

These blocks allow a program to perform the same operation as the Make a variable button, making global, sprite local, or script variables, but allowing the program to compute the variable name(s). It can also set and find the values of these variables, show and hide their stage watchers, delete them, and find out if they already exist.

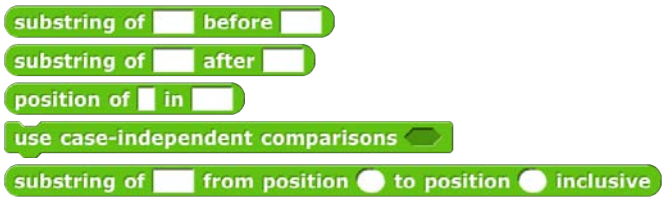
The JSON library includes these blocks:

Listify takes some text in JSON format (see page 42) and converts it to a structured list. ***Value at key*** looks up a key-value pair in a (listified) JSON dictionary.

The parallelization library contains these blocks:

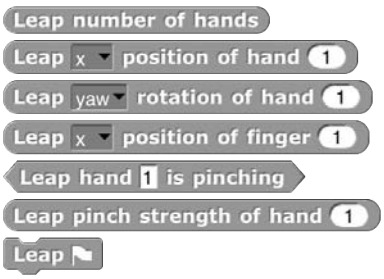
The two ***do in parallel*** blocks take any number of scripts as inputs. Those scripts will be run in parallel, like ordinary independent scripts in the scripting area. The ***and wait*** version waits until all of those scripts have finished before continuing the script below the block.

The string processing library provides these blocks:



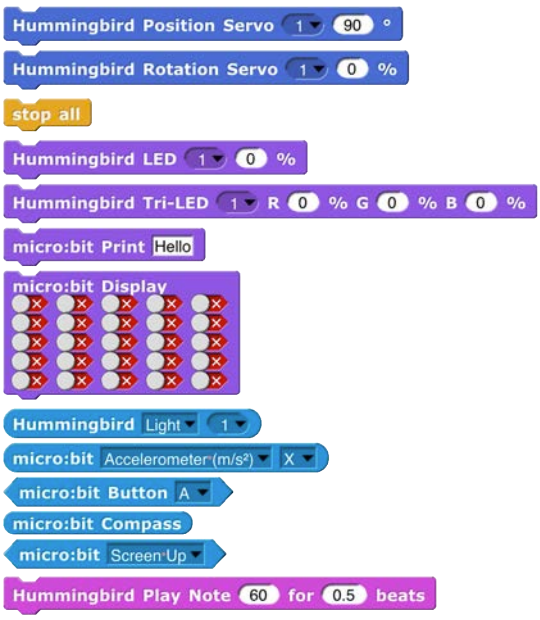
All of these could be written in Snap! itself, but these are implemented using the corresponding JavaScript library functions directly, so they run fast. They can be used, for example, in scraping data from a web site. The command ***use case-independent comparisons*** applies only to this library.

The LEAP Motion library has these blocks:



The LEAP Motion controller is a piece of hardware that senses the position of the user's hands, modeling the angles of each knuckle. It's one of many external devices that Snap! can support. Generally, to use a device with Snap!, you must download and install a separate controller program. That's true for the LEAP too, but that controller is installed as part of setting up the device, so all that's needed is the block library to go with it.

The Hummingbird library provides these blocks:



The blocks are intended for use with the Hummingbird robotics kit and for the micro:bit controller.

Back to the file menu: The **Costumes…** option opens a browser into the costume library:

You can import a single costume by clicking it and then clicking the Import button. Alternatively, you can import more than one costume by double-clicking each one, and then clicking Cancel when done. Notice that some costumes are tagged with "svg" in this picture; those are vector-format costumes that are not (yet, coming soon) editable within Snap!.

If you have the stage selected in the sprite corral, rather than a sprite, the **Costumes…** option changes to a **Backgrounds…** option, with different choices in the browser:

The costume and background libraries include both bitmap (go jagged if enlarged) and vector (enlarge smoothly) images. Thanks to Scratch 2.0/3.0 for most of these images! Some older browsers refuse to import a vector image, but instead convert it to bitmap.

The **Sounds…** option opens the third kind of media browser:

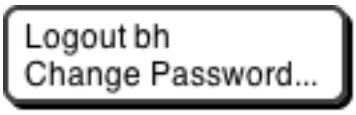
The Play buttons can be used to preview the sounds.

The Cloud Menu

The cloud icon  shows a menu of options relating to your Snap! cloud account. If you are not logged in, you see the outline icon  and get this menu:

Choose **Login…** if you have a Snap! account and remember your password. Choose **Signup…** if you don't have an account. Choose **Reset Password…** if you've forgotten your password or just want to change it. You will then get an email, at the address you gave when you created your account, with a new temporary password. Use that password to log in, then you can choose your own password, as shown below. Choose **Resend Verification Email…** if you have just created a Snap! account but can't find the email we sent you with the link to verify that it's really your email. (If you still can't find it, check your spam folder.)

If you are already logged in, you'll see the solid icon  and get this menu:



Logout is obvious, but has the additional benefit of showing you who's logged in. **Change password…** will ask for your old password (the temporary one if you're resetting your password) and the new password you want, entered twice because it doesn't echo.

The Settings Menu

The settings icon shows a menu of Snap! options, either for the current project or for you permanently, depending on the option:

The **Language…** option lets you see the Snap! user interface (blocks and messages) in a language other than English. (Note: Translations have been provided by Snap! users. If your native language is missing, send us an email!)

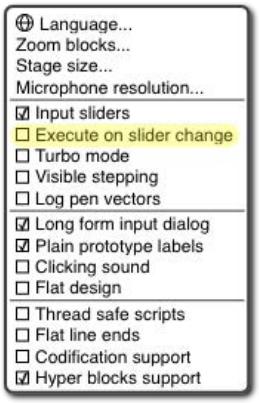
The **Zoom blocks...** option lets you change the size of blocks, both in the palettes and in scripts. The standard size is 1.0 units. The main purpose of this option is to let you take very high-resolution pictures of scripts for use on posters. It can also be used to improve readability when projecting onto a screen while lecturing, but bear in mind that it doesn't make the palette or script areas any wider, so your computer's command-option-+ feature may be more practical. Note that a zoom of 2 is gigantic! Don't even try 10.

The **Stage size…** option lets you set the size of the *full-size* stage in pixels. If the stage is in half-size or double-size (presentation mode), the stage size values don't change; they always reflect the full-size stage.

The **Microphone resolution…** option sets the buffer size used by the **microphone** block in Settings. "Resolution" is an accurate name if you are getting frequency domain samples; the more samples, the narrower the range of frequencies in each sample. In the time domain, the buffer size determines the length of time over which samples are collected.

The remaining options let you turn various features on and off. There are three groups of checkboxes. The first is for temporary settings not saved in your project nor in your user preferences.

Input sliders provides an alternate way to put values in numeric input slots; if you click in such a slot, a slider appears that you can control with the mouse:



The range of the slider will be from 25 less than the input's current value to 25 more than the current value. If you want to make a bigger change than that, you can slide the slider all the way to either end, then click on the input slot again, getting a new slider with a new center point. But you won't want to use this technique to change the input value from 10 to 1000, and it doesn't work at all for non-integer input ranges. This feature was implemented because software keyboard input on phones and tablets didn't work at all in the beginning, and still doesn't on Android devices, so sliders provide a workaround. It

has since found another use in providing "lively" response to input changes; if **Input sliders** is checked, reopening the settings menu will show an additional option called **Execute on slider change**. If this option is also checked, then changing a slider in the scripting area automatically runs the script in which that input appears. The project **live-tree** in the **Examples** collection shows how this can be used; it features a fractal tree custom block with several inputs, and you can see how each input affects the picture by moving a slider.

Turbo mode makes many projects run much faster, at the cost of not keeping the stage display up to date. (Snap! ordinarily spends most of its time drawing sprites and updating variable watchers, rather than actually carrying out the instructions in your scripts.) So turbo mode isn't a good idea for a project with **glide** blocks or one in which the user interacts with animated characters, but it's great for drawing a complicated fractal, or computing the first million digits of π, so that you don't need to see anything until the final result. While in turbo mode, the button that normally shows a green flag instead shows a green lightning bolt. (But **when**  **clicked** hat blocks still activate when the button is clicked.)

Visible stepping enables the slowed-down script evaluation described in Chapter I. Checking this option is equivalent to clicking the footprint button above the scripting area. You don't want this on except when you're actively debugging, because even the fastest setting of the slider is still slowed a lot.

Log pen vectors tells Snap! to remember lines drawn by sprites as exact vectors, rather than remember only the pixels that the drawing leaves on the stage. This remembered vector picture can be used in two ways: First, right-clicking on a **pen trails** block gives an option to relabel it into a **pen vectors** block which, when run, reports the logged lines as a vector (svg) costume. Second, right-clicking on the stage when there are logged vectors shows an extra option, **svg…**, that exports a picture of the stage in vector format. Only lines are logged, not color regions made with the **fill** block.

The next group of four are user preference options, preserved when you load a new project. **Long form input dialog**, if checked, means that whenever a custom block input name is created or edited, you immediately see the version of the input name dialog that includes the type options, default value setting, etc., instead of the short form with just the name and the choice between input name and title text. The default (unchecked) setting is definitely best for beginners, but more experienced Snap! programmers may find it more convenient always to see the long form.

Plain prototype labels eliminates the plus signs between words in the Block Editor prototype block. This makes it harder to add an input to a custom block; you have to hover the mouse where the plus sign would have been, until a single plus sign appears temporarily for you to click on. It's intended for people making pictures of scripts in the block editor for use in documentation, such as this manual. You probably won't need it otherwise.

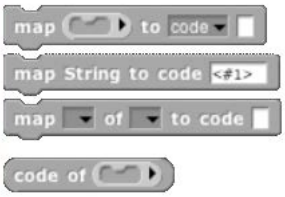
Clicking sound causes a really annoying sound effect whenever one block snaps next to another in a script. Certain very young children, and our colleague Dan Garcia, like this, but if you are such a child you should bear in mind that driving your parents or teachers crazy will result in you not being allowed to use Snap!. It might, however, be useful for visually impaired users.

Flat design changes the "skin" of the Snap! window to a really hideous design with white and pale-grey background, rectangular rather than rounded buttons, and monochrome blocks (rather than the shaded, somewhat 3D-looking normal blocks). The monochrome blocks are the reason for the "flat" in the name of this option. The only thing to be said for this option is that, because of the white background, it may blend in better with the rest of a web page when a Snap! project is run in a frame in a larger page.

The final group of settings change the way Snap! interprets your program; they are saved with the project, so anyone who runs your project will experience the same behavior. **Thread safe scripts** changes the way Snap! responds when an event (clicking the green flag, say) starts a script, and then, while the script is still running, the same event happens again. Ordinarily, the running process stops where it is, ignoring the remaining commands in the script, and the entire script starts again from the top. This behavior is inherited from Scratch, and some converted Scratch projects depend on it; that's why it's the default. It's also sometimes the right thing, especially in projects that play music in response to mouse clicks or keystrokes. If a note is still playing but you ask for another one, you want the new one to start right then, not later after the old process finishes. But if your script makes several changes to a database and is interrupted in the middle, the result may be that the database is inconsistent. When you select **Thread safe scripts**, the same event happening again in the middle of running a script is simply ignored. (This is arguably still not the right thing; the event should be remembered and the script run again as soon as it finishes. We'll probably get around to adding that choice eventually.) Keyboard events (when __ key pressed) are always thread-safe.

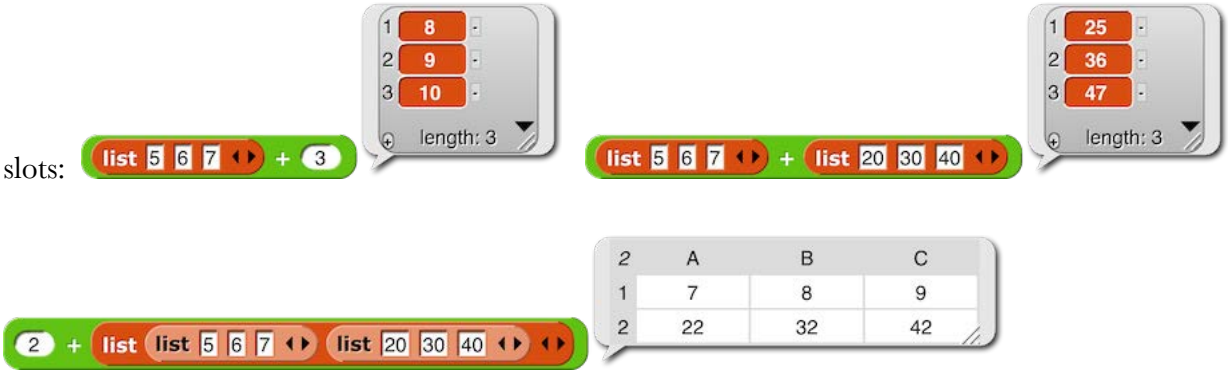
Flat line ends affects the drawing of thick lines (large pen width). Usually the ends are rounded, which looks best when turning corners. With this option selected, the ends are flat. It's useful for drawing a brick wall or filling a rectangle.

Codification support enables a feature that can translate a Snap! project to a text-based (rather than block-based) programming language. The feature doesn't know about any particular other language; instead, you can provide a translation for each primitive block using these special blocks:



Using these primitive blocks, you can build a block library to translate into any programming language. Watch for such libraries to be added to our library collection (or contribute one). To see some examples, open the project "**Codification**" in the **Examples** project list. Edit the blocks **map to Smalltalk**, **map to JavaScript**, etc., to see examples of how to provide translations for blocks.

