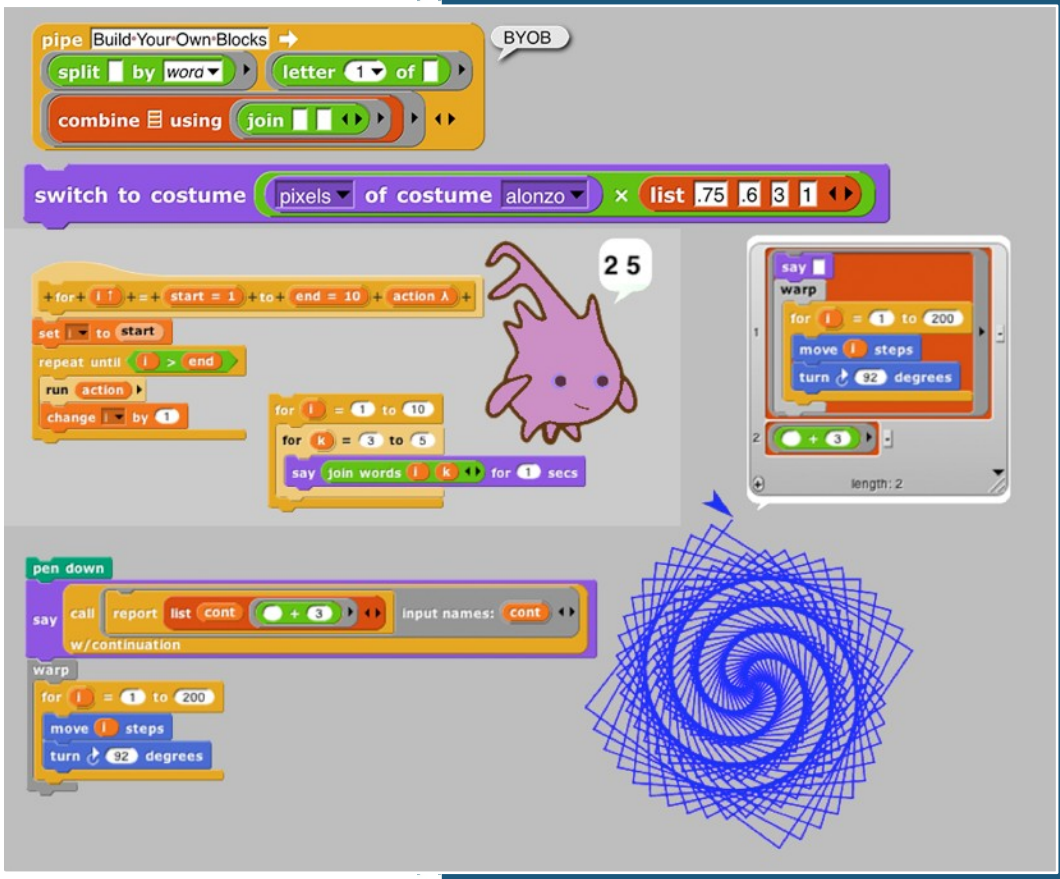
Build Your Own Blocks

8.0

SNAP! Reference Manual



Brian Harvey

Jens Mönig

Table of Contents

Acknowledgements

We have been extremely lucky in our mentors. Jens cut his teeth in the company of the Smallta
Alan Kay, Dan Ingalls, and the rest of the gang who invented personal computing and object ori
programming in the great days of Xerox PARC. He worked with John Maloney, of the MIT Scratc
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,
Snap! ***Our earlier version, BYOB, was a direct modification of the Scratch source code.*** **Snap!**
complete rewrite, but its code structure and its user interface remain deeply indebted to Sc
the Scratch Team, who could have seen us as rivals, have been entirely supportive and welc

Brian grew up at the MIT and Stanford Artificial Intelligence Labs, learning from Lisp inventor Jo
McCarthy, Scheme inventors Gerald J. Sussman and Guy Steele, and the authors of the world's l
science book, *Structure and Interpretation of Computer Programs,* Hal Abelson and Gerald J. Sus
Sussman, among many other heroes of computer science. (Brian was also lucky enough, while
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." No
first class procedures, lexical scope, and first class continuations, **Snap!** ***Scheme disguised as Scratch.***

Four people have made such massive contributions to the implementation of Snap! that we ha
declared them members of the team: Michael Ball and Bernat Romagosa, in addition to contribu
throughout the project, have primary responsibility for the web site and cloud storage. Joan Guil
has contributed very careful and wise analysis of outstanding issues, including help in taming th
of translations to non-English languages. Jadga Hügle, has energetically contributed to online m
about Snap! and leading workshops for kids and for adults. Jens, Jadga, and Bernat are paid to
SAP, which also supports our computing needs.

We have been fortunate to get to know an amazing group of brilliant middle school(!) and high
through the Scratch Advanced Topics forum, several of whom (since grown up) have contributed
Kartik Chandra, Nathan Dinsmore, Connor Hudson, Ian Reynolds, and Deborah Servilla. Many m
contributed ideas and alpha-testing bug reports. UC Berkeley students who've contributed code
Dave. Kyle Hotchkiss, Ivan Motyashov, and Yuan Yuan. Contributors of translations are too num
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 G
1138596, 1143566, and 1441075; and in part by MioSoft, Arduino.org, SAP, and YC Research. A
findings, and conclusions or recommendations expressed in this material are those of the autho
necessarily reflect the views of the National Science Foundation or other funders.

Snap! Reference Manual
Version 8.0

Snap! (formerly BYOB) is an extended reimplementation of Scratch (**https://scratch.mit.edu**) 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 sui 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! fea something familiar in Scratch. It's very helpful to have some experience with Scratch before re manual, but not essential.

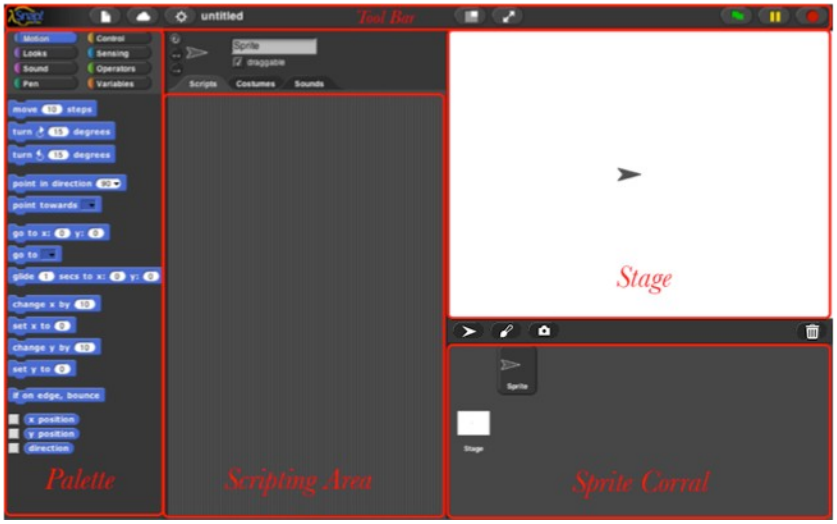
To run Snap!, open a browser window and connect to **https://snap.berkeley.edu/run**. The Sna web site at **https://snap.berkeley.edu** is not part of this manual's scope.

I. Blocks, Scripts, and Sprites

This chapter describes the features inherited from Scratch; experienced Scratch users can skip Section B.

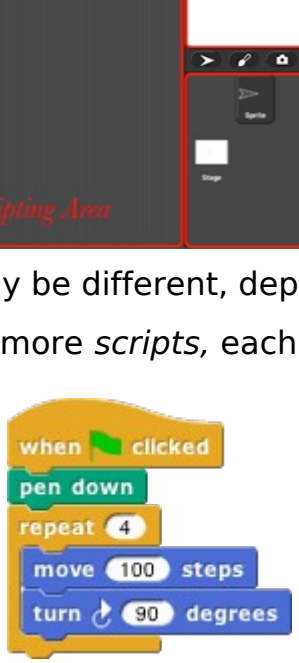
Snap! is a programming language—a notation in which you can tell a computer what you want it Unlike most programming languages, though, Snap! is a *visual* language; instead of writing a program usin keyboard, the Snap! programmer uses the same drag-and-drop interface familiar to computer users.

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

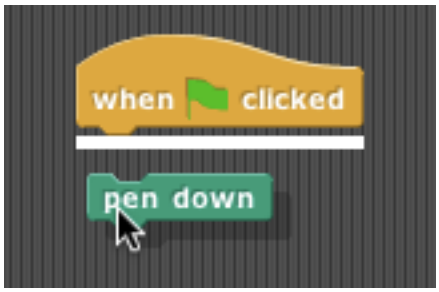


(The proportions of these areas may be different, depending on the size and shape of your brow

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



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



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

Hat Blocks and Command Blocks

At the top of the script is a *hat* block, which indicates when the script should be carried out. H
typically start with the word "**when**"; in the square-drawing example on page 5, the script shou
the green flag near the right end of the Stage toolbar is clicked. (The Stage toolbar 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
not, then the script will be run only if the user clicks on the script itself. A script can't have mor
block, and the hat block can be used only at the top of the script; its distinctive shape is meant
that[1].

The other blocks in our example script are *command* blocks. Each command block corresponc
that Snap! already knows how to carry out. For example 

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

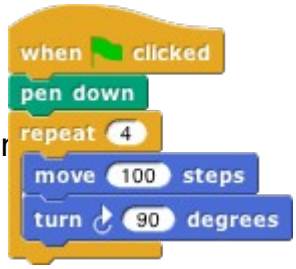
The number 10 in the **move** block above is called an *input* to the block. By clicking on the wh
type any number in place of the 10. The sample script on the previous page uses 100 as the in
see later that inputs can have non-oval shapes that accept values other than numbers. We'll al
can compute input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

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

, is subtly different from the others
sign is clicked, or when a project or sprite is loaded, this block doesn't test whether the condition in its hexagonal
the script beneath it will not run, until some *other* script in the project runs (because, for example, you click the g
generic **when** blocks are disabled, the stop sign will be square instead of octagonal.

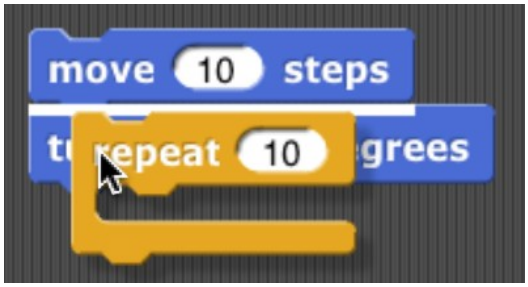
Most command blocks have that brick shape, but some, like the **repeat** block in the sample sc
Most C-shaped blocks are found in the Control palette. The slot inside the C shape is a special k
that accepts a *script* as the input.