Hyper blocks support enables the use of vector and matrix inputs in what are normally scalar (non-list) input slots:



For more information on this feature, see Section I.G, page 18.

Visible Stepping Controls

After the menu buttons you'll see the project name. After that comes the footprint  button used to turn on visible stepping and, when it's on, the slider to control the speed of stepping.

Stage Resizing Buttons

Still in the tool bar, but above the left edge of the stage, are two buttons that change the size of the stage. The first is the **shrink/grow** button. Normally it looks like this:  Clicking the button displays the stage at half-normal size horizontally and vertically (so it takes up ¼ of its usual area). When the stage is half size the button looks like this:  and clicking it returns the stage to normal size. The main reason you'd want a half size stage is during the development process, when you're assembling scripts with wide input expressions and the normal scripting area isn't wide enough to show the complete script. You'd typically then switch back to normal size to try out the project. The next **presentation mode** button normally looks like this:  Clicking the button makes the stage double size in both dimensions and eliminates most of the other user interface elements (the palette, the scripting area, the sprite corral, and most of the tool bar). When you open a shared project using a link someone has sent you, the project starts in presentation mode. While in presentation mode, the button looks like this:  Clicking it returns to normal (project development) mode.

Project Control Buttons

Above the right edge of the stage are three buttons that control the running of the project.

Technically, the **green flag**  is no more a project control than anything else that can trigger a hat block: typing on the keyboard or clicking on a sprite. But it's a convention that clicking the flag should start the action of the project from the beginning. It's only a convention; some projects have no flag-controlled scripts at all, but respond to keyboard controls instead. Clicking the green flag also deletes temporary clones. Shift-clicking the button enters Turbo mode, and the button then looks like a lightning bolt:  Shift-clicking again turns Turbo mode off.

Scripts can simulate clicking the green flag by **broadcast**ing the special message **__shout__go__** (two underscores in each of the three positions shown). If you forget where the underscores go, shift-click on the block's input slot to see a green flag in the menu.

The **pause** button  suspends running all scripts. If clicked while scripts are running, the button changes shape to become a **play** button:  Clicking it while in this form resumes the suspended scripts. There is also a **pause all** block in the Control palette that can be inserted in a script to suspend all scripts; this provides the essence of a breakpoint debugging capability. The use of the pause button is slightly different in visible stepping mode, described in Chapter I.

The **stop** button  stops all scripts, like the **stop all** block. It does *not* prevent a script from starting again in response to a click or keystroke; the user interface is always active. There is one exception: generic **when** blocks  will not fire until some non-generic event starts a script. The **stop** button also deletes all temporary clones.

B. The Palette Area

At the top of the palette area are the eight buttons that select which palette (which block category) is shown: Motion, Looks, Sound, Pen, Control, Sensing, Operators, and Variables (which also includes the List and Other blocks). There are no menus behind these buttons.

Buttons in the Palette

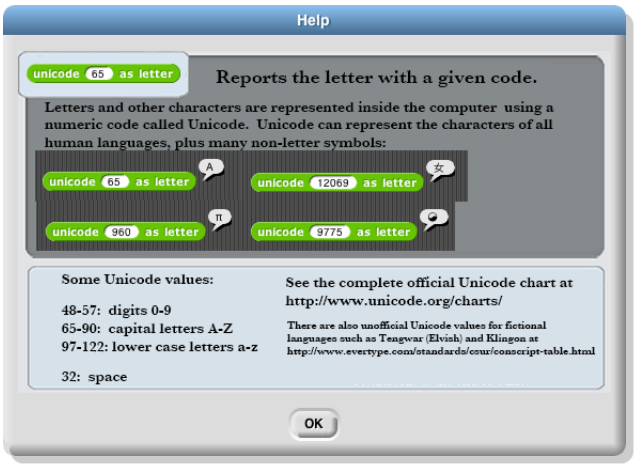
Under the eight palette selector buttons, at the top of the actual palette, are two semi-transparent buttons. The first is the *search*  button, which is equivalent to typing control-F: It replaces the palette with a search bar into which you can type part of the title text of the block you're trying to find. To leave this search mode, click one of the eight palette selectors, or type the Escape key.

The other button  is equivalent to the "Make a block" button, except that the dialog window that it opens has the current palette (color) preselected.

Context Menus for Palette Blocks

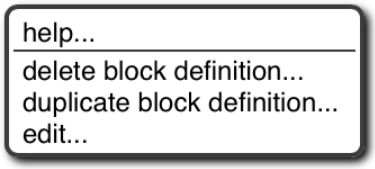
Most elements of the Snap! display can be control-clicked/right-clicked to show a *context menu*, with items relevant to that element. If you control-click/right-click a *primitive* block in the palette, you see this menu:

The **help…** option displays a box with documentation about the block. Here's an example:

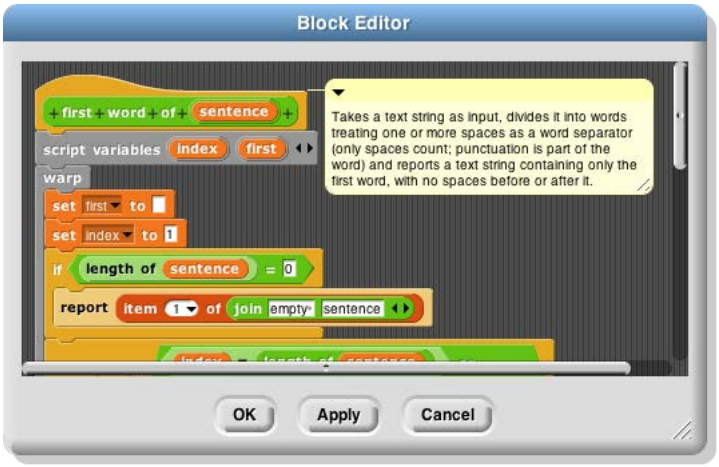


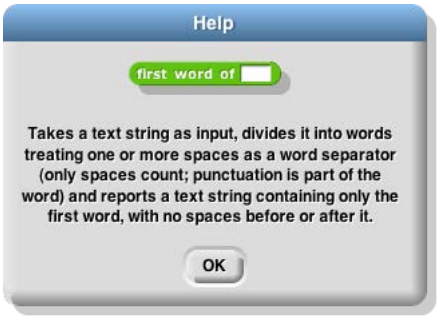
The **hide** option removes that block from the palette. (This option is available only when clicking the block in the palette itself, not in a script.) The purpose of the option is to allow teachers to present students with a simplified Snap! with some features effectively removed. The hiddenness of primitives is saved with each project, so students can load a shared project and see just the desired blocks.

If you control-click/right-click a *custom* (user-defined) block in the palette, you see this menu:



The **help…** option for a custom block displays the comment, if any, attached to the custom block's hat block in the Block Editor. Here is an example of a block with a comment and its help display:





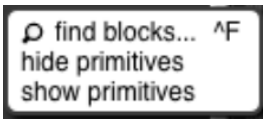
The **delete block definition…** option asks for confirmation, then deletes the custom block and removes it from any scripts in which it appears. (The result of this removal may not leave a sensible script; it's best to find and correct such scripts *before* deleting a block.) Note that there is no option to *hide* a custom block.

The **duplicate block definition…** option makes a *copy* of the block and opens that copy in the Block Editor. Since you can't have two custom blocks with the same title text and input types, the copy is created with "(2)" (or a higher number if necessary) at the end of the block prototype.

The **edit…** option opens a Block Editor with the definition of the custom block.

Context Menu for the Palette Background

Right-click/control-click on the grey *background* of the palette area shows this menu:



The **find blocks…** option does the same thing as the magnifying-glass button. The **hide primitives** option hides *all* of the primitives in that palette. The **show primitives** option, which is in the menu only if some primitives of this palette are hidden, unhides all of them.

Palette Resizing

At the right end of the palette area, just to the left of the scripting area, is a resizing handle that can be dragged rightward to increase the width of the palette area. This is useful if you write custom blocks with very long names. You can't reduce the width of the palette below its standard value.

C. The Scripting Area

The scripting area is the middle vertical region of the Snap! window, containing scripts and also some controls for the appearance and behavior of a sprite. There is always a *current sprite,* whose scripts are shown in the scripting area. A dark grey rounded rectangle in the sprite corral shows which sprite (or the stage) is current. Note that it's only the visible *display* of the scripting area that is "current" for a sprite; all scripts of all sprites may be running at the same time. Clicking on a sprite thumbnail in the sprite corral makes it current. The stage itself can be selected as current, in which case the appearance is different, with some primitives not shown.

Sprite Appearance and Behavior Controls

At the top of the scripting area are a picture of the sprite and some controls for it:

Note that the sprite picture reflects its rotation, if any. There are three things that can be controlled here:

1. The three circular buttons in a column at the left control the sprite's *rotation* behavior. Sprite costumes are designed to be right-side-up when the sprite is facing toward the right (direction = 90). If the topmost button is lit, the default as shown in the picture above, then the sprite's costume rotates as the sprite changes direction. If the middle button is selected, then the costume is reversed left-right when the sprite's direction is roughly leftward (direction between 180 and 359, or equivalently, between -180 and -1). If the bottom button is selected, the costume's orientation does not change regardless of the sprite's direction.

2. The sprite's *name* can be changed in the text box that, in this picture, says "Sprite."

3. Finally, if the ***draggable*** checkbox is checked, then the user can move the sprite on the stage by clicking and dragging it. The common use of this feature is in game projects, in which some sprites are meant to be under the player's control but others are not.

Scripting Area Tabs

Just below the sprite controls are three *tabs* that determine what is shown in the scripting area:

Scripts and Blocks Within Scripts

Most of what's described in this section also applies to blocks and scripts in a Block Editor.

Clicking on a script (which includes a single unattached block) runs it. If the script starts with a hat block, clicking on the script runs it even if the event in the hat block doesn't happen. (This is a useful debugging technique when you have a dozen sprites and they each have five scripts with green-flag hat blocks, and you want to know what a single one of those scripts does.) The script will have a green "halo" around it while it's running. If the script is shared with clones, then while it has the green halo it will also have a count of how many instances of the script are running. Clicking a script with such a halo *stops* the script. (If the script includes a ***warp*** block, which might be inside a custom block used in the script, then Snap! may not respond immediately to clicks.)

If a script is shown with a *red* halo, that means that an error was caught in that script, such as using a list where a number was needed, or vice versa. Clicking the script will turn off the halo.

If any blocks have been dragged into the scripting area, then in its top right corner you'll see an *undo*  and/or *redo*  button that can be used to undo or redo block and script drops. When you undo a drop into an input slot, whatever used to be in the slot is restored. The redo button appears once you've used undo.

The third button  starts keyboard editing mode (Section D, page 115).

Control-click/right-clicking a primitive block within a script shows a menu like this one:

The **help…** option shows the help screen for the block, just as in the palette. The other options appear only when a block is right-clicked/control-clicked in the scripting area.

Not every primitive block has a **relabel…** option. When present, it allows the block to be replaced by another, similar block, keeping the input expressions in place. For example, here's what happens when you choose **relabel…** for an arithmetic operator:

Note that the inputs to the existing **–** block are displayed in the menu of alternatives also. Click a block in the menu to choose it, or click outside the menu to keep the original block.

The **duplicate** option makes a copy of the *entire script* starting from the selected block. The copy is attached to the mouse, and you can drag it to another script (or even to another Block Editor window), even though you are no longer holding down the mouse button. Click the mouse to drop the script copy.

The block picture underneath the word **duplicate** is another duplication option, but it duplicates only the selected block, not everything under it in the script. Note that if the selected block is a C-shaped control block, the script inside its C-shaped slot is included. If the block is at the end of its script, this option does not appear.

The **delete** option deletes the selected block from the script.

The **script pic…** option opens a new browser tab containing a picture of the entire script, not just from the selected block to the end, or, in some browsers, saves the picture directly. In the first case, you can use the browser's Save feature to put the picture in a file. This is a super useful feature if you happen to be writing a Snap! manual! (If you have a Retina display, consider turning off Retina support before making script pictures; if not, they end up huge.)

If the script does *not* start with a hat block, or you clicked on a reporter, then there's one more option: **ringify** (and, if there is already a grey ring around the block or script, **unringify**). **Ringify** surrounds the block (reporter) or the entire script (command) with a grey ring, meaning that the block(s) inside the ring are themselves data, as an input to a higher order procedure, rather than something to be evaluated within the script. See Section VI, Procedures as Data.

Clicking a *custom* block in a script gives a similar but different menu:

The **relabel…** option for custom blocks shows a menu of other same-shape custom blocks with the same inputs. At present you can't relabel a custom block to a primitive block or vice versa. The two options at the bottom, for custom blocks only, are the same as in the palette.

If a reporter block is in the scripting area, possibly with inputs included, but not itself serving as input to another block, then the menu is a little different again:

What's new here is the **result pic…** option. It's like **script pic…** but it includes in the picture a speech balloon with the result of calling the block.

Scripting Area Background Context Menu

Control-click/shift-click on the grey striped background of the scripting area gives this menu:

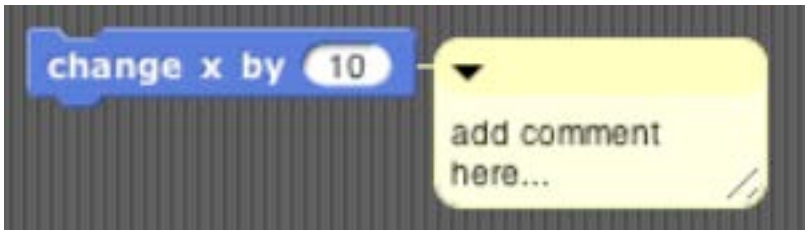
The **undrop** option is a sort of "undo" feature for the common case of dropping a block somewhere other than where you meant it to go. It remembers all the dragging and dropping you've done in this sprite's scripting area (that is, other sprites have their own separate drop memory), and undoes the most recent, returning the block to its former position, and restoring the previous value in the relevant input slot, if any. Once you've undropped something, the **redrop** option appears, and allows you to repeat the operation you just undid. These menu options are equivalent to the  and  buttons described earlier.