In the sample scr

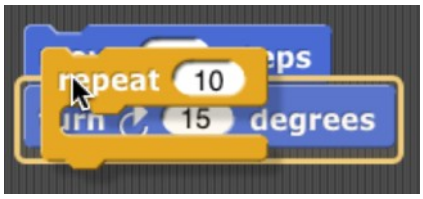


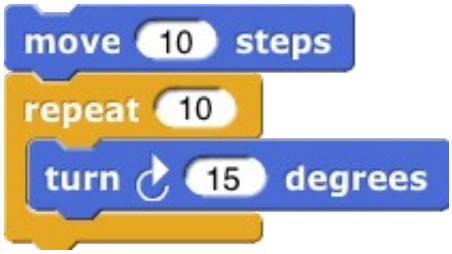
the **repeat** block has two inputs:
the number 4 and the script

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

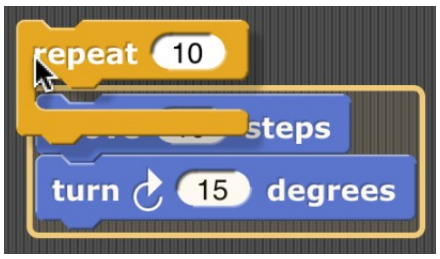


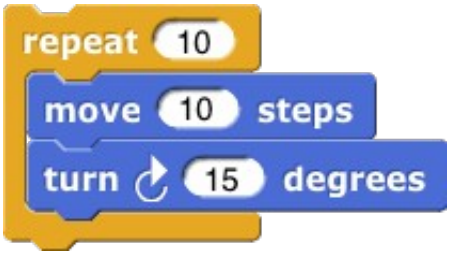
But if you see an orange halo and let go, the block will *wrap* around the haloed blocks:





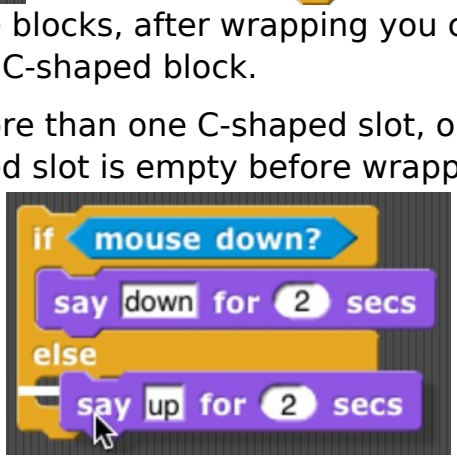
The halo will always extend from the cursor position to the bottom of the script:





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

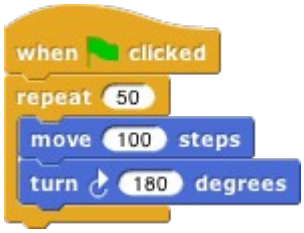
For "E-shaped" blocks with more than one C-shaped slot, only the first slot will wrap around exis
script, and only if that C-shaped slot is empty before wrapping. (You can fill the other slots by d
into the desired slot.)



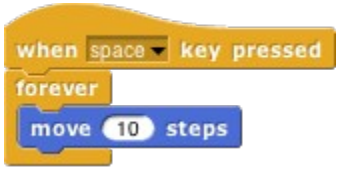
A. Sprites and Parallelism

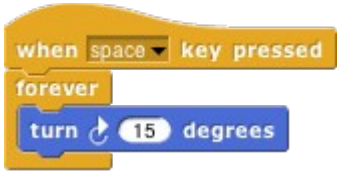
Just below the stage is the "new sprit utton . Click the button to add a new sprite to th new sprite will appear in a random position on the stage, with a random color, but always facing

Each sprite has its own scripts. To see the scripts for a particular sprite in the scripting area, c picture of that sprite in the *sprite corral* in the bottom right corner of the window. Try putting o 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 experiment illustrates the way different scripts can run in parallel. The turning and the moving together. Parallelism can be seen with multiple scripts of a single sprite also. Try this example:

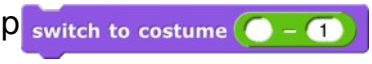


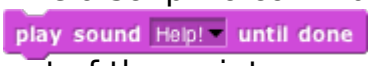


When you press the space key, the sprite should move forever in a circle, because the **move** ar run in parallel. (To stop the program, click the red stop sign at the right end of the tool bar.)

Costumes and Sounds

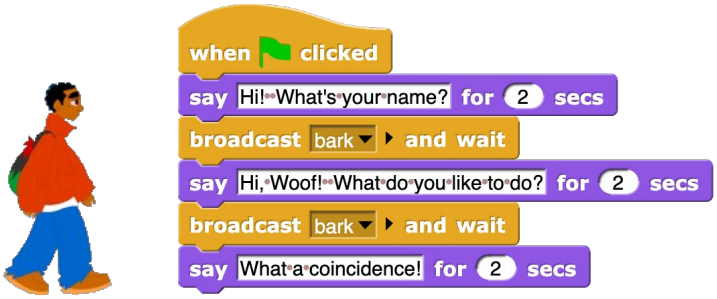
To change the appearance of a sprite, paint or import a new *costume* for it. To paint a costume **Costumes** tab above the scripting area, and click th aint button. The *Paint Editor* that appears is explained on page 128. There are three ways to import a costume. First select the desired spri corral. Then, one way is to click on the file icon in t toolbar, choose the "**Costumes...**" menu item. You will see a list of costumes from the public media library, and can choose one. The se costume stored on your own computer, is to click on the file icon and choose the "**Import...**" m can then select a file in any picture format (PNG, JPEG, etc.) supported by your browser. The th quicker if the file you want is visible on the desktop: Just drag the file into the Sprite window. In any of these cases, the scripting area will be replaced by something like this:

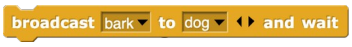
Just above this part of the window is a set of three tabs: Scripts, Costumes, and Sounds. You'll s
Costumes tab is now selected. In this view, the sprite's *wardrobe,* you can choose whether the
its Turtle costume or its Alonzo costume. (Alonzo is named after Alonzo Church, a
mathematician who invented the idea of procedures as data, the most important way in which S
from Scratch.) You can give a sprite as many costumes as you like, and then choose which it w
clicking in its wardrobe or by us   or block in a s
costume has a number as well as a name. The **next costume** block selects the next costume i
the highest-numbered costume it switches to costume 1. The Turtle, costume 0, is never chose
costume.) The Turtle costume is the only one that changes color to match a change in the spr
Protip  switches to the *previous* costume, wrapping like **next costu**

In addition to its costumes, a sprite can have *sounds;* the equivalent for sounds of the sprite's
its *jukebox.* Sound files can be imported in any format (WAV, OGG, MP3, etc.) supported by you
Two blocks accomplish the task of playing sounds. If you would like a script to continue running
sound is playing, use the  . In contrast, 
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 prog
sprites on stage will *interact* to tell a story, play a game, etc. Often one sprite will have to tell a
a script. Here's a simple example:



In the  block, the word "bark" is just an arbitrary name I made up.
on the downward arrowhead in that input slot, one of the choices (the only choice, the first time
then prompts you to enter a name for the new broadcast. When this block is run, the chosen m
every sprite, which is why the block is called "broadcast." (But if you click the right arrow after
name, the block bec  , and you 
send the message just to one sprite.) In this program, though, only one sprite has a script to ru
broadcast is sent, namely the dog. Because the boy's script uses **broadcast and wait** rather t
the boy doesn't go on to his next **say** block until the dog's script finishes. That's why the two s
talking, instead of both talking at once. In Chapter VII, "Object-Oriented Programming with Spri
see a more flexible way to send a message to a specific sprite using the **tell** and **ask** blocks.

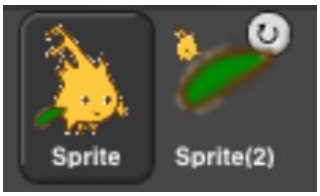
Notice, by the way, that the **say** block's first input slot is rectangular rather than oval. This me
can be any text string, not only a number. In text input slots, a space character is shown as a b
you can count the number of spaces between words, and in particular you can tell the differenc
empty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

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

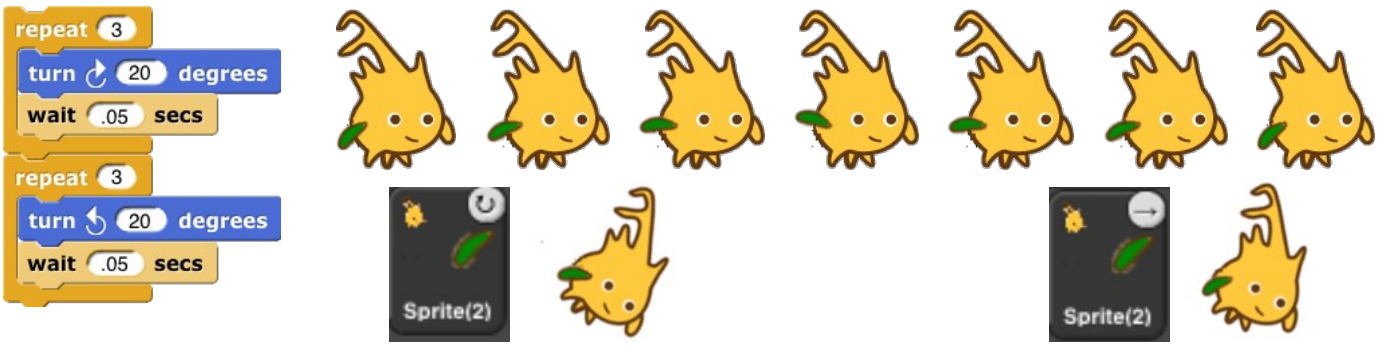
B. Nesting Sprites: Anchors and Parts

Sometimes it's desirable to make a sort of "super-sprite" composed of pieces that can move tog be separately articulated. The classic example is a person's body made up of a torso, limbs, an allows one sprite to be designated as the *anchor* of the combined shape, with other sprites as it sprite nesting, drag the sprite corral icon of a *part* sprite onto the stage display (not the sprite c desired *anchor* sprite. The precise place where you let go of the mouse button will be the attac part on the anchor.

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 thi make the relationship of the two sprites clearer, but in a real project they'd be the same color, p representing Alonzo's body, is the anchor; Sprite(2) is the arm. The icon for the anchor shows s up to three attached parts at the bottom. The icon for each part shows a small image of the an corner, and a *synchronous/dangling rotation flag* in the top right corner. In its initial, synchrono above, it means that the when the anchor sprite rotates, the part sprite also rotates as well as r the anchor. When clicked, it changes from a circular arrow to a straight arrow, and indicates tha anchor sprite rotates, the part sprite revolves around it, but does not rotate, keeping its original (The part can also be rotated separately, using its **turn** blocks.) Any change in the position or s is always extended to its parts. Also, cloning the anchor (see Section VII. B) will also clone all its



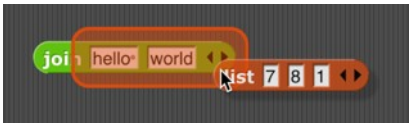
Top: turning the part: the green arm. Bottom: turning the anchor, with the arm synchronou

C. Reporter Blocks and Expressions

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

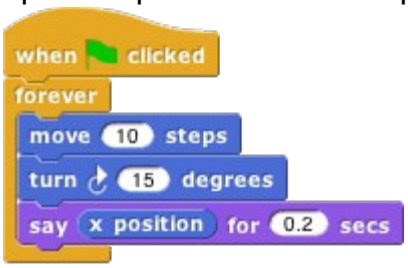
When you drag a reporter block over another block's input slot, a white "halo" appears around t
analogous to the white line that appears when snapping command blocks together:

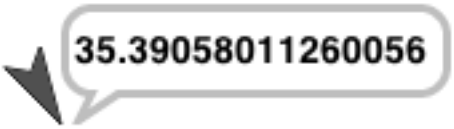
Don't drop the input over a *red* halo:



That's used for a purpose explained on page 68.

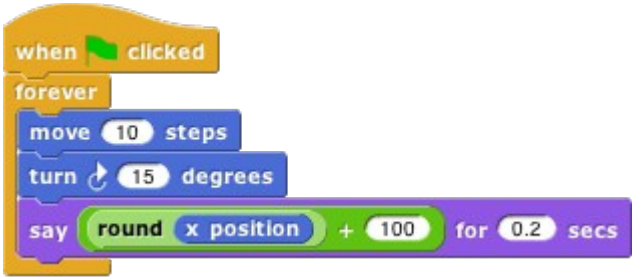
Here's a simple script that uses a reporter block:



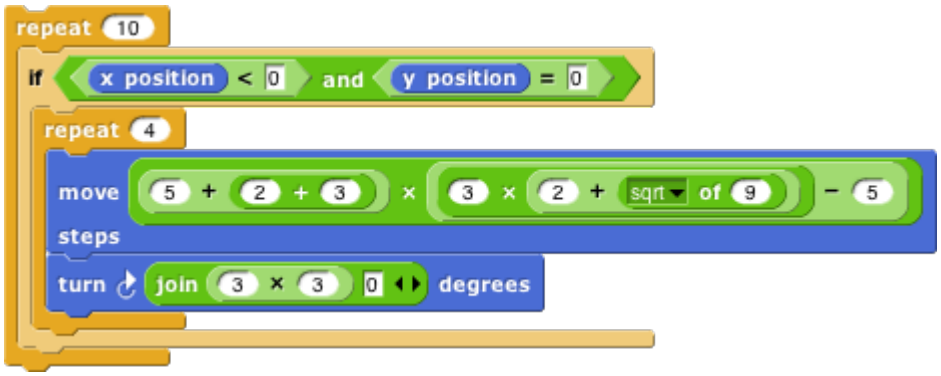


Here the **x position** reporter provides the first input to the **say** block. (The sprite's X position i
position, how far left (negative values) or right (positive values) it is compared to the center of t
Similarly, the Y position is measured vertically, in steps above (positive) or below (negative) the

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 **r**
the Operators palette, just like **+**, but in this script it's a lighter color with black lettering because
light and dark versions of the palette colors when a block is nested inside another block from th



This aid to readability is called *zebra coloring.*) A reporter block with its inputs, maybe including
blocks, such as round x position + 100 , is called an *expression.*

D. Predicates and Conditional Evaluation

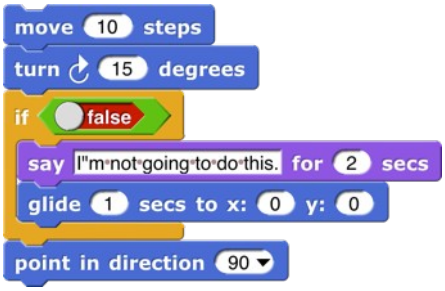
Most reporters report either a num , or a e . special kind of reporter that always reports **true** or **false**. Predicates have a hexagonal shape:

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

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

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 yo forget where they belong, you can say



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

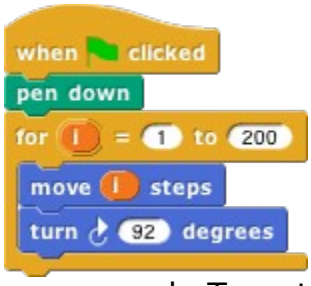
The technical term for a **true** or **false** value is a "Boolean" value; it has a capital B because it' person, George Boole, who developed the mathematical theory of Boolean values. Don't get co hexagonal block is a *predicate,* but the value it reports is a *Boolean.*

Another quibble about vocabulary: Many programming languages reserve the name "procedu Commands (that carry out an action) and use the name "function" for Reporters and Predicates manual, a *procedure* is any computational capability, including those that report values and tho Commands, Reporters, and Predicates are all procedures. The words "a Procedure type" are sh "Command type, Reporter type, or Predicate type."

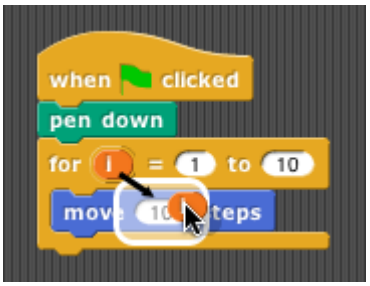
If you want to put a *constant* Boolean value in a hexagonal slot instead of a predicate-based ex the mouse over the block and click on the control that appears:

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



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

The variable **i** is created by the **for** block, and it can only be used in the script inside the block
way, if you don't like the name **i**, you can change it by clicking on the orange oval without dragg
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
because mathematicians tend to use letters between **i** and **n** to represent integer values, but in
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 th
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
want) or to make it visible only in the current sprite. You'd do that if you're going to give severa
individual variables *with the same name,* so that you can share a script between sprites (by dra
current sprite's scripting area to the picture of another sprite in the sprite corral), and the differ
slightly different things when running that script because each has a different value for that var

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
orange oval in the **for** block. You can drag the variable into any script in the scripting area. Ne
a checkbox, initially checked. When it's checked, you'll also see a *variable watcher* on the stag

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:

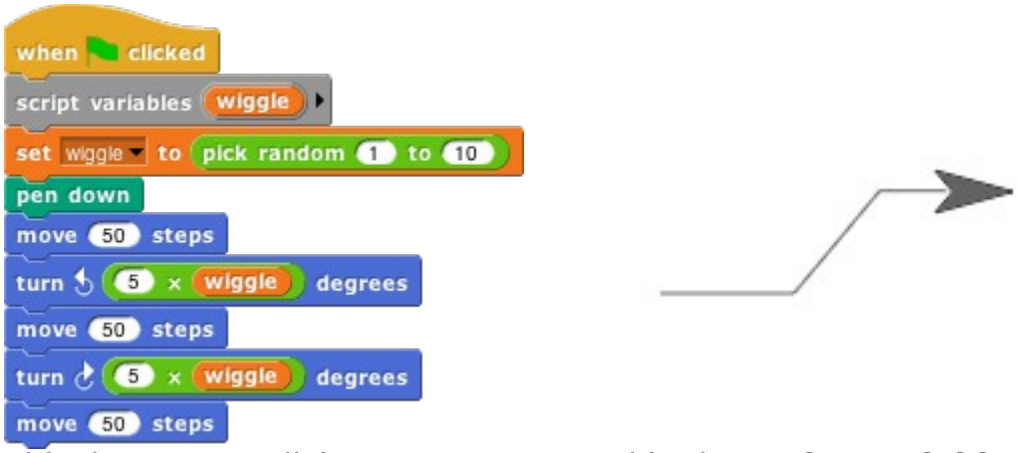
Note that you *don't* drag the variable's oval into the **set** block! You click on the downarrow in t slot, and you get a menu of all the available variable names.

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

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

Script Variables

In the name example above, our project is going to carry on an interaction with the user, and w remember their name throughout the project. That's a good example of a situation in which a *g* (the kind you make with the "**Make a variable**" button) is appropriate. Another common exam called "**score**" in a game project. But sometimes you only need a variable temporarily, during t 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 dragg name. You can also make more than one temporary variable by clicking on the right arrow at th block to add another variable oval:

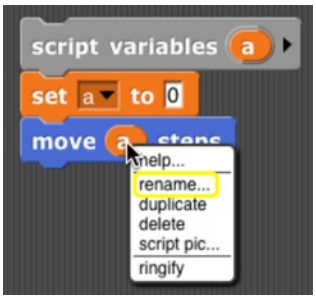
Renaming variables

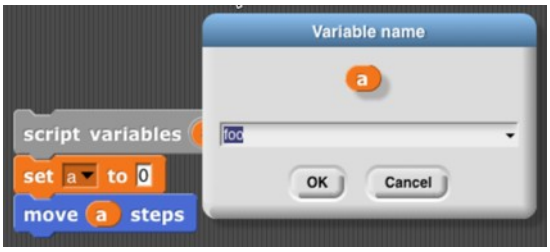
There are several reasons why you might want to change the name of a variable:

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
3. You just decide a different name would be more self-documenting.

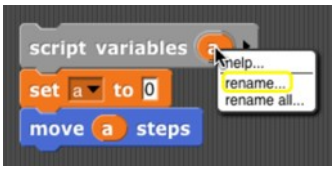
In the first and third case, you probably want to change the name everywhere it appears in that in all scripts. In the second case, if you've already used both variables in the script before realiz have the same name, you'll want to look at each instance separately to decide which ones to re these operations 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 that one orange oval:

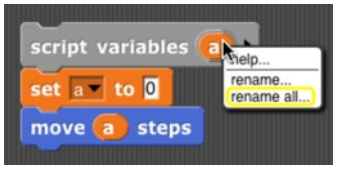




If you right-click on the place where the variable is *defined* (a **script variables** block, the orang variable in the Variables palette, or an orange oval that's built into a block such as the "**i**" in **for** given two renaming options, "rename" and "rename all." If you choose "rename," then the nam 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 vari 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 some there's no limit to the amount of information you can put in a variable; in Chapter IV you'll see h to collect many values in one data structure, and in Chapter VIII you'll see how to read informat sites. When you use these capabilities, your project may take up a lot of memory in the comput close to the amount of memory available to Snap!, then it may become impossible to save you space is needed temporarily to convert from Snap! 's internal representation to the form in whic exported or saved.) If your program reads a lot of data from the outside world that will still be a you use it next, you might want to have values containing a lot of data removed from memory b project. To do this, right-click or control-click on the orange oval in the Variables palette, to see

You already know about the rename options, and **help...** displays a help screen about variables Here we're interested in the check box next to **transient**. If you check it, this variable's value w when you save your project. Of course, you'll have to ensure that when your project is loaded, 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 *paus* a script partway through, so that you can examine the values of variables.

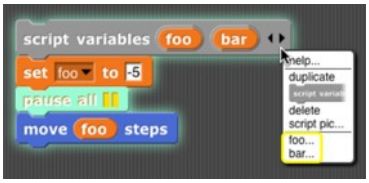
The pause button

The simplest way to pause a program is manually, by clicking t pause button in the top r the window. While the program is paused, you can run other scripts by clicking on them, show stage with the checkbox next to the variable in the Variables palette or with the **show variable** the other things you can generally do, including modifying the paused scripts by adding or remo The button changes sha 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 *breakpoint,* a particular point in a script at which you wan pause all e. The block, ne the Control palette, can be inserted in a script to pause when it is run. So, for example, if your getting an error message in a particular block, you could use **pause all** just before that block to of variables just before the error happens.

The **pause all** block turns bright cyan while paused. Also, during the pause, you can right-clic script and the menu that appears will give you the option to show watchers for temporary varia



But what if the block with the error is run many times in a loop, and it only errors when a parti is true—say, the value of some variable is negative, which shouldn't ever happen. In the iterati page 25 for more about how to use libraries) is a breakpoint block that lets you set a *conditiona* 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 beha continue (with the pause button), the temporary variable watchers are removed by this breakpo resuming the script. The breakpoint block isn't magic; you could alternatively just put a **pause**

---

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

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 cho 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
understand better, you'd like to watch the program as it runs, at human speed rather than at co
You can do this by clicking the *visible stepping button* ( ), before running a script or while th
The button will light ( ) and a speed control slider will appear in the toolbar. W
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
expression involving several reporter blocks, each of those would light up as they are called. No
to a block is evaluated before the block itself is called, so, for example, the **100** lights up before

The speed of stepping is controlled by the slider. If you move the slider all the way to the left,
zero, the pause button turns into a st button , and the script takes a single step each tim
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
parallel. However, consider the case of two **repeat** loops with different numbers of blocks. Wh
each script goes through a complete cycle of its loop in each display cycle, despite the differenc
a cycle. In order to ensure that the visible result of a program on the stage is the same when st
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
those blocks.