The **clean up** option rearranges the position of scripts so that they are in a single column, with the same left margin, and with uniform spacing between scripts. This is a good idea if you can't read your own project!

The **add comment** option puts a comment box, like the picture to the right, in the scripting area. It's attached to the mouse, as with duplicating scripts, so you position the mouse where you want the comment and click to release it. You can then edit the text in the comment as desired.

You can drag the bottom right corner of the comment box to resize it. Clicking the arrowhead at the top left changes the box to a single-line compact form, as below, so that you can have a number of collapsed comments in the scripting area and just expand one of them when you want to read it in full.

If you drag a comment over a block in a script, the comment will be attached to the block with a yellow line:



Comments have their own context menu, with obvious meanings:

Back to the options in the menu for the background of the scripting area:

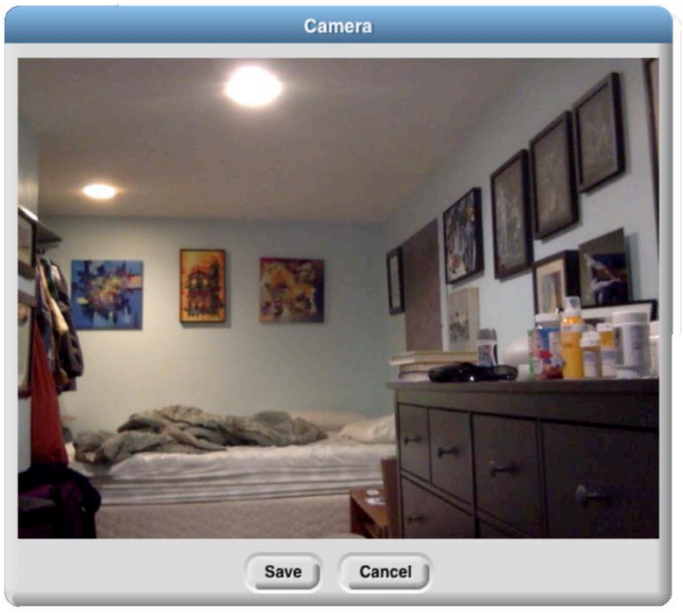
The **scripts pic…** option saves, or opens a new browser tab with, a picture of *all* scripts in the scripting area, just as they appear, but without the grey striped background. Note that "all scripts in the scripting area" means just the top-level scripts of the current sprite, not other sprites' scripts or custom block definitions.

Finally, the **make a block…** option does the same thing as the "Make a block" button in the palettes. It's a shortcut so that you don't have to keep scrolling down the palette if you make a lot of blocks.

Controls in the Costumes Tab

If you click on the word "Costumes" under the sprite controls, you'll see something like this:

The **Turtle** costume is always present in every sprite; it is costume number 0. Other costumes can be painted within **Snap!** or imported from files or other browser tabs if your browser supports that. Clicking on a costume selects it; that is, the sprite will look like the selected costume. Clicking on the paint brush icon  opens the *Paint Editor,* in which you can create a new costume. Clicking on the camera icon  opens a window in which you see what your computer's camera is seeing, and you can take a picture (which will be the full size of the stage unless you shrink it in the Paint Editor). This works only if you give **Snap!** permission to use the camera, and maybe only if you opened **Snap!** in secure (HTTPS) mode, and then only if your browser loves you.



Brian's bedroom when he's staying at Paul's house.

Control-clicking/right-clicking on the turtle picture gives this menu:

In this menu, you choose the turtle's *rotation point,* which is also the point from which the turtle draws lines. The two pictures below show what the stage looks like after drawing a square in each mode; **tip** (otherwise known as "Jens mode") is on the left in the pictures below, **middle** ("Brian mode") on the right:

As you see, "tip" means the front tip of the arrowhead; "middle" is not the middle of the shaded region, but actually the middle of the four vertices, the concave one. (If the shape were a simple isosceles triangle instead of a fancier arrowhead, it would mean the midpoint of the back edge.) The advantage of **tip** mode is that the sprite is less likely to obscure the drawing. The advantage of **middle** mode is that the rotation point of a sprite is rarely at a tip, and students are perhaps less likely to be confused about just what will happen if you ask the turtle to turn 90 degrees from the position shown. (It's also the traditional rotation point of the Logo turtle, which originated this style of drawing.)

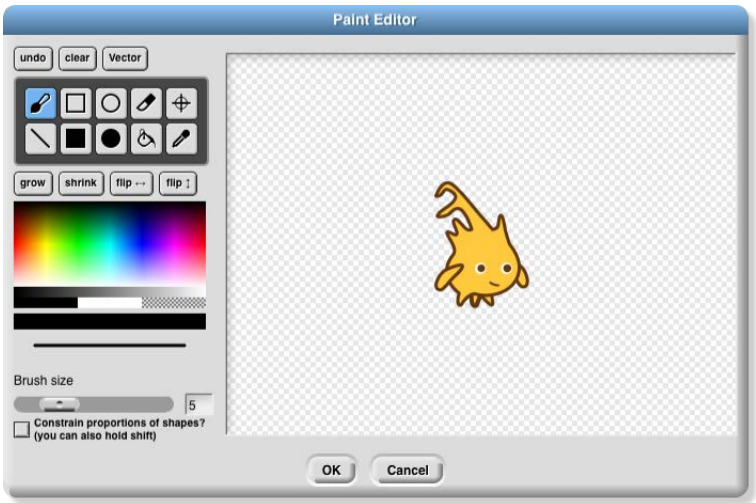
Costumes other than the turtle have a different context menu:

The **edit** option opens the Paint Editor on this costume. The **rename** option opens a dialog box in which you can rename the costume. (A costume's initial name comes from the file from which it was imported, if any, or is something like **costume5**.) **Duplicate** makes a copy of the costume, in the same sprite. (Presumably you'd do that because you intend to edit one of the copies.) **Delete** is obvious. The **export** option opens a new browser tab with a picture of the costume. You can then save it to a file, or select a different sprite in the Snap! tab, return to the picture tab, and drag the costume onto the Snap! tab to copy the costume to another sprite.

You can drag costumes up and down in the Costumes tab in order to renumber them, so that **next costume** will behave as you prefer.

The Paint Editor

Here is a picture of a Paint Editor window:



If you've used any painting program, most of this will be familiar to you. Currently, costumes you import can be edited only if they are in a bitmap format (png, jpeg, gif, etc.). There is a vector editor, but it works only for creating a costume, not editing an imported vector (svg) picture. Unlike the case of the Block Editor, only one Paint Editor window can be open at a time.

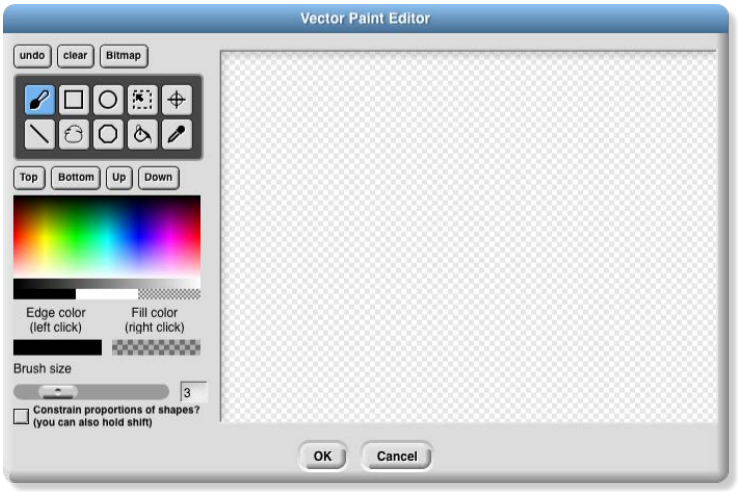
The ten square buttons in two rows of five near the top left of the window are the *tools*. The top row, from left to right, are the paintbrush tool, the outlined rectangle tool, the outlined ellipse tool, the eraser tool, and the rotation point tool. The bottom row tools are the line drawing tool, the solid rectangle tool, the solid ellipse tool, the floodfill tool, and the eyedropper tool. Below the tools is a row of four buttons that immediately change the picture. The first two change its overall size; the next two flip the picture around horizontally or vertically. Below these are a color palette, a greyscale tape, and larger buttons for black, white, and transparent paint. Below these is a solid bar displaying the currently selected color. Below that is a picture of a line showing the brush width for painting and drawing, and below that, you can set the width either with a slider or by typing a number (in pixels) into the text box. Finally, the checkbox constrains the line tool to draw horizontally or vertically, the rectangle tools to draw squares, and the ellipse tools to draw circles. You can get the same effect temporarily by holding down the shift key, which makes a check appear in the box as long as you hold it down. (But the Caps Lock key doesn't affect it.)

You can correct errors with the **undo** button, which removes the last thing you drew, or the **clear** button, which erases the entire picture. (Note, it does *not* revert to what the costume looked like before you started editing it! If that's what you want, click the **Cancel** button at the bottom of the editor.) When you're finished editing, to keep your changes, click **OK**.

Note that the ellipse tools work more intuitively than ones in other software you may have used. Instead of dragging between opposite corners of the rectangle circumscribing the ellipse you want, so that the endpoints of your dragging have no obvious connection to the actual shape, in Snap! you start at the center of the ellipse you want and drag out to the edge. When you let go of the button, the mouse cursor will be on the curve. If you drag out from the center at 45 degrees to the axes, the resulting curve will be a circle; if you drag more horizontally or vertically, the ellipse will be more eccentric. (Of course if you want an exact circle you can hold down the shift key or check the checkbox.) The rectangle tools, though, work the way you expect: You start at one corner of the desired rectangle and drag to the opposite corner.

Using the eyedropper tool, you can click anywhere in the Snap! window, even outside the Paint Editor, and the tool will select the color at the mouse cursor for use in the Paint Editor. You can only do this once, because the Paint Editor automatically selects the paintbrush when you choose a color. (Of course you can click on the eyedropper tool button again.)

The only other non-obvious tool is the rotation point tool. It shows in the Paint Editor where the sprite's current rotation center is (the point around which it turns when you use a **turn** block); if you click or drag in the picture, the rotation point will move where you click. (You'd want to do this, for example, if you want a character to be able to wave its arm, so you use two sprites connected together. You want the rotation point of the arm sprite to be at the end where it joins the body, so it remains attached to the shoulder while waving.)



The vector editor's controls are much like those in the bitmap editor. One point of difference is that the bitmap editor has two buttons for solid ■ and outline □ rectangles, and similarly for ellipses, but in the vector editor there is always an edge color and a fill color, even if the latter is "transparent paint," and so only one button per shape is needed. Since each shape that you draw is a separate layer (like sprites on the stage), there are controls to move the selected shape up (frontward) or down (rearward) relative to other shapes. There is a selection tool ⬚ to drag out a rectangular area and select all the shapes within that area.

Controls in the Sounds Tab

There is no Sound Editor in Snap!, and also no current sound the way there's a current costume for each sprite. (The sprite always has an appearance unless hidden, but it doesn't sing unless explicitly asked.) So the context menu for sounds has only ***rename, delete***, and ***export*** options, and it has a clickable button labeled **Play** or ***Stop*** as appropriate. There is a sound *recorder*, which appears if you click the red record button ( ):

The first, round button starts recording. The second, square button stops recording. The third, triangular button plays back a recorded sound. If you don't like the result, click the round button again to re-record. When you're satisfied, push the ***Save*** button. If you need a sound editor, consider the free (both senses) `http://audacity.sourceforge.net`.

D. Keyboard Editing

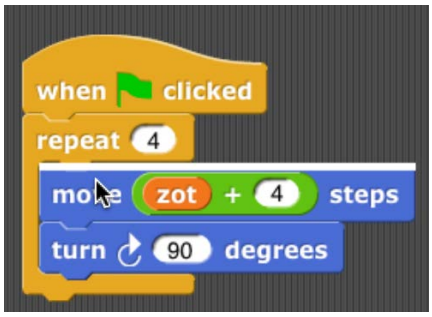
An ongoing area of research is how to make visual programming languages usable by people with visual or motoric disabilities. As a first step in this direction, we provide a keyboard editor, so that you can create and edit scripts without tracking the mouse. So far, not every user interface element is controllable by keyboard, and we haven't even begun providing *output* support, such as interfacing with a speech synthesizer. This is an area in which we know we have a long way to go! But it's a start. The keyboard editor may also be useful to anyone who can type faster than they can drag blocks.

Starting and stopping the keyboard editor

There are three ways to start the keyboard editor. **Shift-clicking** anywhere in the scripting area will start the editor at that point: either editing an existing script or, if you shift-click on the background of the scripting area,

editing a new script at the mouse position. Alternatively, typing **shift-enter** will start the editor on an existing script, and you can use the tab key to switch to another script. Or you can click the keyboard button at the top of the scripting area.