G. Etcetera

This manual doesn't explain every block in detail. There are many more motion blocks, sound b
and graphics effects blocks, and so on. You can learn what they all do by experimentation, and
the "help screens" that you can get by right-clicking or control-clicking a block and selecting "**h**
menu that appears. If you forget what palette (color) a block is, but you remember at least part
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
clicking the block in the scripting area gives the option t 

if vector logging is
enabled. See page 116.

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

Takes a sprite as input. Like **stamp** except that the costume is stamped onto th sprite instead of onto the stage. (Does nothing if the current sprite doesn't over chosen sprite.)

Takes a sprite as input. Erases from that sprite's costume the area that overlap current sprite's costume. (Does not affect the costume in the chosen sprite's wa the copy currently visible.)

See page 6.

See pa

Runs only this script until finished. In the Control palette even though it's gray.

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

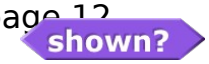
Looping block like **repeat** but with an index variable.

Declare local variables in a script.

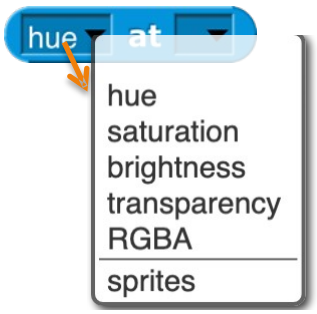
See page 91.

reports the value of a graphics effect.

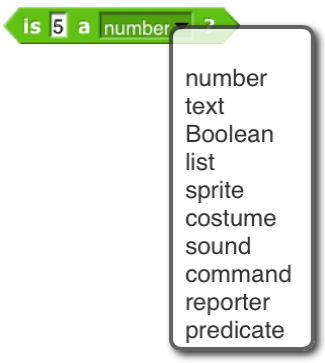
Constant **true** or **false** value. See page 12.



Create a primitive using JavaScript. (This block is disabled by default must check "Javascript extensions" in the setting menu *each time* a project is loaded.)

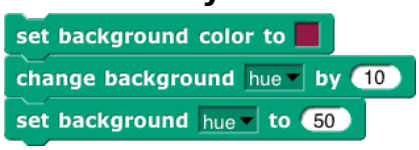


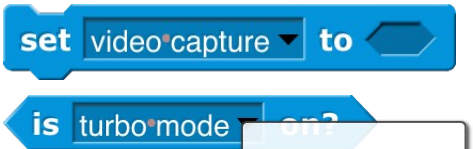
The **at** block lets you examine the screen pixel directly behind the rotatio a sprite, the mouse, or an arbitrary (x,y) coordinate pair dropped onto the menu slot. The first five items of the left menu let you examine the color the position. (The "**RGBA**" option reports a list.) The "**sprites**" option re of all sprites, including this one, any point of which overlaps this sprite's r center (behind or in front). This is a hyperblock with respect to its second



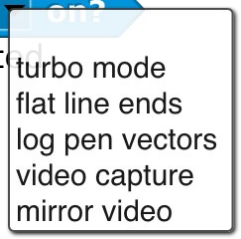
Checks the data type of a value.

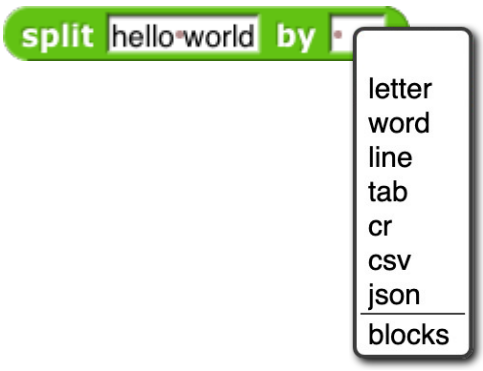
Blocks only for the Stage:





Get or set selected global flags.





Turn the text into a list, using the second input as the delimiter b items. The default delimiter, indicated by the brown dot in the in slot, is a single space character. "**Letter**" puts each character o in its own list item. "**Word**" puts each word in an item. (Words a separated by any number of consecutive space, tab, carriage ret newline characters.) "**Line**" is a newline character (0xa); "**tab**" character (0x9); "**cr**" is a carriage return (0xd). "**Csv**" and "**jso** formatted text into lists of lists; see page 54. "**Blocks**" takes a s the first input, reporting a list structure representing the structur script. See Chapter XI.

For lists, reports **true** only if its two input values are the very same list, so c item in one of them is visible in the other. (For **=**, lists that look the same are the same.) For te case-sensitive comparison, unlike **=**, which is case-independent.

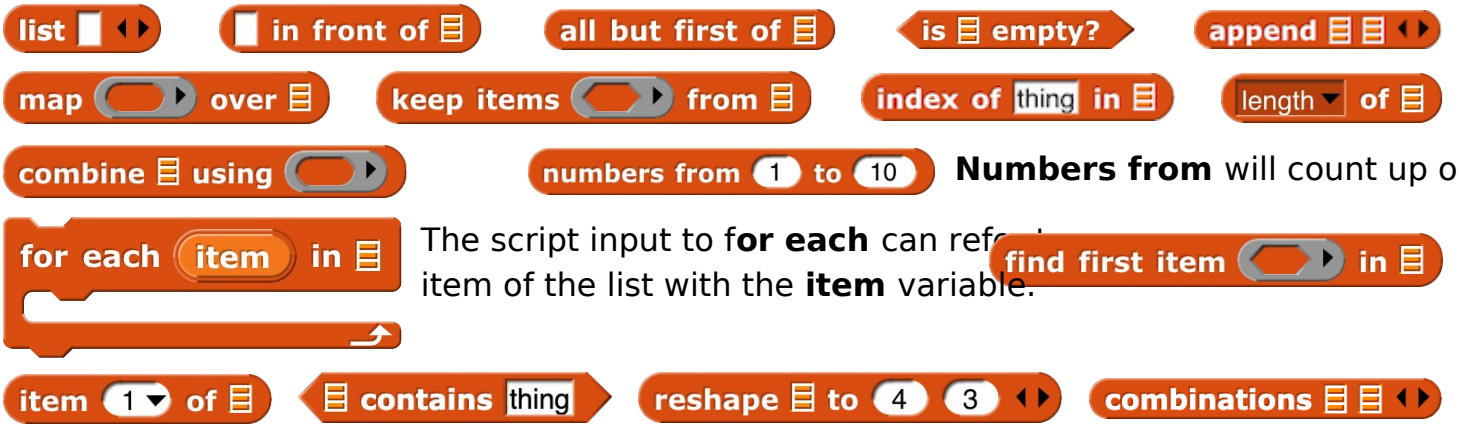
These *hidden* blocks can be found with the **relabel** option o dyadic arithmetic block. They're hidden partly because writing them in Snap! is a good, pretty programming exercise. Note: the two inputs to **atan2** are Δ*x* and Δ*y* in that order, because we clockwise from north. **Max** and **min** are *variadic;* by clicking the arrowhead, you can provide a

Similarly, these hidden predicates can be found by relabeling the rela predicates.

Metaprogramming (see Chapter XI. , page 101)

These blocks support *metaprogramming,* which means manipulating blocks and scripts as data. same as manipulating procedures (see Chapter VI.), which are what the blocks *mean;* in metap actual blocks, what you see on the screen, are the data. This capability is new in version 8.0.

First class list blocks (see Chapter IV, page 46):



Numbers from will count up or dow

The script input to f**or each** can refe item of the list with the **item** variable.

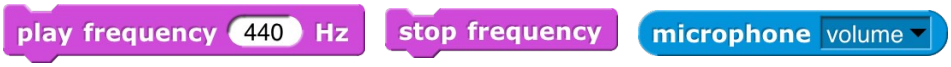
report the sprite or mouse position as a two-item vector (x,y).

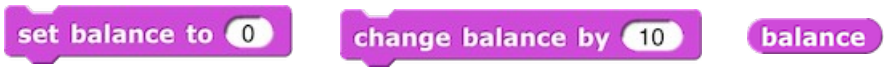
First class procedure blocks (see Chapter VI, page 65):

First class continuation blocks (see Chapter X, page 93):

First class sprite, costume, and sound blocks (see Chapter VII, page 73):

Object is a hyperblock.





Scenes:

The major new feature of version 7.0 is *scenes:* A project can inclu
it sub-projects, called scenes, each with its own stage, sprites, scr
so on. This block makes another scene active, replacing the curre

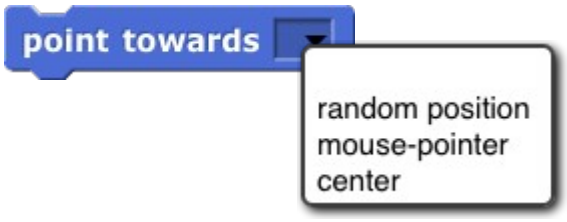
Nothing is automatically shared between scenes: no sprites, no bl
variables. But the old scene can send a message to the new one,
running, with optional payload as in **broadcast** (page 23).

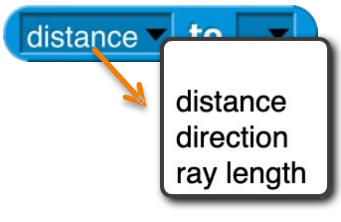
In particular, you can say

if the new scene expects to be started with a green flag signal.

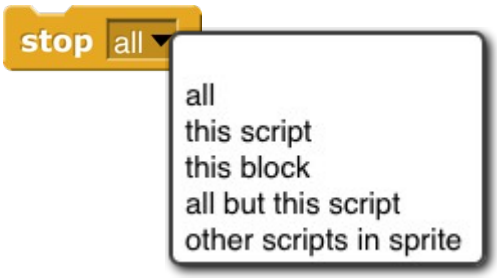
These aren't new blocks but they have a new feature:

These accept two-item (x,y) lists as input, and have extended menus (also including other sprite

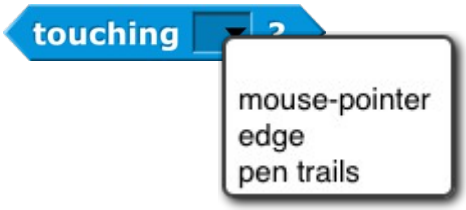




"**Center**" means the center of the stage, the point at (0,0). "**Direction**" is in the **point in dire** direction that would leave this sprite pointing toward another sprite, the mouse, or the center. the distance from the center of this sprite to the nearest point on the other sprite, in the current

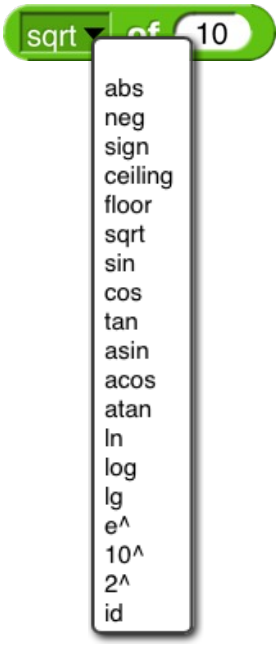


The **stop** block has two extra menu choices. **Stop this block** inside the definition of a custom block to stop just this invocati this custom block and continue the script that called it. **Stop a this script** is good at the end of a game to stop all the game p from moving around, but keep running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



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

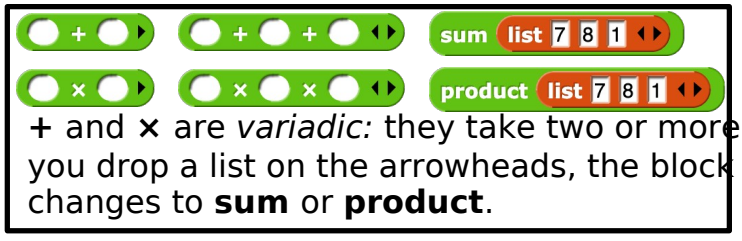
The **video** block has a **snap** option that takes a snapshot and rep costume. It is hyperized with respect to its second input.



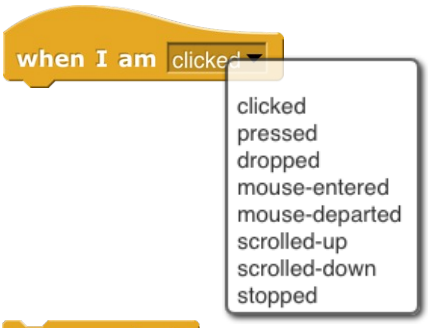
The "**neg**" option is a monadic negation operator, e

"**lg**" is log2; "**id**" is the identity function, which reports its input. "**sign**" reports positive input, 0 for zero input, or -1 for negative input.

name changed to clar



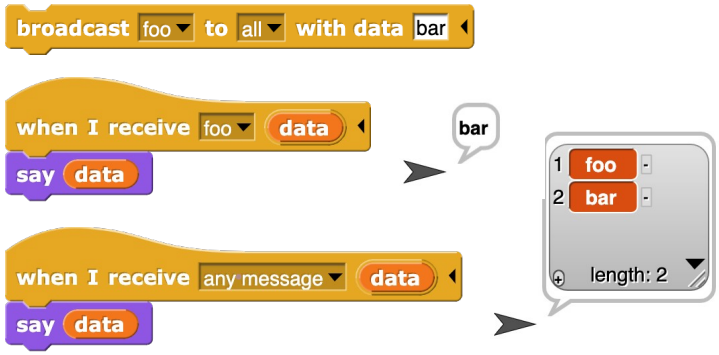
+ and **×** are *variadic:* they take two or more input you drop a list on the arrowheads, the block name changes to **sum** or **product**.



Extended mouse interaction events, sensing clicking, dragging, hov
The "**stopped**" option triggers when all scripts are stopp, as with
stop button; it is useful for robots whose hardware interface must b
turn off motors. A **when I am stopped** script can run only for a lin

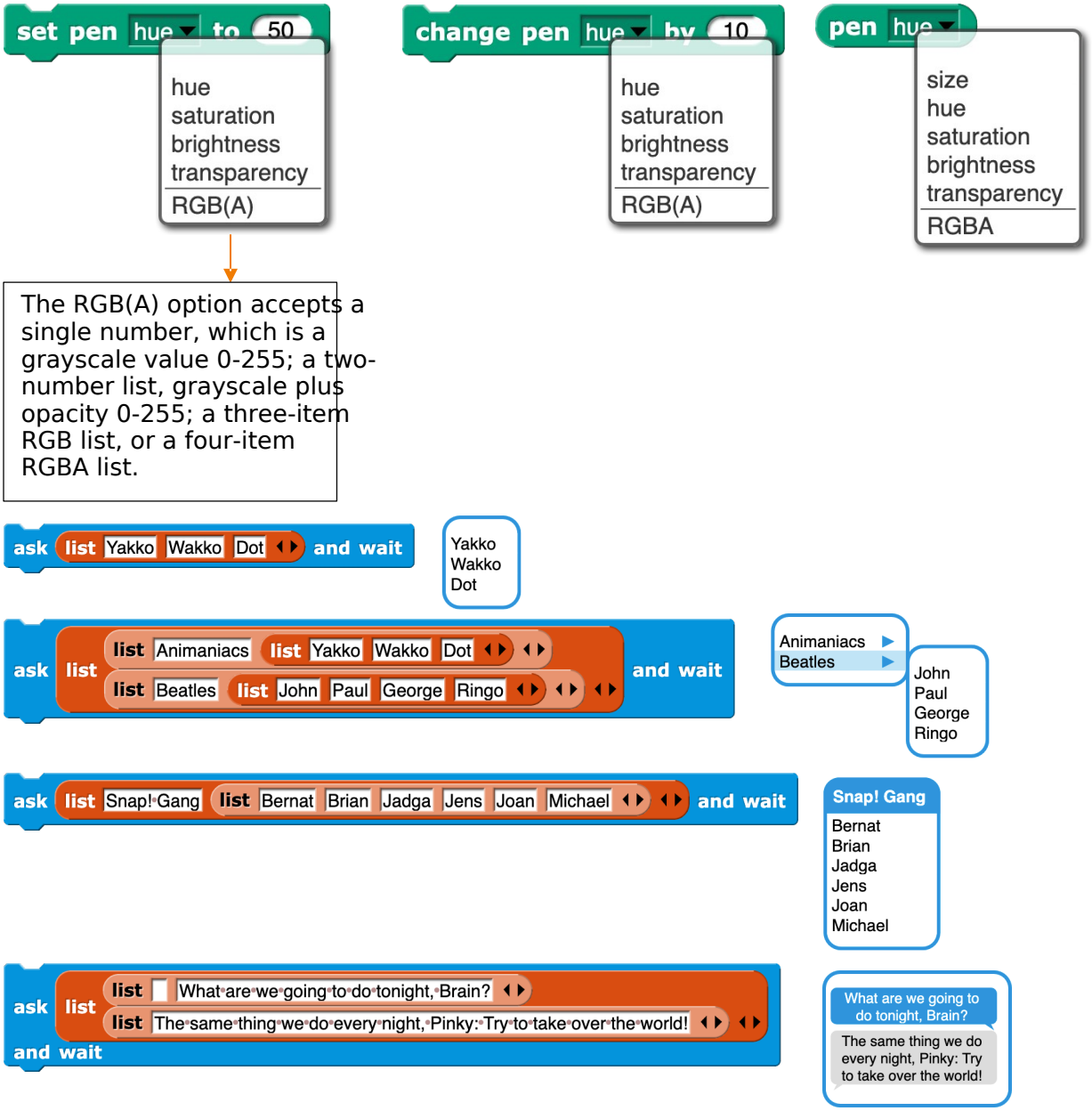
Extended **broadcast**: Click the right arrowhead to direct th
message to a single sprite or the stage. Click again to add a
as a payload to the message.

Extended **when I receive**: Click the right arrowhead to expose a script v
on it to change its name, like any script variable) that will be set to the d
matching **broadcast**. If the first input is set to "**any message**," then th
variable will be set to the message, if no payload is included with the bro
to a two-item list containing the message and the payload.



If the input is set to "any key," then a right arrowhead appears:

and if you click it, a script variable **key** is created whose value is the key th
pressed. (If the key is one that' represented in the input menu by a word o
e.g., "enter" or "up arrow," then the value of **key** will be that word or phra
the space character, which is represented as itself in **key**.)



These **ask** features and more in the **Menus** library.

The **of** block has an extended menu of attributes of a sprite. **Position** repo vector. **Size** reports the percentage of normal size, as controlled by the **set** the **Looks** category. **Left**, **right**, etc. report the stage coordinates of the cor edge of the sprite's bounding box. **Variables** reports a list of the names of a scope (global, sprite-local, and script variables if the right input is a script.

H. Libraries

There are several collections of useful procedures that aren't Snap! primitives, but are provide include a library in your project, choose the **Libraries...** op in the file ( ) menu.

The library menu is divided into five broad categories. The fir is, broadly, utilities: blocks that might well be primitives. The might be useful in all kinds of projects.

The second category is blocks related to media computation: ones that help in dealing with costumes and sounds (a/k/a Je libraries). There is some overlap with "big data" libraries, for dealing with large lists of lists.

The third category is, roughly, specific to non-media applicati (a/k/a Brian libraries). Three of them are imports from other programming languages: words and sentences from Logo, arr functions from APL, and streams from Scheme. Most of the others are to meet the needs of the BJC curriculum.

The fourth category is major packages (extensions) provided users.

The fifth category provides support for hardware devices such robots, through general interfaces, replacing specific hardwar libraries in versions before 7.0.

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

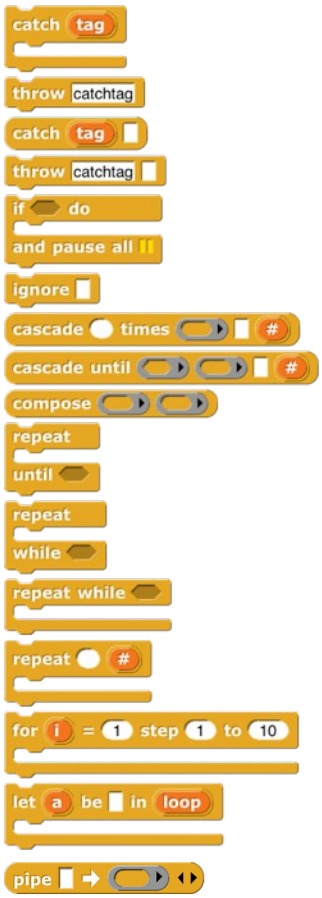
The libraries and their contents may change, but as of this writing the **list library** has these blo

(The lightning bolt before the name in several of these blocks means th compiled HOFs or JavaScript primitives to achieve optimal speed. They officially considered experimental.) **Remove duplicates from** reports no two items are equal. The **sort** block takes a list and a two-input com predicate, such as **<**, and reports a list with the items sorted according t comparison. The **assoc** block is for looking up a key in an *association li* two-item lists. In each two-item list, the first is a *key* and the second is inputs are a key and an association list; the block reports the first key-va whose key is equal to the input key.

For each item is a variant of the primitive version that provides a **#** va containing the position in the input list of the currently considered item. is a version of **map** that allows multiple list inputs, in which case the ma function must take as many inputs as there are lists; it will be called wit first items, all the second items, and so on. **Zip** takes any number of lists as inputs; it reports a first items, all the second items, and so on. The no-name identity function reports its input.

Sentence and **sentence→list** are borrowed from the word and sentence library (page 27) to s **append** that accepts non-lists as inputs. **Printable** takes a list structure of any depth as input compact representation of the list as a text string.

The **iteration, composition library** has these blocks:



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

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

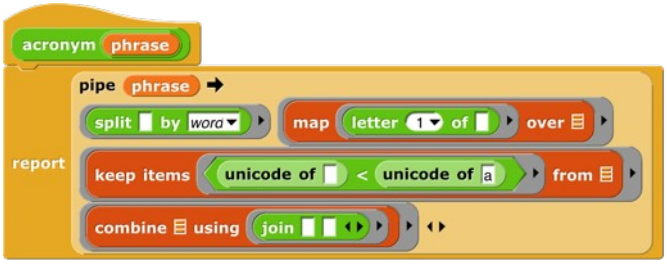
Ignore is used when you need to call a reporter but you don't care about reports. (For example, you are writing a script to time how long the repor

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

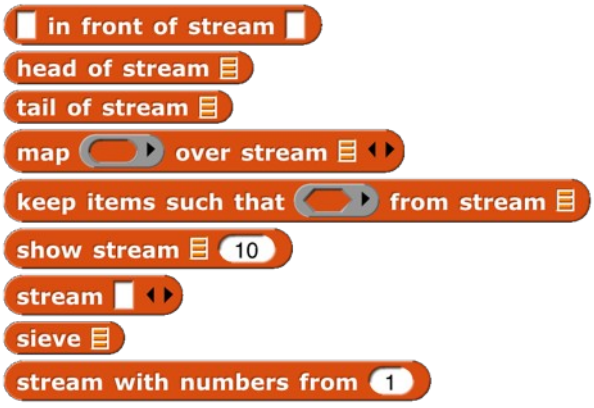
The **compose** block takes two functions and reports the function $f(g(x))$.