When the script editor is running, its position is represented by a blinking white bar:

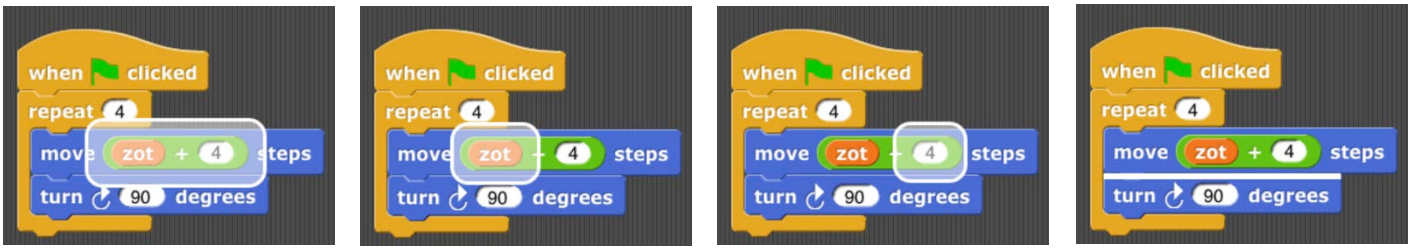


To leave the keyboard editor, type the **escape** key, or just click on the background of the scripting area.

Navigating in the keyboard editor

To move to a different script, type the **tab** key. **Shift-tab** to move through the scripts in reverse order.

A script is a vertical stack of command blocks. A command block may have input slots, and each input slot may have a reporter block in it; the reporter may itself have input slots that may have other reporters. You can navigate through a script quickly by using the **up arrow** and **down arrow** keys to move between command blocks. Once you find the command block that you want to edit, the **left and right arrow** keys move between editable items within that command. (Left and right arrow when there are no more editable items within the current command block will move up or down to another command block, respectively.) Here is a sequence of pictures showing the results of repeated right arrow keys starting from the position shown above:



You can rearrange scripts within the scripting area from the keyboard. Typing **shift-arrow** keys (left, right, up, or down) will move the current script. If you move it onto another script, the two won't snap together; the one you're moving will overlap the one already there. This means that you can move across another script to get to a free space.

Editing a script

Note that the keyboard editor *focus,* the point shown as a white bar or halo, is either *between* two command blocks or *on* an input slot. The editing keys do somewhat different things in each of those two cases.

The **backspace** key deletes a block. If the focus is between two commands, the one *before* (above) the blinking bar is deleted. If the focus is on an input slot, the reporter in that slot is deleted. (If that input slot has a default value, it will appear in the slot.) If the focus is on a *variadic* input (one that can change the number of inputs by clicking on arrowheads), then *one* input slot is deleted. (When you right-arrow into a variadic input, the focus first covers the entire thing, including the arrowheads; another right-arrow focuses on the first slot within that input group. The focus is "on the variadic input" when it covers the entire thing.)

The **enter** key does nothing if the focus is between commands, or on a reporter. If the focus is on a variadic input, the enter key adds one more input slot. If the focus is on a white input slot (one that doesn't have a reporter in it), then the enter key selects that input slot for *editing;* that is, you can type into it, just as if you'd clicked on the input slot. (Of course, if the focus is on an input slot containing a reporter, you can use the backspace key to delete that reporter, and then use the enter key to type a value into it.) When you finish typing the value, type the enter key again to accept it and return to navigation, or the escape key if you decide not to change the value already in the slot.

The **space** key is used to see a menu of possibilities for the input slot in focus. It does nothing unless the focus is on a single input slot. If the focus is on a slot with a pulldown menu of options, then the space key shows that menu. (If it's a block-colored slot, meaning that only the choices in the menu can be used, the enter key will do the same thing. But if it's a white slot with a menu, such as in the **turn** blocks, then enter lets you type a value, while space shows the menu.) Otherwise, the space key shows a menu of variables available at this point in the script. In either case, use the up and down arrow keys to navigate the menu, use the enter key to accept the highlighted entry, or use the escape key to leave the menu without choosing an option.

Typing **any other character** key (not special keys on fancy keyboards that do something other than generating a character) activates the *block search palette.* This palette, which is also accessible by typing control-F or command-F outside the keyboard editor, or by clicking the search button floating at the top of the palette, has a text entry field at the top, followed by blocks whose title text includes what you type. The character key you typed to start the block search palette is entered into the text field, so you start with a palette of blocks containing that character. Within the palette, blocks whose titles *start* with the text you type come first, then blocks in which *a word* of the title starts with the text you type, and finally blocks in which the text appears inside a word of the title. Once you have typed enough text to see the block you want, use the arrow keys to navigate to that block in the palette, then enter to insert that block, or escape to leave the block search palette without inserting the block. (When not in the keyboard editor, instead of navigating with the arrow keys, you drag the block you want into the script, as you would from any other palette.)

If you type an **arithmetic operator** (**+-*/**) or **comparison operator** (**<=>**) into the block search text box, you can type an arbitrarily complicated expression, and a collection of arithmetic operator blocks will be constructed to match:

As the example shows, you can also use **parentheses** for grouping, and non-numeric operands are treated as variables or primitive functions. (A variable name entered in this way may or may not already exist in the script. Only **round** and the ones in the pulldown menu of the **sqrt** block can be used as function names.)

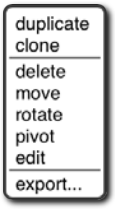
Running the selected script

Type **control-shift-enter** to run the script with the editor focus, like clicking the script.

E. Controls on the Stage

The stage is the area in the top right of the Snap! window in which sprites move.

Most sprites can be moved by clicking and dragging them. (If you have unchecked the ***draggable*** checkbox for a sprite, then dragging it has no effect.) Control-clicking/right-clicking a sprite shows this context menu:



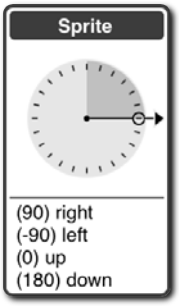
The ***duplicate*** option makes another sprite with the same scripts, same costumes, etc., as this sprite. The new sprite starts at a randomly chosen position different from the original, so you can see quickly which is which. The new sprite is *selected:* It becomes the current sprite, the one shown in the scripting area. The ***clone*** option makes a permanent clone of this sprite and selects it.

The ***delete*** option deletes the sprite. It's not just hidden; it's gone for good. The ***edit*** option selects the sprite. It doesn't actually change anything about the sprite, despite the name; it's just that making changes in the scripting area will change this sprite.

The ***move*** option shows a "move handle" inside the sprite (the diagonal striped square in the middle):

You can ordinarily just grab and move the sprite without this option, but there are two reasons you might need it: First, it works even if the "draggable" checkbox above the scripting area is unchecked. Second, it works for part sprites relative to their anchor; ordinarily, dragging a part moves the entire nested sprite.

The ***rotate*** option displays a rotation menu:



You can choose one of the four compass directions in the lower part (the same as in the ***point in direction*** block) or use the mouse to rotate the handle on the dial in 15° increments.

The ***pivot*** option shows a crosshair inside the sprite:

You can click and drag the crosshair anywhere onstage to set the costume's pivot point. (If you move it outside the sprite, then turning the sprite will revolve as well as rotate it around the pivot.) When done, click on the stage not on the crosshair. Note that, unlike moving the pivot point in the Paint Editor, this technique does not visibly move the sprite on the stage. Instead, the values of **x position** and **y position** will change.

The **edit** option makes this the selected sprite, highlighting it in the sprite corral and showing its scripting area. If the sprite was a temporary clone, it becomes permanent.

The **export…** option saves, or opens a new browser tab containing, the XML text representation of the sprite. (Not just its costume, but all of its costumes, scripts, local variables and blocks, and other properties.) You can save this tab into a file on your computer, and later import the sprite into another project. (In some browsers, the sprite is directly saved into a file.)

Control-clicking/right-clicking on the stage background (that is, anywhere on the stage except on a sprite) shows the stage's own context menu:

The stage's **edit** option selects the stage, so the stage's scripts and backgrounds are seen in the scripting area. Note that when the stage is selected, some blocks, especially the Motion ones, are not in the palette area because the stage can't move.

The **show all** option makes all sprites visible, both in the sense of the **show** block and by bringing the sprite onstage if it has moved past the edge of the stage.

The **pic…** option saves, or opens a browser tab with, a picture of everything on the stage: its background, lines drawn with the pen, and any visible sprites. What you see is what you get. (If you want a picture of just the background, select the stage, open its **costumes** tab, control-click/right-click on a background, and export it.)

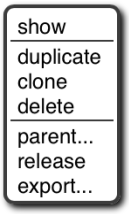
The **pen trails** option creates a new costume for the currently selected sprite consisting of all lines drawn on the stage by the pen of any sprite. The costume's rotation center will be the current position of the sprite.

F. The Sprite Corral and Sprite Creation Buttons

Between the stage and the sprite corral at the bottom right of the Snap! window is a dark grey bar containing three buttons. These are used to create a new sprite. The first button makes a sprite with just the turtle costume, with a randomly chosen position and pen color. (If you hold down the Shift key while clicking, the new sprite's direction will also be random.) The second button makes a sprite and opens the Paint Editor so that you can make your own costume for it. (Of course you could click the first button and then click the paint button in its **costumes** tab; this paint button is a shortcut for all that.) Similarly, the third button  uses your camera, if possible, to make a costume for the new sprite.

In the sprite corral, you click on a sprite's "thumbnail" picture to select that sprite (to make it the one whose scripts, costumes, etc. are shown in the scripting area). You can drag sprite thumbnails (but not the stage one) to reorder them; this has no special effect on your project, but lets you put related ones next to each other, for example. Double-clicking a thumbnail flashes a halo around the actual sprite on the stage.

You can right-click/control-click a sprite's thumbnail to get this context menu:



The **show** option makes the sprite visible, if it was hidden, and also brings it onto the stage, if it had moved past the stage boundary. The next three options are the same as in the context menu of the actual sprite on the stage, discussed above.

The **parent…** option displays a menu of all other sprites, showing which if any is this sprite's parent, and allowing you to choose another sprite (replacing any existing parent). The **release** option is shown only if this sprite is a (permanent, or it wouldn't be in the sprite corral) clone; it changes the sprite to a temporary clone. (The name is supposed to mean that the sprite is released from the corral.) The **export…** option exports the sprite, like the same option on the stage.

The context menu for the stage thumbnail has only one option, **pic…**, which takes a picture of everything on the stage, just like the same option in the context menu of the stage background.

G. Preloading a Project when Starting Snap!

There are several ways to include a pointer to a project in the URL when starting Snap! in order to load a project automatically. You can think of such a URL as just running the project rather than as running Snap!, especially if the URL says to start in presentation mode and click the green flag. The general form is

`http://snap.berkeley.edu/run#`***verb*:*project*&*flag*&*flag*…**

The "verb" above can be any of `open`, `run`, `cloud`, `present`, or `dl`. The last three are for shared projects in the Snap! cloud; the first two are for projects that have been exported and made available anywhere on the Internet.

Here's an example that loads a project stored at the Snap! web site (not the Snap! cloud!):

```
http://snap.berkeley.edu/run#open:http://snap.berkeley.edu/snapsource/Examples/vee.xml
```

The project file will be opened, and Snap! will start in edit mode (with the program visible). Using `#run:` instead of `#open:` will start in presentation mode (with only the stage visible) and will "start" the project by clicking the green flag. ("Start" is in quotation marks because there is no guarantee that the project includes any scripts triggered by the green flag. Some projects are started by typing on the keyboard or by clicking a sprite.)

If the verb is `run`, then you can also use any subset of the following flags:

`&editMode`	Start in edit mode, not presentation mode.
`&noRun`	Don't click the green flag.
`&hideControls`	Don't show the row of buttons above the stage (edit mode, green flag, pause, stop).
`&lang=fr`	Set language to (in this example) French.
`&noExitWarning`	When closing the window or loading a different URL, don't show the browser "are you sure you want to leave this page" message.

The last of these flags is intended for use on a web page in which a Snap! window is embedded.

Here's an example that loads a shared (public) project from the Snap! cloud:

```
http://snap.berkeley.edu/run#present:Username=jens&ProjectName=tree%20animation
```

(Note that "Username" and "ProjectName" are TitleCased, even though the flags such as "noRun" are camelCased. Note also that a space in the project name must be represented in Unicode as %20.) The verb present behaves like run: it ordinarily starts the project in presentation mode, but its behavior can be modified with the same four flags as for run. The verb cloud (yes, we know it's not a verb in its ordinary use) behaves like open except that it loads from the Snap! cloud rather than from the Internet in general. The verb dl (short for "download") does not start Snap! but just downloads a cloud-saved project to your computer as an .xml file. This is useful for debugging; sometimes a defective project that Snap! won't run can be downloaded, edited, and then re-saved to the cloud.

H. Mirror Sites

If the site snap.berkeley.edu is ever unavailable, you can load Snap! at any of the following mirror sites:

- http://bjc.edc.org/snapsource/snap.html
- http://media.mit.edu/~harveyb/snap
- http://cs10.org/snap

Appendix A. Snap! color library

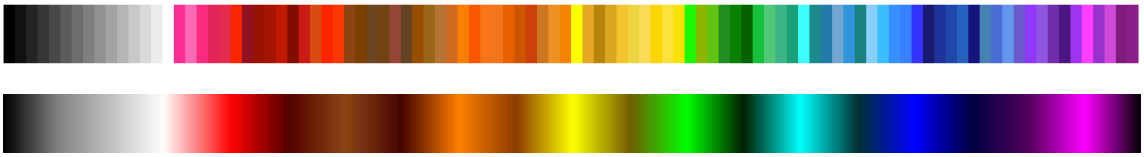
Your computer monitor can display millions of colors, but you probably can't distinguish that many. For example, here's red 57, green 180, blue 200: And here's red 57, green 182, blue 200: You might be able to tell them apart if you see them side by side: … but maybe not even then.

Color space—the collection of all possible colors—is three-dimensional, but there are many ways to choose the dimensions. RGB (red-green-blue), the one most commonly used, matches the way TVs and displays produce color. Behind every dot on the screen are three tiny lights: a red one, a green one, and a blue one. But if you want to print colors on paper, your printer probably uses a different set of three colors: CMY (cyan-magenta-yellow). You may have seen the abbreviation CMYK, which represents the common technique of adding black ink to the collection. (Mixing cyan, magenta, and yellow in equal amounts is supposed to result in black ink, but typically it comes out a not-very-intense gray instead.) Other systems that try to mimic human perception are HSL (hue-saturation-lightness) and HSV (hue-saturation-value).