The first three **repeat** blocks are variants of the primitive **repeat until** bl all four combinations of whether the first test happens before or after the repetition, and whether the condition must be true or false to continue rep The last **repeat** block is like the **repeat** primitive, but makes the number repetitions so far available to the repeated script. The next two blocks are on **for**: the first allows an explicit step instead of using ±1, and the secon 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 allows reordering a nested composition with a left-to-right one:

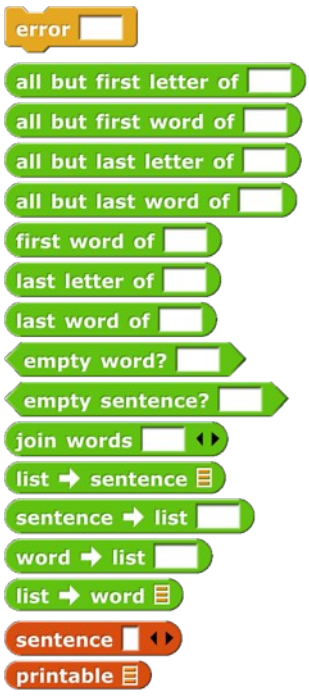


The **stream library** has these blocks:



Streams are a special kind of list whose items are not co until they are needed. This makes certain computations efficient, and also allows the creation of lists with infinite many items, such as a list of all the positive integers. T five blocks are stream versions of the list blocks **in fron item 1 of**, **all but first of**, **map**, and **keep**. **Show stre** a stream and a number as inputs, and reports an ordina of the first *n* items of the stream. **Stream** is like the pri **list**; it makes a finite stream from explicit items. **Sieve** example block that takes as input the stream of integer starting with 2 and reports the stream of all the prime numbers. **Stream with numbers from 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: isn't best viewed as a string of characters, but rather as a *sentence*, made of which is a string of *letters.* With a few specialized exceptions, this is wh text into computers: The text is sentences of natural (i.e., human) langua emphasis is on words as constitutive of sentences. You barely notice the words, and you don't notice the spaces between them at all, unless reading. (Even then: Proofreading is *diffciult,* because you see what you see, what will make the snetence make sense, rather than the misspelling of your eyes.) Internally, Logo stores a sentence as a list of words, and a string of letters.

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

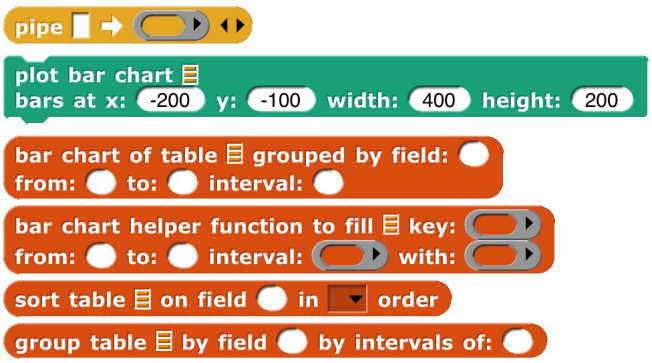
In Logo, the visual representation of a sentence (a list of words) looks like language sentence: a string of words with spaces between them. In Snap representation of a list looks nothing at all like natural language. On the other hand, representi a string means that the program must continually re-parse the text on every operation, looking treating multiple consecutive spaces as one, and so on. Also, it's more convenient to treat a se words rather than a string of words because in the former case you can use the higher order fun **keep**, and **combine** on them. This library attempts to be agnostic as to the internal representa The sentence selectors accept any combination of lists and strings; there are two sentence cons 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 bl you don't have to type the block name, instead of the terse **butfirst** or the cryptic **bf** we spell o and include "word" or "sentence" to indicate the intended domain. There's no **first letter of** b **letter 1 of** serves that need. **Join words** (the sentence-as-string constructor) is like the primit it puts a space in the reported value between each of the inputs. **Sentence** (the List-colored se constructor) accepts any number of inputs, which can be words, sentences-as-lists, or sentence inputs are lists of lists, only one level of flattening is done.) **Sentence** reports a list of words; th empty words or words containing spaces. The four blocks with right-arrows in their names conv forth between text strings (words or sentences) and lists. (Splitting a word into a list of letters i you're a linguist investigating orthography.) **Printable** takes a list (including a deep list) of wor reports a text string in which parentheses are used to show the structure, as in Lisp/Scheme.

The **pixels library** has one block:

Costumes are first class data in Snap!. Most of the processing of costume data is don blocks in the Looks category. (See page 79.) This library provides **snap**, which takes using your computer's camera and reports it as a costume.

The **bar charts library** has these blocks:



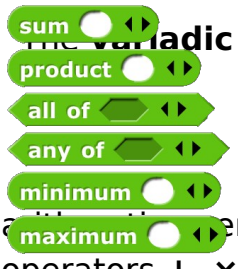
Bar chart takes a table (typically from a CSV data se input and reports a summary of the table grouped by field in the specified column number. The remaining three inputs are used only if the field values are num in which case they can be grouped into buckets (e.g. decades, centuries, etc.). Those inputs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

Bar chart reports a new table with three columns. The first column contains the bucket name number. The second column contains a nonnegative integer that says how many records in the into this bucket. The third column is a subtable containing the actual records from the original t into the bucket. **Plot bar chart** takes the table reported by **bar chart** and graphs it on the sta 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 on "Frequency Distribution Analysis" library instead.

The **multi-branched conditional library** has these blocks:

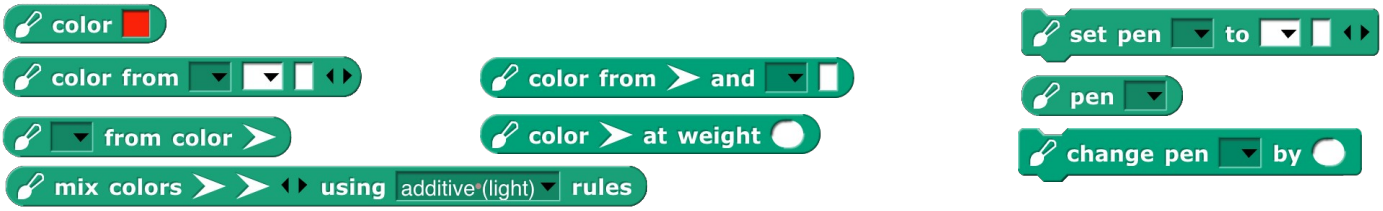
The **catch** and **throw** blocks duplicate ones in the iteration library, and included because they are used to implement the others. The **cases** bl a multi-branch conditional, similar to **cond** in Lisp or **switch** in C-family languages. The first branch is built into the **cases** block; it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



The **variadic library** has these blocks:

These are versions of the associative operators **and**, and **or** that take any numb instead of exactly two inputs. As with any variadic input, you can also drop a lis onto the arrowheads instead of providing the inputs one at a time As of version erators **sum**, **product**, **minimum**, and **maximum** are no longer included, becaus operators **+**. **×**, **min**, and **max** are themselves variadic.

The **colors and crayons library** has these blocks:

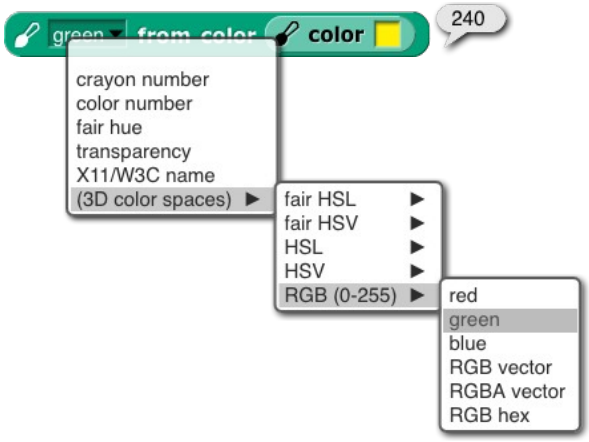


It is intended as a more powerful replacement for the primitive **set pen** block, including *first cla* HSL color specification as a better alternative to the HSV that Snap! inherits from JavaScript; a ' that compensates for the eye's grouping a wide range of light frequencies as green while labelli orange or yellow; the X11/W3C standard color names; RGB in hexadecimal; a linear color scale days, but better) based on fair hues and including shades (darker colors) and grayscale. Anothe curated set of 100 "crayons," explained further on the next page.

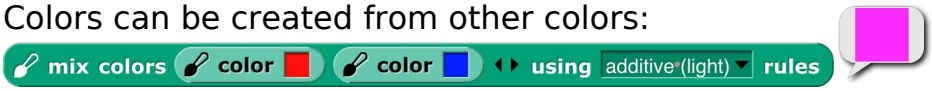
Colors are created by 

block (for direct user selection), the **color from** block to speci numerically, or 

, which reports the color currently in use by the pen. The **from col** reports names or numbers associated with a color:

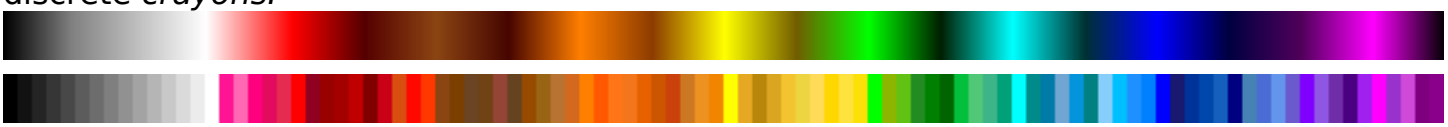


Colors can be created from other colors:



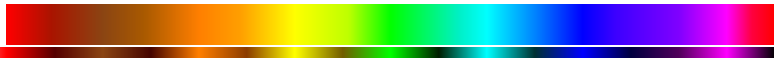
The three blocks with **pen** in their names are improved versions of primitive Pen blocks. In pri for example, could be implemented using a (hypothetical) **set pen to color** composed with the but in fact **set pen** benefits from knowing how the pen color was set in its previous invocation, implemented separately from **color from**. Details in Appendix A.