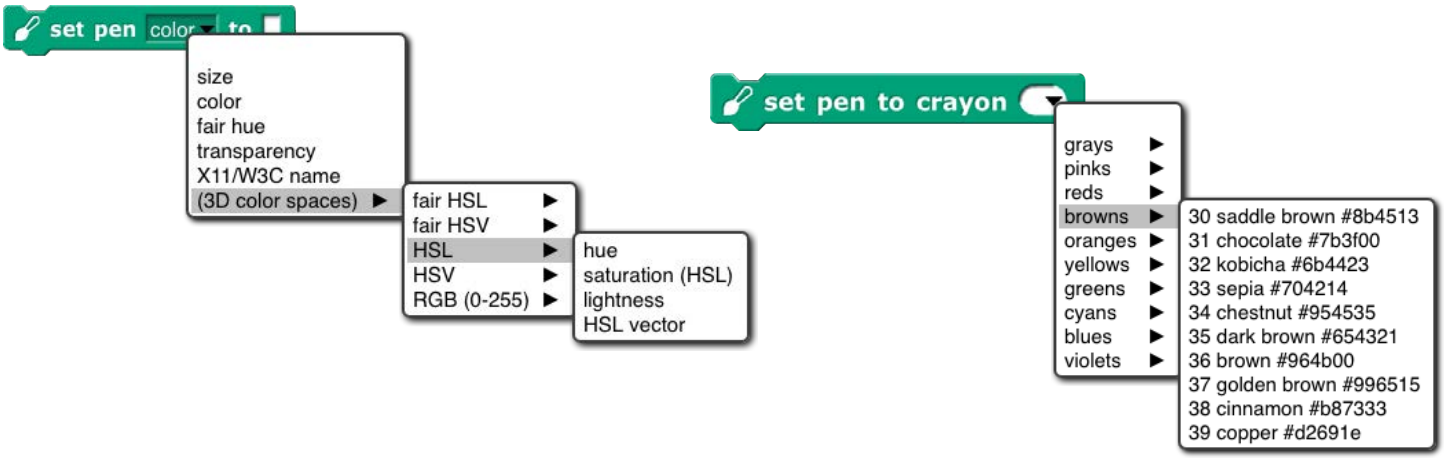
If you are a color professional—a printer, a web designer, a graphic designer, an artist—then you need to understand all this. It can also be interesting to learn about. For example, there are colors that you can see but your computer display can't generate. If that intrigues you, look up color theory in Wikipedia.

Crayons and Colors

But if you just want some colors in your project, we provide a simple, one-dimensional subset of the available colors. Two subsets, actually: *crayons* and *colors*. Here's the difference:



The first row shows 100 distinct colors. They have names; this is pumpkin, and this is denim. You're supposed to think of them as a big box of 100 crayons. They're arranged in families: grays, pinks, reds, browns, oranges, etc. But they're not ordered within a family; you'd be unlikely to say "next crayon" in a project. (But look at the crayon spiral on page 126.) Instead, you'd think "I want this to look like a really old-fashioned photo" and so you'd find sepia as crayon number 33. You don't have to memorize the numbers! You can find them in a menu with a submenu for each family.



The crayon numbers are chosen so that skipping by 10 gives a box of ten crayons:

Alternatively, skipping by 5 gives a still-sensible set of twenty crayons:

The Snap! block colors (Motion blue, Looks purple, etc.) are included among the crayons.

The set of *colors* is arranged so that each color is visually near each of its neighbors. Bright and dark colors alternate for each family. Color numbers range from 0 to 99, like crayon numbers, but you can use fractional numbers to get as tiny a step as you like:

("As tiny as you like" isn't *quite* true because in the end, your color has to be rounded to integer RGB values for display.)

That's all you have to know about colors! *Crayons* for specific interesting ones, *colors* for gradual transformation from one color to the next. But there's a bit more to say, if you're interested. If not, stop here. (But look at the samples of the different scales on page 126.)

Both of these scales include the range of shades of gray, from black to white. Since black is the initial pen color, and black isn't a hue, Scratch and Snap! users would try to use SET COLOR to escape from black, and it wouldn't work. By including black in the same scale as other colors, we eliminate the Black Hole problem if people use only the recommended color scales.

More about Colors: Fair Hues and Shades

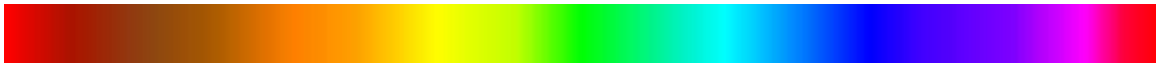
Several of the three-dimensional arrangements of colors use the concept of "hue," which more or less means where a color would appear in a rainbow (magenta, at the right, is a long story):

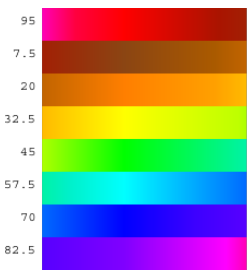
These are called "spectral" colors, after the "spectrum" of rainbow colors. But these colors aren't equally distributed. There's an awful lot of green, hardly any yellow, and just a sliver of orange. And no brown at all.

And this is already a handwave, because the range of colors that can be generated by RGB monitors doesn't include the *true* spectral colors. See Spectral color in Wikipedia for all the gory details.

This isn't a problem with the physics of rainbows. It's in the human eye and the human brain that certain ranges of frequency of light waves are lumped together as named colors. The eye is just "tuned" to recognize a wide range of colors as green. (See Rods and Cones.) And different human cultures give names to different color ranges. Nevertheless, in old Scratch projects, you'd say `change pen color by 1` and it'd take forever to reach a color that wasn't green.

For color professionals, there are good reasons to want to work with the physical rainbow hue layout. But for amateurs using a simplified, one-dimensional color model, there's no reason not to use a more programmer-friendly hue scale:

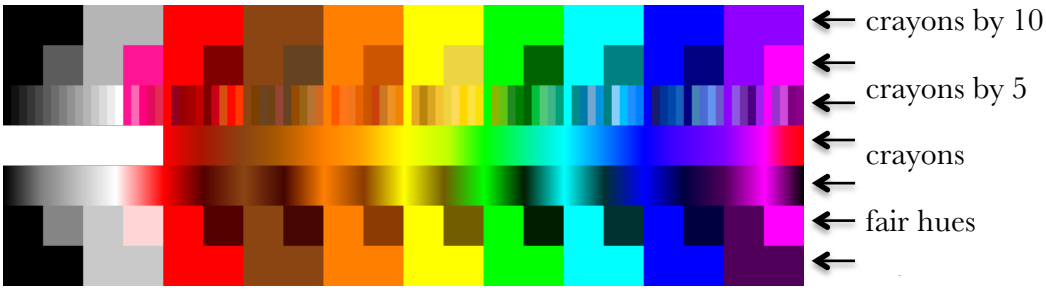




In this scale, each of the seven rainbow colors and brown get an equal share. (Red's looks too small, but that's because it's split between the two ends: hue 0 is pure red, brownish reds are to its right, and purplish reds are wrapped around to the right end.) We call this scale "fair hue" because each color family gets a fair share of the total hue range. (By the way, you were probably taught "… green, blue, indigo, violet" in school, but it turns out that color names were different in Isaac Newton's day, and the color he called "blue" is more like modern cyan, while his "indigo" is more like modern blue. See Wikipedia Indigo.)

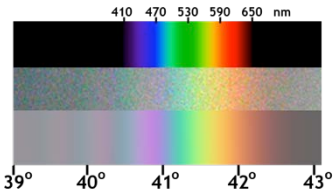
Our *color* scale is based on fair hues, adding a range of grays from black (color 0) to white (color 14) and also adding *shades* of the hue colors. (In color terminology, a *shade* is a darker version of a color; a lighter version is called a *tint*.) Why do we add shades but not tints? Partly because I find shades more exciting. A shade of red can be dark candy apple red or maroon, but a tint is just some kind of pink. This admitted prejudice is supported by an objective fact: Most projects are made on a white background, so dark colors stand out better than light ones.

So, in our color scale, colors 0 to 14 are kinds of gray; the remaining colors go through the fair hues, but alternating full-strength colors with shades.



This chart shows how the color scales discussed so far are related. Note that all scales range from 0 to 100; the fair hues scale has been compressed in the chart so that similar colors line up vertically. (Its dimensions are different because it doesn't include the grays at the left. Since there are eight color families, the pure, named spectral colors are at multiples of 100/8=12.5, starting with red=0.)

White is crayon 14 and color 14. This value was deliberately chosen *not* to be a multiple of 5 so that the every-fifth-crayon and every-tenth-crayon selections don't include it, so that all of the crayons in those smaller boxes are visible against a white stage background.



Among purples, the official spectral violet (crayon and color 90) is the end of the spectrum. Magenta, brighter than violet, isn't a pure spectral color at all. (In the picture at the left, the top part is the spectrum of white light spread out through a prism; the middle part is a photograph of a rainbow, and the bottom part is a digital simulation of a rainbow.) Magenta is a mixture of red and blue. (attribution: Wikipedia user Andys. CC BY-SA.)

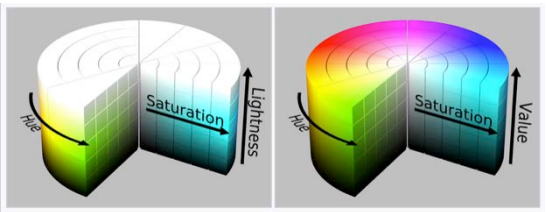
The light gray at color 10 is slightly different from crayon 10 just because of roundoff in computing crayon values. Color 90 is different from crayon 90 because in the purple family the darker color comes first; color and crayon 95 are the same. Otherwise, the colors, crayons, and (scaled) fair hues all agree at multiples of ten. These multiple-of-ten positions are the standard RGB primary and secondary colors, e.g., the yellow at color 50 is (255, 255, 0) in RGB. (Gray, brown, and orange don't have such well-defined RGB settings.)

The colors at odd multiples of five are generally darker shades than the corresponding crayons. The latter are often official named shades, e.g., teal, crayon 65, is a half-intensity shade of cyan. The odd-five colors, though, are often darker, since the usable color range in a given family has this as its darkest representative. The pink at color 15, though, is quite different from crayon 15, because the former is a pure tint of red, whereas the crayon, to get a more interesting pink, has a little magenta mixed in. Colors at multiples of five are looked up in a table; other color values are determined by linear interpolation in RGB space. (*Crayons* are of course all found by table lookup.)

Perceptual Spaces: HSL and HSV

RGB is the right way to think about colors if you're building or programming a display monitor; CMYK is the right way if you're building or programming a color printer. But neither of those coordinate systems is very intuitive if you're trying to understand what color *you see* if, for example, you mix 37% red light, 52% green, and 11% blue. The *hue* scale is one dimension of most attempts at a perceptual scale. The square at the right has pale blues along the top edge, dark blues along the right edge, various shades of gray toward the left, black at the bottom, and pure spectral blue in the top right corner. Although no other point in the square is pure blue, you can tell at a glance that no other spectral color is mixed with the blue.

Aside from hue, the other two dimensions of a color space have to represent how much white and/or black is mixed with the spectral color. (Bear in mind that "mixing black" is a metaphor when it comes to monitors. There really is black paint, but there's no such thing as black light.) One such space, HSV, has one dimension for the amount of color (vs. white), called *saturation,* and one for the amount of black, imaginatively called *value.* HSV stands for Hue-Saturation-Value. (I don't know why they couldn't think of a more descriptive name.) The *value* is actually measured backward from this description; that is, if value is 0, the color is pure black; if value is 100, then a saturation of 0 means all white, no spectral color; a saturation of 100 means no white at all. In the square in the previous paragraph, the *x* axis is the saturation and the *y* axis is the value. The entire bottom edge is black, but only the top left corner is white. HSV is the traditional color space used in Scratch and Snap!. `Set pen color` set the hue; `set pen shade` set the value. There was originally no Pen block to set the saturation, but there's a `set brightness effect` Looks block to control the saturation of the sprite's costume. (I speculate that the Scratch designers, like me, thought tints were less vivid than shades against a white background.)



attribution: Wikipedia user SharkD, CC BY-SA 3.0

But if you're looking at colors on a computer display, HSV isn't really a good match for human perception. Intuitively, black and white should be treated symmetrically. This is the HSL (hue-saturation-lightness) color space. *Saturation* is a measure of the *grayness* or *dullness* of a color (how close it comes to being on a black-and-white scale) and *lightness* measures *spectralness* with pure white at one end, pure black at the other end, and full spectral color in the middle. The *saturation* number is actually the opposite of grayness: 0 means pure gray, and 100 means pure spectral color, provided that the *lightness* is 50, midway between black and white. Colors with lightness other than 50 have some black and/or white mixed in, but saturation 100 means that the color is as fully saturated as it can be, given the amount of white or black needed to achieve that lightness. Saturation less than 100 means that *both white and black* are mixed with the spectral color. (Such mixtures are called *tones* of the spectral color. Perceptually, colors with saturation 100% don't look gray:  but colors with saturation 75% do: 

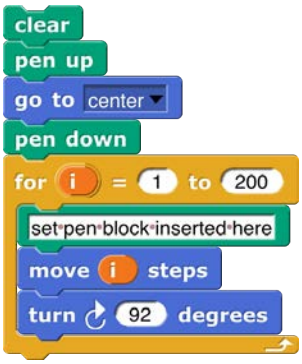
Note that HSV and HSL both have a dimension called "saturation," but *they're not the same thing!* In HSV, "saturation" means non-whiteness, whereas in HSL it means non-grayness (vividness).

Although traditional Scratch and Snap! use HSV in programs, they use HSL in the color picker. The horizontal axis is hue (fair hue, in this version) and the vertical axis is *lightness,* the scale with black at one end and white at the other end. It would make no sense to have only the bottom half of this selector (HSV Value) or only the top half (HSV Saturation). And, given that you can only fit two dimensions on a flat screen, it makes sense to pick HSL saturation (vividness) as the one to keep at 100%. (In this fair-hue picker, some colors appear twice: "spectral" (50% lightness) browns as shades (≈33% lightness) of red or orange, and shades of those browns.)

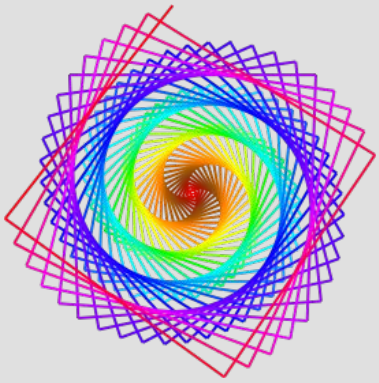
Software that isn't primarily about colors (so, *not* including Photoshop, for example) typically use HSV or HSL, with web-based software more likely to use HSV because that's what's built into the JavaScript programming language provided by browsers. But if the goal is to model human color perception, neither of these color spaces is satisfactory, because they assume that all full-intensity spectral colors are equally bright. But if you're like most people, you see spectral yellow as much brighter than spectral blue. There are better perceptual color spaces with names like L*u*v* and L*a*b* that are based on research with human subjects to determine true visual brightness. Wikipedia explains all this and more at HSL and HSV, where they recommend ditching both of these simplistic color spaces. ☺

tl;dr

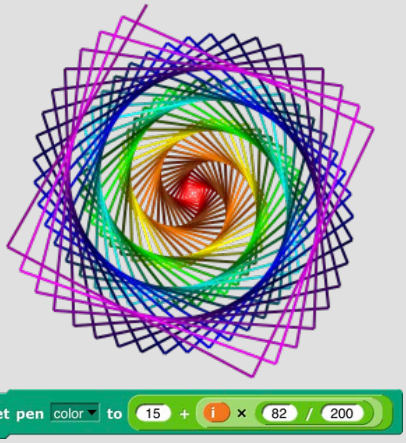
For normal people, Snap! provides three simple, one-dimensional scales: *crayons* for specific interesting colors, *colors* for a continuum of high-contrast colors with a range of hues and shading, and *fair hues* for a continuum without shading. For color nerds, it provides three-dimensional color spaces RGB, HSL, HSV, and fair-hue variants of the latter two. We recommend "fair HSL" for zeroing in on a desired color.



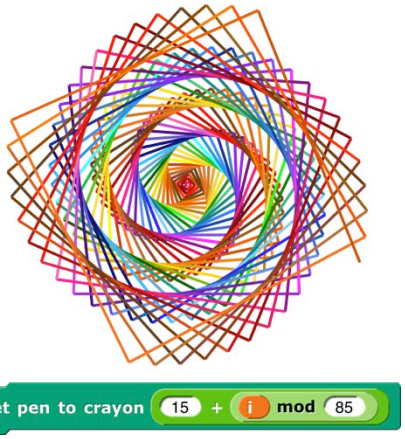
All colors.



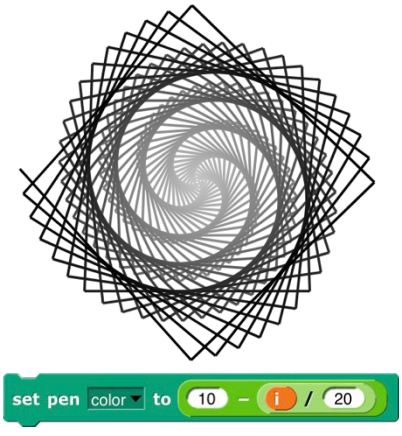
Fair hues.



Colors, no grays.

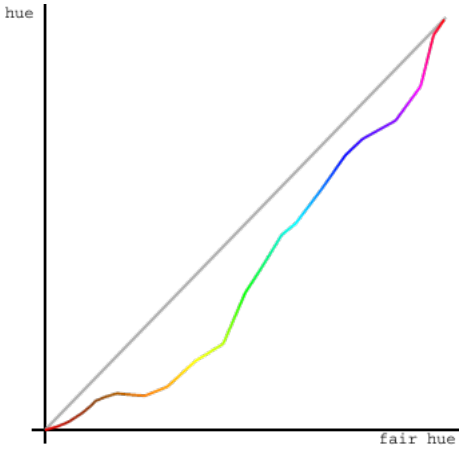


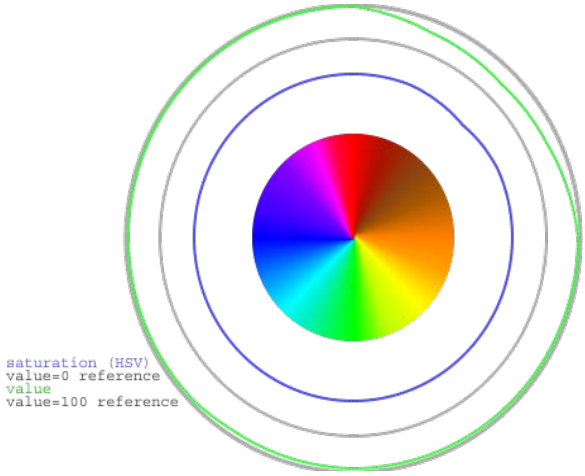
Crayons.



Just grays.

Subappendix: Geeky details on fair hue





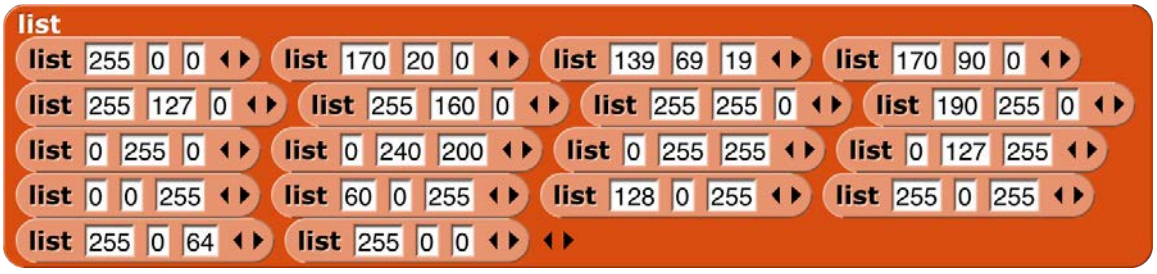
The left graph shows that, unsurprisingly, all of the brown fair hues make essentially no progress in real hue, with the orange-brown section actually a little retrograde, since browns are really shades of orange and so the real hues overlap between fair browns and fair oranges. Green makes up some of the distance, because there are too many green real hues and part of the goal of the fair hue scale is to squeeze that part of the hue spectrum. But much of the catching up happens very quickly, between pure magenta at fair hue 93.75 and the start of the purple-red section at fair hue 97. This abrupt change is unfortunate, but the alternatives involve either stealing space from red or stealing space from purple (which has to include both spectral violet and RGB magenta). The graph has discontinuous derivative at the table-lookup points, of which there are two in each color range, one at the pure-named-RGB colors at multiples of 12.5, and the other *roughly* halfway to the next color range, except for the purple range, which has lookup points at 87.5 (approximate spectral violet), 93.75 (RGB magenta), and 97 (turning point toward the red family). (In the color picker, blue captures cyan and purple space in dark shades. This, too, is an artifact of human vision.)

The right graph shows the HSV saturation and value for all the fair hues. Saturation is at 100%, as it should be in a hue scale, except for a very slight drop in part of the browns. (Browns are shades of orange, not tints, so one would expect full saturation, except that some of the browns are actually mixtures with related hues.) But value, also as expected, falls substantially in the browns, to a low of about 56% (halfway to black) for the "pure" brown at 45° (fair hue 12.5). But the curve is smooth, without inflection points other than that minimum-value pure brown.

"Fair saturation" and "fair value" are by definition 100% for the entire range of fair hues. This means that in the browns, the real saturation and value are the product (in percent) of the innate shading of the specific brown fair hue and the user's fair saturation/value setting. When the user's previous color setting was in a real scale and the new setting is in a fair scale, the program assumes that the previous saturation and value were entirely user-determined; when the previous color setting was in a brown fair hue and the new setting is also in a fair scale, the program remembers the user's intention from the previous setting. (Internal calculations are based on HSV, even though we recommend HSL to users, because HSV comes to us directly from the JavaScript color management implementation.) This is why the `set pen` block includes options for "fair saturation" and so on.

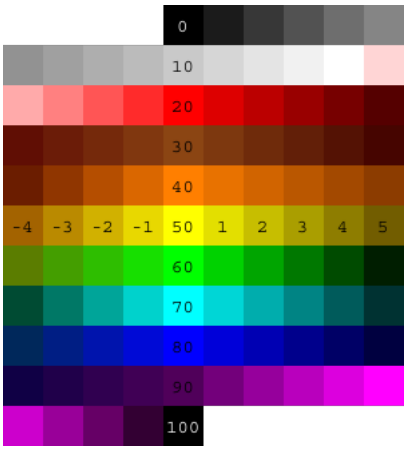
For the extra-geeky, here are the exact table lookup points (fair hue, [0,100]):

and here are the RGB settings at those points:



Subappendix: Geeky details on colors

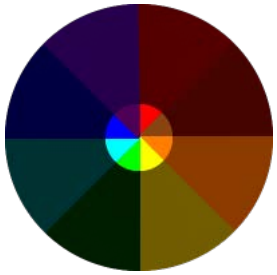
Here is a picture of integer color numbers, but remember that color numbers are continuous. (As usual, "continuous" values are ultimately converted to integer RGB colors, so there's really some granularity.) Colors 0-14 are continuously varying grayscale, from 0=black to 14=white. Colors 14+ε to 20 are linearly varying shades of pink, with RGB Red at color 20.



Beyond that point, in each color family, the multiple of ten color number in the middle is the RGB standard color of that family, in which each component is either 255 or 0. (Exceptions are brown, which is of course darker than any of those colors; orange, with its green component half-strength: [255, 127, 0]; and violet, discussed below.) The following multiple of five is the darkest color in that family, although not necessarily the same hue as the multiple of ten color. Colors between the multiple of ten and the following multiple of five are shades of colors entirely within the family. Colors in the four *before* the multiple of ten are mixtures of this family and the one before it. So, for example, in the green family, we have

55 Darkest yellow.
(55, 60) shades of yellow-green mixtures. As the color number increases, both the hue and the lightness (or value, depending on your religion) increase, so we get brighter and greener colors.
60 Canonical green, [0, 255, 0], whose W3C color name is "lime," not "green."
(60, 65) Shades of green. No cyan mixed in.
65 Darkest green.
(65,70) Shades of green-cyan mixtures.

In the color chart, all the dark colors look a lot like black, but they're quite different. Here are the darkest colors in each family.



Darkest yellow doesn't look entirely yellow. You might see it as greenish or brownish. As it turns out, the darkest color that really looks yellow is hardly dark at all. This color was hand-tweaked to look neither green nor brown to me, but ymmv.

In some families, the center+5 *crayon* is an important named darker version of the center color: In the red family, [0, 128, 0] is "maroon." In the cyan family, [0, 128, 128] is "teal." An early version of the color scale used these named shades as the center+5 color also. But on this page we use the word "darkest" advisedly: You can't find a darker shade of this family anywhere in the color scale, but you *can* find lighter shades. Teal is color number 73.1, $(70 + 5 \cdot \frac{255-128}{255-50})$, because darkest cyan, color 75, is [0, 50, 50]. The color number for maroon is left as an exercise for the reader.

The purple family is different from the others, because it has to include both spectral violet and extraspectral RGB magenta. Violet is usually given as [128, 0, 255], but that's much brighter than the violet in an actual spectrum (see page 3). We use [80, 0, 90], a value hand-tweaked to look as much as possible like the violet in rainbow photos, as color 90. (*Crayon* 90 is [128, 0, 255].) Magenta, [255, 0, 255], is color 95. This means that the colors get *brighter,* not darker, between 90 and 95. The darkest violet is actually color 87.5, so it's bluer than standard violet, but still plainly a purple and not a blue. It's [39,0,76]. It's *not* hand-tweaked; it's a linear interpolation between darkest blue, [0, 0, 64], and the violet at color 90. I determined by experiment that color 87.5 is the darkest one that's still unambiguously purple. (According to Wikipedia, "violet" names only the spectral color, while "purple" is the name of the whole color family.)

Here are the reference points for color numbers that are multiples of five, except for item 4, which is used for color 14, not color 15:

The very pale three-input list blocks are for color numbers that are odd multiples of five, generally the "darkest" members of each color family. (The block colors were adjusted in Photoshop; don't ask how to get blocks this color in Snap!.)

Appendix B. APL features

The book *A Programming Language* was published by mathematician Kenneth E. Iverson in 1962. He wanted a formal language that would look like what mathematicians write on chalkboards. The then-unnamed language would later take its name from the first letters of the words in the book's title. It was little-known until 1964, when a formal description of the just-announced IBM System/360 in the *IBM Systems Journal* used APL notation. (Around the same time, Iverson's associate Adin Falkoff gave a talk on APL to a New York Association for Computing Machinery chapter, with an excited 14-year-old Brian Harvey in the audience.) But it wasn't until 1966 that the first public implementation of the language for the System/360 was published by IBM. (It was called "APL\360" because the normal slash character / represents the "reduce" operator in APL, while backslash is "expand.")