The recommended way to choose a color is from one of two linear scales: the continuous *color* discrete *crayons:*



Color numbers are based on *fair hues,* a modification of the spectrum (rainbow) hue scale that to green and more to orange and yellow, as well as promoting brown to a real color. Here is the scale, for reference:

Here is the fair hue scale:

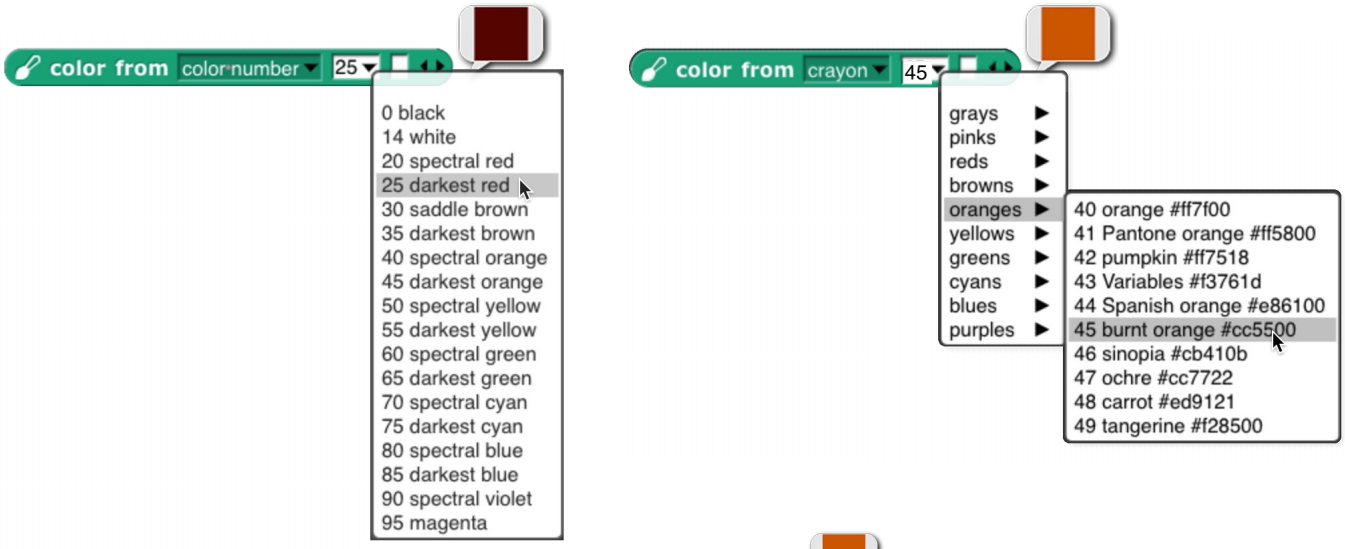


Here is the color number scale

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

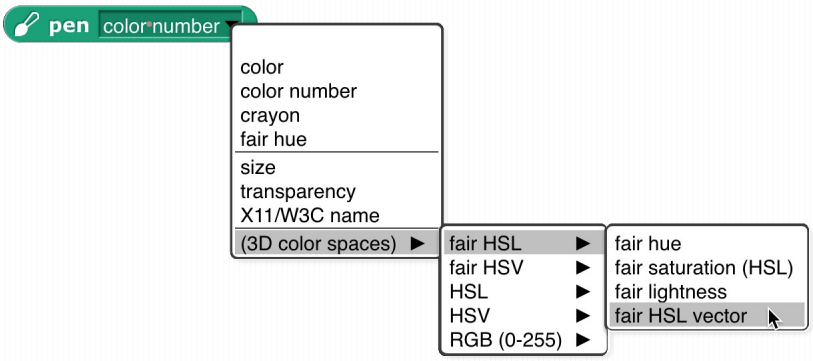
And here are the 100 crayons

The **color from** block, for example, provides different pulldown menus depending on which sca



You can also type the crayon name: 

There are many scales:



The white slot at the end of some of the blocks has two purposes. It can be used to add a tra color (0=opaque, 100=transparent):

or it can be expanded to enter three or four numbers for a vector directly into the block, so thes

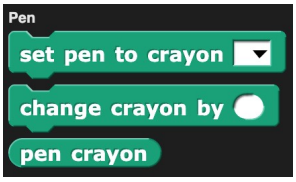
But note that a transparency number in a four-number RGBA vector is on the scale 255=opaque 0=transparent, so the following are *not* equivalent:

Set pen crayon to provides the equivalent of a box of 100 crayons. They are divided into colo menu in the **set pen crayon to** input slot has submenus. The colors are chosen so that startin **change pen crayon by 10** rotates through an interesting, basic set of ten colors:

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

(Why didn't we use the colors of the 100-crayon Crayola™ box? A few reasons, one of which is Crayola colors aren't representable on RGB screens. Some year when you have nothing else to "color space" on Wikipedia. Also "crayon." Oh, it's deliberate that **change pen crayon by 5** c white, since that's the usual stage background color. White is crayon 14.) Note that crayon 43 all the standard block colors are included.

See Appendix A (page 139) for more information.



The **crayon library** has only the crayon features, without the rest of the col package.

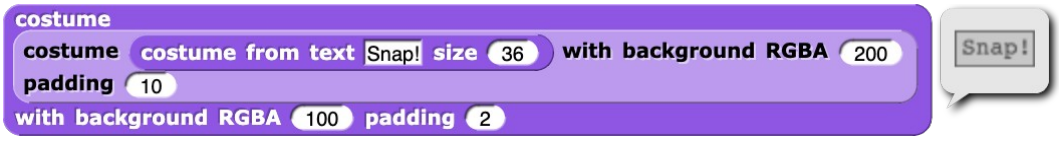
The **catch errors library** has these blocks:

The **safely try** block allows you to handle errors that hap 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 happens, the code in the second C-slot is run. While that second script is running, the v error ble contains the text of the error message that would have been displayed if you weren't catching the error. The **error** the opposite: it lets your program *generate* an error message, which will be displayed with a re caught by **safely try**. **Safely try reporting** is the reporter version of **safely try**.

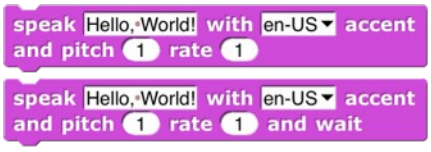
The **text costumes library** has only two blocks:

Costume from text reports a costume that can be used with the **swit costume** block to make a l Snap!:

Costume with background reports a costume made from another costume by coloring its ba color input like the **set pen color to RGB(A)** block and a number of turtle steps of padding ar costume. These two blocks work together to make even better buttons:



The **text to speech library** has these blocks:



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

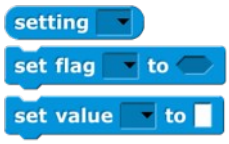
The **parallelization library** contains these blocks:

The two **do in parallel** blocks take any number of scripts as inputs. Those scrip in parallel, like ordinary independent scripts in the scripting area. The **and wai** waits until all of those scripts have finished before continuing the script below th

The **create variables library** has these blocks:

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

The **getters and setters library** has these blocks:



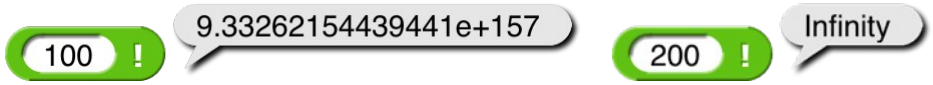
The purpose of this library is to allow program access to the settings co user interface elements, such as th ettings menu. The **setting** a setting; the **set flag** block sets yes-or-no options that have checkbox interface, while the **set value** block controls settings with numeric or te such as project name.

Certain settings are ordinarily remembered on a per-user basis, such as the “zoom blocks” valu these settings are changed by this library, the change is in effect only while the project using th loaded. No permanent changes are made. Note: this library has not been converted for versio have to enable Javascript extensions to use it.

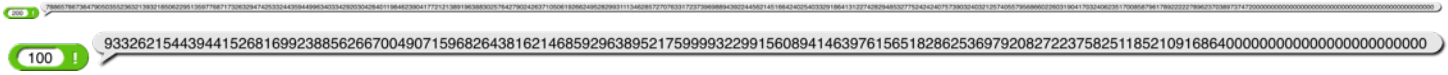
The **bignums, rationals, complex #s library** has these blocks:

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

The **!** block computes the factorial function, useful to test whether bignums are turned on. With

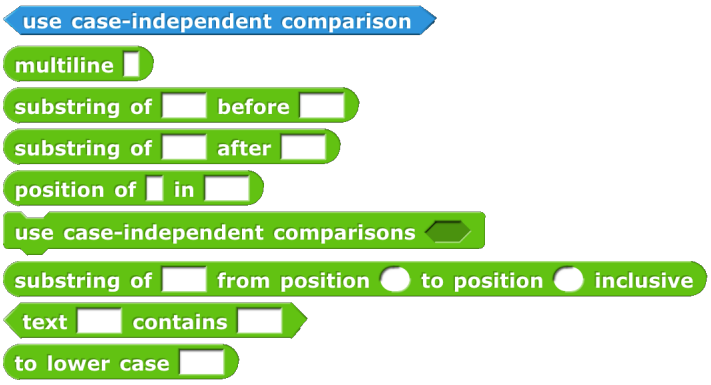


With bignums:



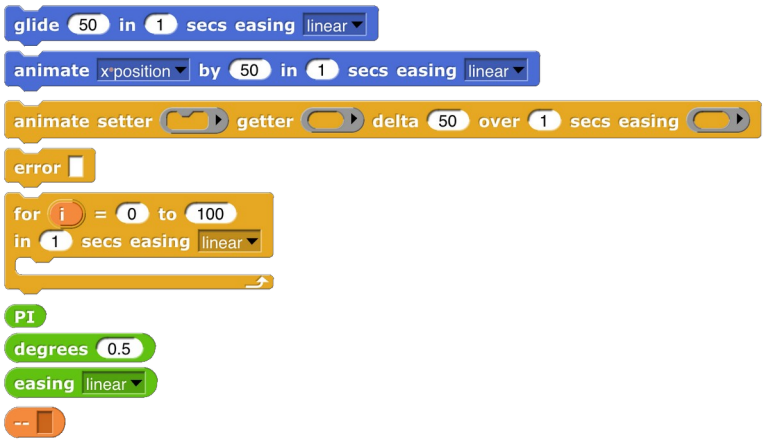
The 375-digit value of 200! isn’t readable on this page, but if you right-click on the block and ch pic,” you can open the resulting picture in a browser window and scroll through it. (These value bunch of zero digits. That’s not roundoff error; the prime factors of 100! and 200! include many 5.) The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

The **strings, multi-line input library** provides these blocks:



All of these could be written in Snap! itself, but are implemented using the corresponding JavaScript library functions directly, so they run fast. They can be used, for example, in scrapin data from a web site. The command **use case independent comparisons** applies only to th library. The **multiline** block accepts and repor text input that can include newline characters.

The **animation library** has these blocks:



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

The central idea of this library is an *easing fur* a reporter whose domain and range are real numbers between 0 and 1 inclusive. The func represents what fraction of the "distance" (in quotes because it might be any numeric value such as temperature in a simulation of weathe 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 it's a requirement that $f(0)=0$ and $f(1)=1$, there is only one linear easing function, $f(x)=x$, and s categories.) T 

block reports some of the common easing functions.

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

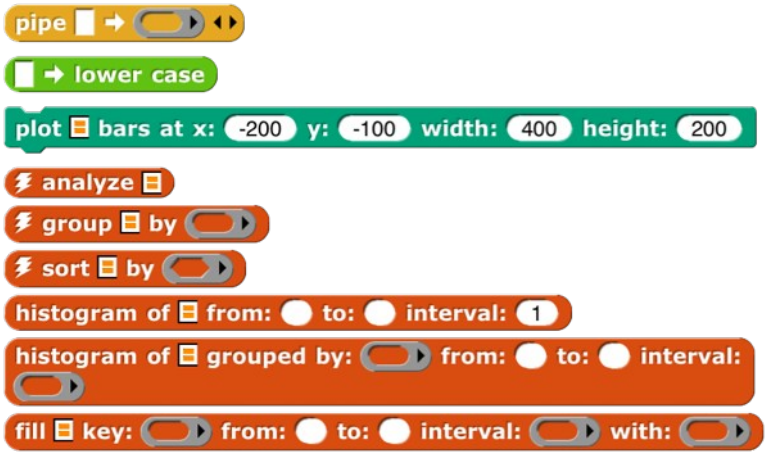
is equivalent to

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

The **serial ports library** contains these blocks:

It is used to allow hardware developers to control devices such as robots t connected to your computer via a serial port.