The crucial idea behind APL is that mathematicians think about collections of numbers, one-dimensional *vectors* and two-dimensional *matrices,* as valid objects in themselves, what computer scientists later learned to call "first class data." A mathematician who wants to add two vectors writes $v_1 + v_2$, not "for i = 1 to length(v1), result[i]=v1[i]+v2[i]." Same for a programmer using APL.

There are three kinds of function in APL: scalar functions, mixed functions, and operators. A *scalar function* is one whose natural domain is individual numbers or text characters. A *mixed function* is one whose domain includes arrays (vectors, matrices, or higher-dimensional collections). In Snap!, scalar functions are generally found in the green Operators palette, while mixed functions are in the red Lists section. The third category, confusingly for Snap! users, is called *operators* in APL, but corresponds to what we call higher order functions: functions whose domain includes functions.

Snap! hyperblocks are scalar functions that behave like APL scalar functions: they can be called with arrays as inputs, and the underlying function is applied to each number in the arrays. (If the function is *monadic,* meaning that it takes one input, then there's no complexity to this idea. Take the square root of an array, and you are taking the square root of each number in the array. If the function is *dyadic,* taking two inputs, then the two arrays must have the same shape. Snap! is more forgiving than APL; if the arrays don't agree in number of dimensions, called the *rank* of the array, the lower-rank array is matched repeatedly with subsets of the higher-rank one; if they don't agree in length along one dimension, the result has the shorter length and some of the numbers in the longer-length array are ignored. An exception in both languages is that if one of the two inputs is a scalar, then it is matched with every number in the other array input.)

As explained in Section I.G, this termwise extension of scalar functions is the only APL-like feature built into Snap! itself, other than an extension of the **item** block to address multiple dimensions. The APL library extends the implementation of APL features to include a few missing scalar functions and several missing mixed functions and operators.

Programming in APL really is *very* different in style from programming in other languages, even Snap!. This appendix can't hope to be a complete reference for APL, let alone a tutorial. If you're interested, find one of those in a library or a (probably used) bookstore, read it, and *do the exercises.* Sorry to sound like a teacher, but the notation is sufficiently weird as to take a lot of practice before you start to think in APL.

A note on versions: There is a widely standardized APL2, several idiosyncratic extensions, and a successor language named J. The latter uses plain ASCII characters, unlike the ones with APL in their names, which use the mathematician's character set, with Greek letters, typestyles (boldface and/or italics in books; underlined, upper case, or lower case in APL) as loose type declarations, and symbols not part of anyone's alphabet, such as

⌊ for floor and ⌈ for ceiling. To use the original APL, you needed expensive special computer terminals. (This was before you could download fonts in software. Today the more unusual APL characters are in Unicode at U+2336 to U+2395.) The character set was probably the main reason APL didn't take over the world. APL2 has a lot to recommend it for Snap! users, mainly because it moves from the original APL idea that all arrays must be uniform in dimension, and the elements of arrays must be numbers or single text characters, to our idea that a list can be an element of another list, and that such elements don't all have to have the same dimensions. Nevertheless, its mechanism for allowing both old-style APL arrays and more general "nested arrays" is complicated and hard for an APL beginner (probably all but two or three Snap! users) to understand. So we are starting with the original APL. If it turns out to be wildly popular, we may decide later to include APL2 features.

Here are some of the guiding ideas in the design of the APL library:

Goal: Enable interested Snap! users to learn the feel and style of APL programming. It's really worth the effort. For example, we didn't hyperize the = block because Snap! users expect it to give a single yes-or-no answer about the equality of two complete structures, whatever their types and shapes. In APL, = is a scalar function; it compares two numbers or two characters. How could APL users live without the ability to ask if two *structures* are equal? Because in APL you can say ∧/,a=b to get that answer. Reading from right to left, a=b reports an array of Booleans (represented in APL as 0 for False, 1 for True); the comma operator turns the shape of the array into a simple vector; and ∧/ means "reduce with ***and***"; "reduce" is our ***combine*** function. That six-character program is much less effort than the equivalent

in Snap!. Note in passing that if you wanted to know *how many* corresponding elements of the two arrays are equal, you'd just use +/ instead of ∧/. Note also that our APLish blocks are a little verbose, because they include up to three notations for the function: the usual Snap! name (e.g., **flatten**), the name APL programmers use when talking about it (**ravel**), and, in yellow type, the symbol used in actual APL code (,). We're not consistent about it; it didn't seem necessary to spell out ***combine*** **(*reduce*)** in naming that block. And **LCM (*and*)** is different even though it has two names; it turns out that if you represent Boolean values as 0 and 1, then the algorithm to compute the least common multiple of two integers computes the ***and*** function if the two inputs happen to be Boolean. Including the APL symbols serves two purposes: the two or three Snap! users who've actually programmed in APL will be sure what function they're using, but more importantly, the ones who are reading an APL tutorial while building programs in Snap! will find the block that matches the APL they're reading.

Goal: Bring the best and most general APL ideas into "mainstream" Snap! programming style. Media computation, in particular, becomes much simpler when scalar functions can be applied to an entire picture or sound. Yes, ***map*** provides essentially the same capability, but the notation gets complicated if you want to map over columns rather than rows. Also, Snap! lists are fundamentally one-dimensional, but real data often have more dimensions. A Snap! programmer has to be thinking all the time about the convention that we represent a matrix as a list of rows, each of which is a list of individual cells. That is, row 23 of a spreadsheet is **item 23 *of* spreadsheet**, but column 23 is **map (item 23 *of* _) *over* spreadsheet**. APL treats rows and columns more symmetrically.

Non-goal: Allow programs written originally in APL to run in Snap! essentially unchanged. For example, in APL the atomic text unit is a single character, and strings of characters are lists. We treat a text

string as scalar, and that isn't going to change. Because APL programmers rarely use conditionals, instead computing functions involving arrays of Boolean values to achieve the same effect, the notation they do have for conditionals is primitive (in the sense of Paleolithic, not in the sense of built in). We're not changing ours.

Non-goal: Emulate the terse APL syntax. It's too bad, in a way; as noted above, the terseness of expressing a computation affects APL programmers' sense of what's difficult and what isn't. But you can't say "terse" and "block language" in the same sentence. Our whole *raison d'être* is to make it possible to build a program without having to memorize the syntax or the names of functions, and to allow those names to be long enough to be self-documenting. And APL's syntax has its own issues, of which the biggest is that it's hard to use functions with more than two inputs; because most mathematical dyadic functions use infix notation (the function symbol between the two inputs), the notion of "left argument" and "right argument" is universal in APL documentation. The thing people most complain about really doesn't turn out to be a problem: there is no operator precedence, like the multiplication-before-addition rule in normal arithmetic notation. Function grouping is strictly right to left, so `2×3+4` means two times seven, not six plus four. That takes some getting used to, but it really doesn't take long if you immerse yourself in APL. The reason is that there are too many infix operators for people to memorize a precedence table. But in any case, block notation eliminates the problem, especially with Snap!'s zebra coloring. You can see the grouping from which block is inside which other block's input slot. Another problem with APL's syntax is that it bends over backward not to have reserved words, as opposed to Fortran, its main competition back then. So the dyadic ○ "circular functions" function uses the left argument to select a trig function. `1○x` is sin(x), `2○x` is cos(x), and so on. `¯1○x` is arcsin(x). What's `0○x`? Glad you asked; it's $\sqrt{1-x^2}$.

Boolean values

Snap! uses distinct Boolean values **true** and **false** that are different from other data types. APL uses 1 and 0, respectively. The APL style of programming depends heavily on doing arithmetic on Booleans, although their conditionals insist on only 0 or 1 in a Boolean input slot, not other numbers. Snap! *arithmetic* functions treat **false** as 0 and **true** as 1, so our APL library tries to report Snap! Boolean values from predicate functions.

Scalar functions

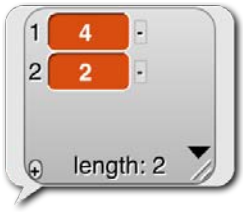
These are the scalar functions in the APL library. Most of them are straightforward to figure out. The **scalar =** block provides an APL-style version of **=** (and other exceptions) as a hyperblock that extends termwise to arrays. **Join**, the only non-predicate non-hyper scalar primitive, has its own **scalar join** block. `7 deal 52` reports a random vector of seven numbers from 1 to 52 with no repetitions, as in dealing a hand of cards. **Signum** of a number reports 1 if the number is positive, 0 if it's zero, or -1 if it's negative. **Roll 6** reports a random roll of a six-sided die. To roll 8 dice, use 

, which would look much more pleasant as `?8⍴6`. But perhaps our version is more instantly readable by someone who didn't grow up with APL. All the library functions have help messages available.

Mixed functions

Mixed functions include lists in their natural domain or range. That is, one or both of its inputs *must* be a list, or it always reports a list. Sometimes both inputs are naturally lists; sometimes one input of a dyadic mixed function is naturally a scalar, and the function treats a list in that input slot as an implicit **map**, as for scalar functions. This means you have to learn the rule for each mixed function individually.

The **shape of** function takes any input and reports a vector of the maximum size of the structure along each dimension. For a vector, it returns a list of length 1 containing the **length of** the input. For a matrix, it returns a two-item list of the number of rows and number of columns of the input. And so on for higher dimensions. If the input isn't a list at all, then it has zero dimensions, and **shape of** reports an empty vector.



Rank of isn't an actual APL primitive, but the composition ρρ (shape of shape of a structure), which reports the number of dimensions of the structure (the length of its shape vector), is too useful to omit. (It's very easy to type the same character twice on the APL keyboard, but less easy to drag blocks together.)

Reshape, the dyadic version of the **shape** function, takes a shape vector (such as **shape** might report) on the left and any structure on the right. It ignores the shape of the right input, stringing the atomic elements into a vector in row-major order (that is, all of the first row left to right, then all of the second row, etc.). It then reports an array with the shape specified by the first input containing the items of the second:

2	A	B	C
1	1	2	3
2	4	5	6

If the right input has more atomic elements than are required by the left-input shape vector, the excess are ignored without reporting an error. If the right input has too *few* atomic elements, the process of filling the reported array starts again from the first element. This is most useful in the specific case of an atomic right input, which produces an array of any desired shape all of whose atomic elements are equal. But other cases are sometimes useful too:

3	A	B	C
1	1	0	0
2	0	1	0
3	0	0	1

ID ← {(⍵,⍵)ρ1,⍵ρ0}

Flatten takes an arbitrary structure as input and reports a vector of its atomic elements in row-major order. Lispians call this flattening the structure, but APLers call it "ravel" because of the metaphor of pulling on a ball of yarn, so what they really mean is "unravel." (But the snarky sound of that is uncalled-for, because a more advanced version that we might implement someday is more like raveling.) One APL idiom is to apply this to a scalar in order to turn it into a one-element vector, but we can't use it that way because you can't type a scalar value into the List-type input slot.

Catenate is like our primitive **append**, with two differences: First, if either input is a scalar, it is treated like a one-item vector. Second, if the two inputs are of different rank, the **catenate** function is recursively **map**ped over the higher-rank input:

2	A	B	C	D	E	F
1	1	2	3	20	30	40
2	4	5	6	20	30	40

Catenate vertically is similar, but it adds new rows instead of adding new columns.

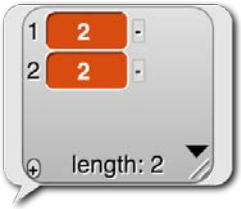
Integers (I think that's what it stands for, although APLers just say "iota") takes a positive integer input and reports a vector of the integers from 1 to the input. This is an example of a function classed as "mixed" not because of its domain but because of its range. The difference between this block and the primitive **numbers from** block is in its treatment of lists as inputs. **Numbers from** is a hyperblock, applying itself to each item of its input list:

3	A	B	C
1	1		
2	1	2	
3	1	2	3

Iota has a special meaning for list inputs: The input must be a shape vector; the result is an array with that shape in which each item is a list of the indices of the cell along each dimension. A picture is worth 10^3 words, but **Snap!** isn't so good at displaying arrays with more than two dimensions, so here we reduce each cell's index list to a string:

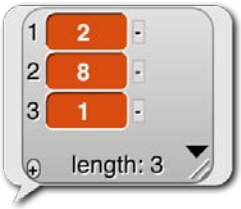
2	A	B	C
1	1,1	1,2	1,3
2	2,1	2,2	2,3

Dyadic **iota** is like the **index of** primitive except for its handling of multi-dimensional arrays. It looks only for atomic elements, so a vector in the second input doesn't mean to search for that vector as a row of a matrix, which is what it means to **index of**, but rather to look separately for each item of the vector, and report a list of the locations of each item. If the first input is a multi-dimensional array, then the location of an item is a vector with the indices along each row.

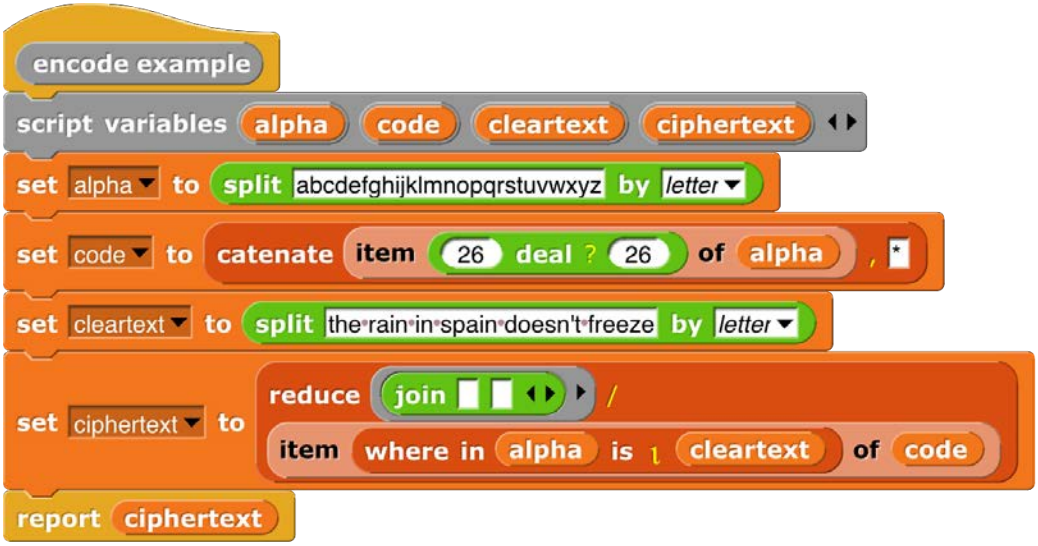


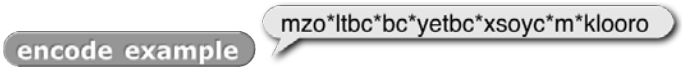
In this example, the 4 is in the second row, second column. (This is actually an extension of APL iota, which is more like a hyperized **index of**.) Generalizing, if the rank of the second input is less than the rank of the first input by two or more, then iota looks for the entire second input in the first input. The reported position is a vector whose length is equal to the difference between the two ranks. If the rank of the second input is one less than the rank of the first, the reported value is a scalar, the index of the entire second input in the first.

However, if the two ranks are equal, then the block is hyperized; each item of the second input is located in the first input. As the next example shows, only the first instance of each item is found (e.g., the 1 in position 2, not the 1 in position 4); if an item does not occur in the left input, what is reported is one more than the length of the left input (here, 8).



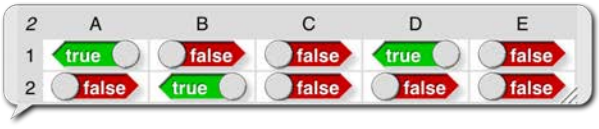
Why the strange design decision to report length+1 when something isn't found, instead of a more obvious flag value such as 0 or **false**? Here's why:



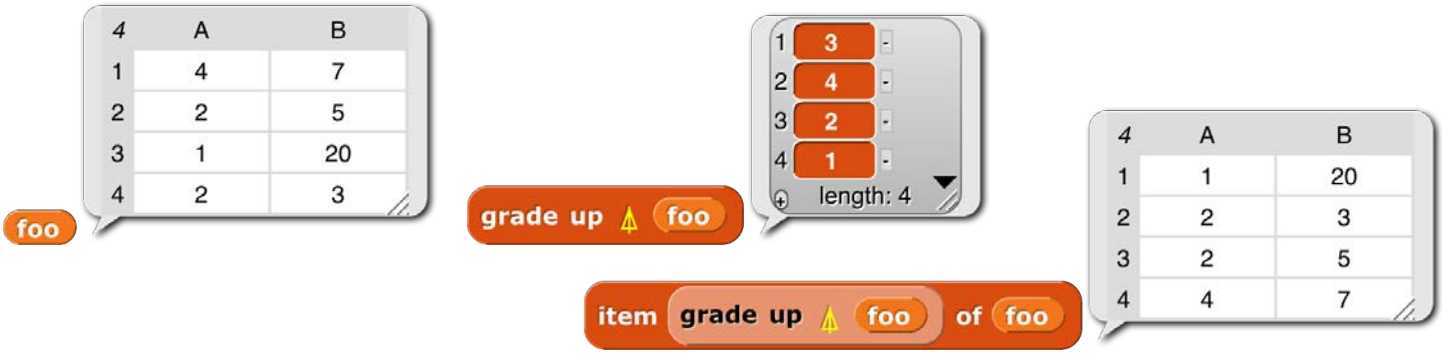


Note that ***code*** has 27 items, not 26. The asterisk at the end is the ciphertext is the translation of all non-alphabet characters (spaces and the apostrophe in "doesn't"). This is a silly example, because it makes up a random cipher every time it's called, and it doesn't report the cipher, so the recipient can't decipher the message. And you wouldn't want to make the spaces in the message so obvious. But despite being silly, the example shows the benefit of reporting length+1 as the position of items not found.

The ***contained in*** block is like a hyperized ***contains*** with the input order reversed. It reports an array of Booleans the same shape as the left input. The shape of the right input doesn't matter; the block looks only for atomic elements.



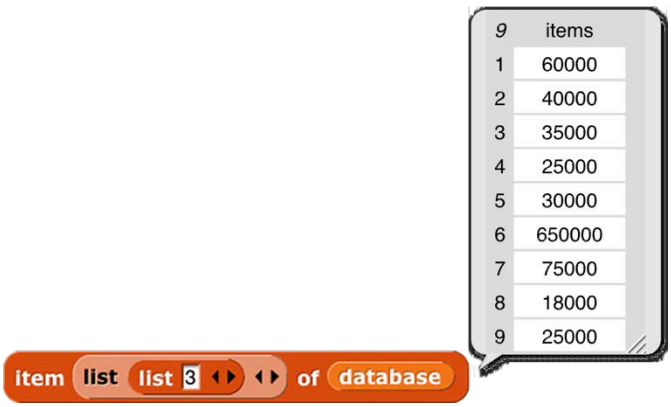
The blocks ***grade up*** and ***grade down*** are used for sorting data. Given an array as input, it reports a vector of the indices in which the items (the rows, if a matrix) should be rearranged in order to be sorted. This will be clearer with an example:



The result from ***grade up*** tells us that item 3 of **foo** comes first in sorted order, then item 4, then 2, then 1. When we actually select items of **foo** based on this ordering, we get the desired sorted version. The result reported by ***grade down*** is almost the reverse of that from ***grade up***, but not quite, if there are equal items in the list. (The sort is stable, so if there are equal items, then whichever comes first in the input list will also be first in the sorted list.)

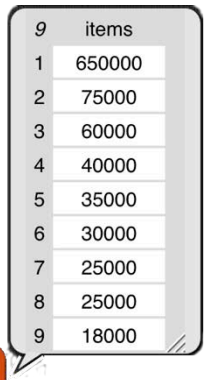
Why this two-step process? Why not just have a ***sort*** primitive in APL? One answer is that in a database application you might want to sort one array based on the order of another array:

This is the list of employees of a small company. (Taken from *Structure and Interpretation of Computer Programs* by Abelson and Sussman. Creative Commons licensed.) Each of the smaller lists contains a person's name, job title, and yearly salary. We would like to sort the employees' names in big-to-small order of salary. First we extract column 3 of the database, the salaries:



Then we use ***grade down*** to get the reordering indices:

At this point we *could* use the index vector to sort the salaries:



But what we actually want is a list of *names,* sorted by salary:

By taking the index vector from ***grade down*** of column 3 and telling **item** to apply it to column 1, we get what we set out to find. As usual the code is more elegant in APL: ***database[⍒database[;3];1]***.

In case you've forgotten, 

would select the third *row* of the database; we need the extra list wrapper 

to select by columns rather than by rows.

Select (***take***) or select all but (***drop***) the first (n>0) or last (n<0) |n| items from a vector, or rows from a matrix. Alternatively, if the left input is a two-item vector, select rows with the first item and columns with the second.

The ***compress*** block selects a subset of its right input based on the Boolean values in its left input, which must be a vector of Booleans whose length equals the length of the array (the number of rows, for a matrix) in the right input. The block reports an array of the same rank as the right input, but containing only those rows whose corresponding Boolean value is **true**. The columns version ≠ is the same but selecting columns rather than selecting rows.

A word about the possibly confusing names of these blocks: There are two ways to think about what they do. Take the standard / version, to avoid talking about both at once. One way to think about it is that it selects some of the rows. The other way is that it shortens the columns. For Lispians, which includes you since you've learned about ***keep***, the natural way to think about / is that it keeps some of the rows. Since we represent a matrix as a list of rows, that also fits with how this function is implemented. (Read the code; you'll find a ***keep*** inside.) But APL people think about it the other way, so when you read APL documentation, / is described as operating on the last dimension (the columns), while ≠ is described as operating on rows. We were more than a month into this project before I understood all this. You get long block names so it won't take you a month!

Don't confuse this block with the ***reduce*** block, whose APL symbol is also a slash. In that block, what comes to the left of the slash is a dyadic combining function; it's the APL equivalent of ***combine***. This block is more nearly equivalent to ***keep***. But ***keep*** takes a predicate function as input, and calls the function for each item of the second input. With ***compress***, the predicate function, if any, has already been called on all the items of the right input in parallel, resulting in a vector of Boolean values. This is a typical APL move; since hyperblocks are equivalent to an implicit ***map***, it's easy to make the vector of Booleans, because any scalar function, including predicates, can be applied to a list instead of to a scalar. The reason both blocks use the / character is that both of them reduce the size of the input array, although in different ways.

The ***reverse row order***, ***reverse column order***, and ***transpose*** blocks form a group: the group of reflections of a matrix. The APL symbols are all a circle with a line through it; the lines are the different axes of reflection. So the ***reverse row order*** block reverses which row is where; the ***reverse column order*** block reverses which column is where; and the ***transpose*** block turns rows into columns and vice versa:

3	A	B	C	D
1	9	10	11	12
2	5	6	7	8
3	1	2	3	4

3	A	B	C	D
1	4	3	2	1
2	8	7	6	5
3	12	11	10	9

4	A	B	C
1	1	5	9
2	2	6	10
3	3	7	11
4	4	8	12

Except for **reverse**, these work only on full arrays, not ragged-right lists of lists, because the result of the other two would be an array in which some rows had "holes": items 1 and 3 exist, but not item 2. We don't have a representation for that. (In APL, all arrays are full, so it's even more restrictive.)

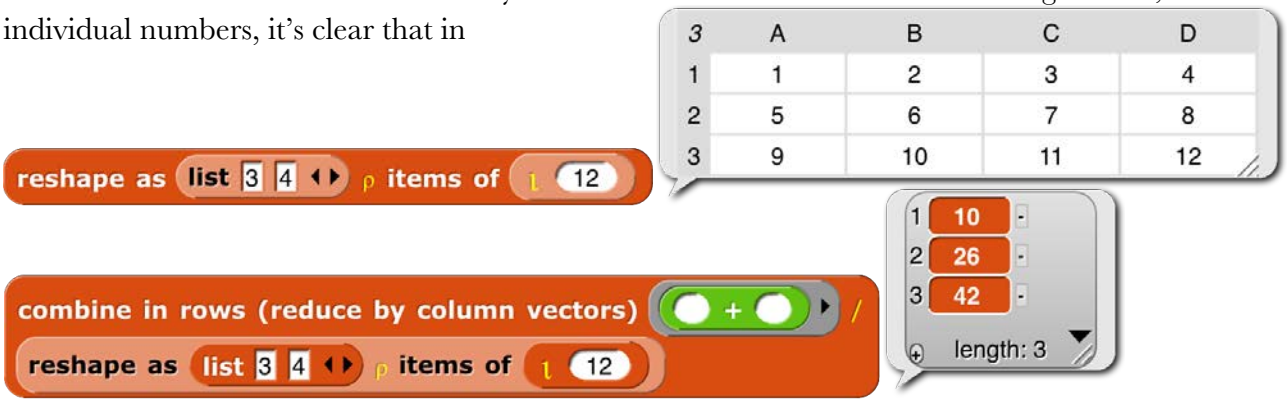
Higher order functions

The final category of function is operators—APL higher order functions. APL has no explicit **map** function, because the hyperblock capability serves much the same need. But APL2 did add an explicit **map**, which we might get around to adding to the library next time around. Its symbol is ¨ (diaeresis or umlaut).

The APL equivalent of **keep** is **compress**, but it's not a higher order function. You create a vector of Booleans (0s and 1s, in APL) before applying the function to the array you want to compress.

But APL does have a higher order version of **combine**:

The **reduce** block works just like **combine**, taking a dyadic function and a list. The **/** version translates each row to a single value; the **⌿** version translates each column to a single value. That's the only way to think about it from the perspective of combining individual elements: you are adding up, or whatever the function is, the numbers in a single row (/) or in a single column (⌿). But APLers think of a matrix as made up of vectors, either row vectors or column vectors. And if you think of what these blocks do as adding vectors, rather than adding individual numbers, it's clear that in



the *vector* (10, 26, 42) is the sum of *column vectors* (1, 5, 9)+(2, 6, 10)+(3, 7, 11)+(4, 8, 12). In pre-6.0 Snap!, we'd get the same result this way:

mapping over the *rows* of the matrix, applying **combine** to each row. Combining rows, reducing column vectors.

The ***outer product*** block takes two arrays (vectors, typically) and a dyadic scalar function as inputs. It reports an array whose rank is the sum of the ranks of the inputs (so, typically a matrix), in which each item is the result of applying the function to an atomic element of each array. The third element of the second row of the result is the value reported by the function with the second element of the left input and the third element of the right "input. (The APL symbol ∘. is pronounced "jot dot.") The way to think about this block is "multiplication table" from elementary school:

10	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

The ***inner product*** block takes two matrices and two operations as input. The number of columns in the left matrix must equal the number of rows in the right matrix. When the two operations are + and ×, this is the matrix multiplication familiar to mathematicians:

3	A	B
1	250	260
2	634	660
3	1018	1060

But other operations can be used. One common inner product is ∨.∧ ("or dot and") applied to Boolean matrices, to find rows and columns that have corresponding items in common.

The ***printable*** block isn't an APL function; it's an aid to exploring APL-in-Snap!. It transforms arrays to a compact representation that still makes the structure clear:

Experts will recognize this as the Lisp representation of list structure.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z