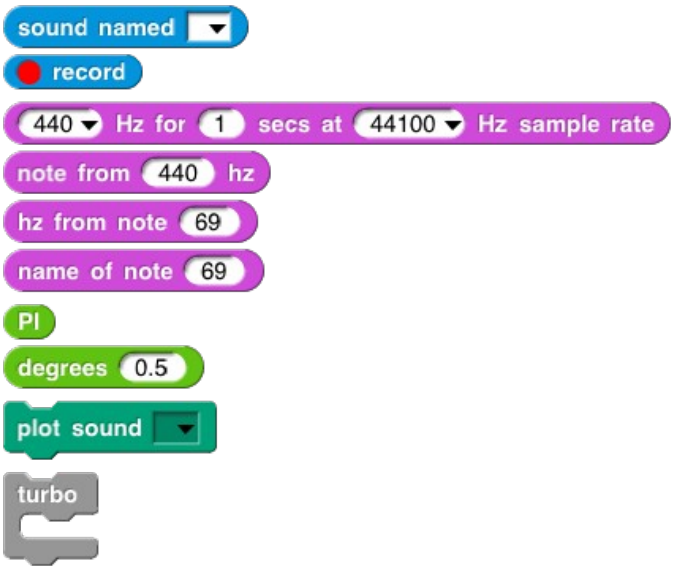
The **frequency distribution analysis library** has these blocks:



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.

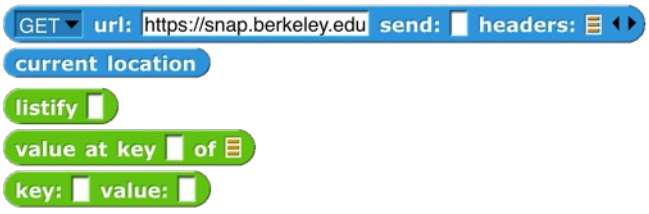
For more information go here:
https://tinyurl.com/jens-data

The **audio comp library** includes these blocks:



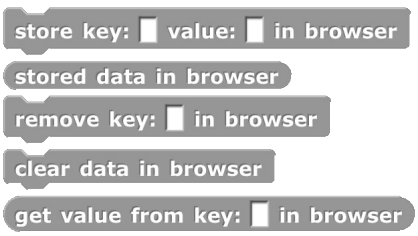
This library takes a sound, one that you record or or from our collection of sounds, and manipulates it by systematically changing the intensity of the samples the sound and by changing the sampling rate at whi the sound is reproduced. Many of the blocks are he for the **plot sound** block, used to plot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

The **web services library** has these blocks:



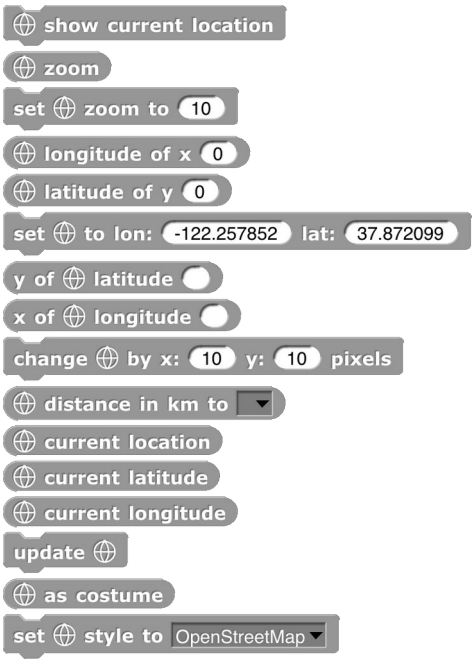
The first block is a generalization of the primitive **url** allowing more control over the various options in web requests: GET, POST, PUT, and DELETE, and fine control over the content of the message sent to the s **Current location** reports your latitude and longitude **Listify** takes some text in JSON format (see page 54) converts it to a structured list. **Value at key** looks up a key-value pair in a (listified) JSON dicti **key:value:** block is just a constructor for an abstract data type used with the other blocks

The **database library** contains these blocks:



It is used to keep data that persist from one Snap! session to the n use the same browser and the same login.

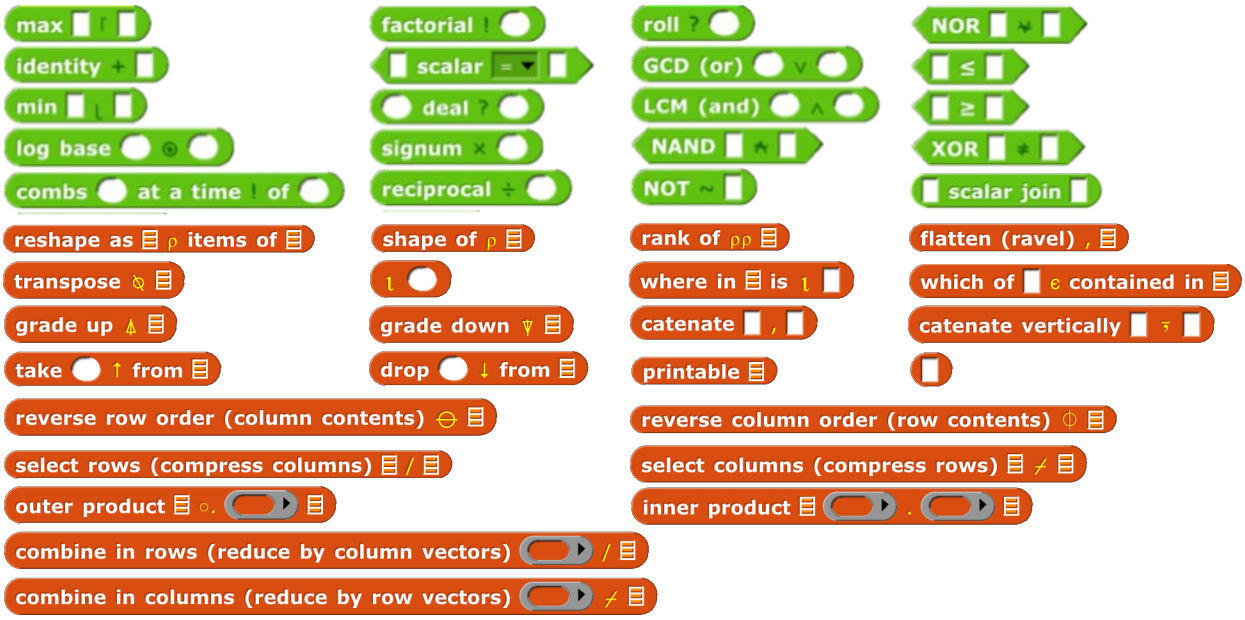
The **world map library** has these blocks:



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

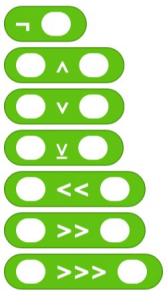
The next five blocks convert between stage coordinates (pixels) a coordinates (latitude and longitude). The **change by x: y:** block map relative to the stage. The **distance to** block measures the distance (in meters) between two sprites. The three reporters wi **current** in their names find *your*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

The **APL primitives library** contains these blocks:

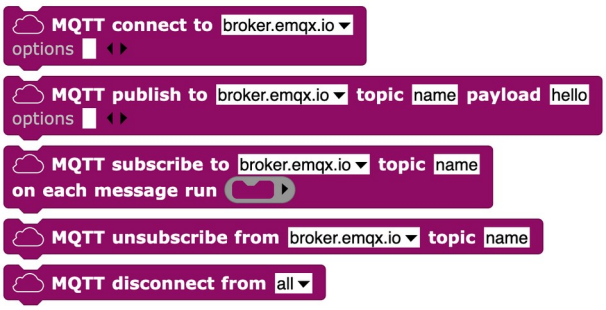


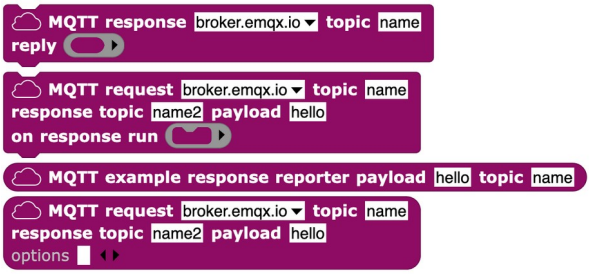
For more information about APL, see Appendix B (page 148).

The **list comprehension library** has one block, **zip**. Its first input is a function inputs. The two Any-type inputs are deep lists (lists of lists of…) interpreted as t function is called with every possible combination of a leaf node of the first tree node of the second tree. But instead of taking atoms (non-lists) as the leaves, **zip** allows the le be vectors (one-dimensional lists), matrices (two-dimensional lists), etc. The Number-type inpu dimension for each tree, so the function input might be called with a vector from the first tree a the second tree.

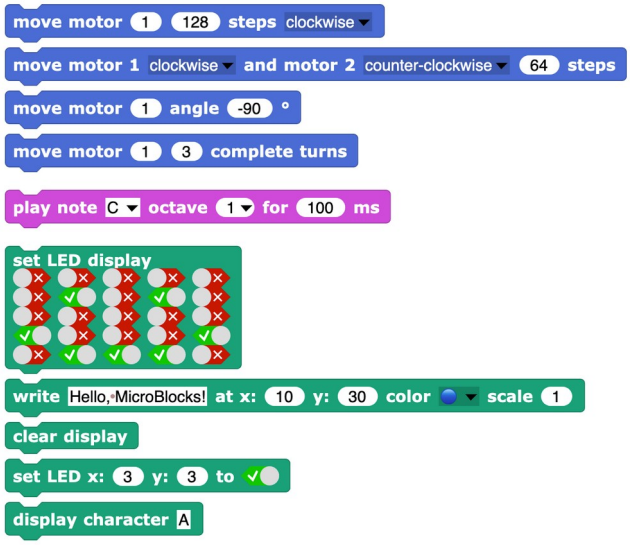


The **bitwise library** provides bitwise logic functions; each bit of the reported value of applying the corresponding Boolean function to the corresponding bits of the inpu Boolean functions are **not** for ¬, **and** for ∧, **or** for ∨, **xor** (exclusive or) for ⊻. remaining functions shift their first input left or right by the number second input. **<<** is left shift, **>>** is arithmetic right shift (shifting in one and **>>>** is logical right shift (shifting in zero bits from the left). what these mean, find a tutorial online.

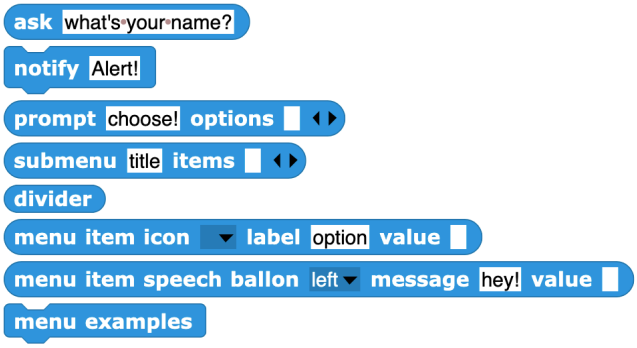




The **MQTT library** supports the Message Queuing Telemetry Transport protocol, devices. See https://mqtt.org/ for more information.



The **Signada library** allows you t control a microBit or device that works with the Signada MicroBlocks project.



The **menus library** provides the ability to disp hierarchical menus on the stage, using the ask b ability to take lists as inputs. See p

The **SciSnap !** library, **TuneScope** the library are too big to discuss here and are do separately at http://emu-online.de/ProgrammingWithSciSnap.pdf and https://maketolearn.org/creating-art-animations-and-music/ respectively.

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
There are two ways to do that. You can save a project on your own computer, or you can save i
web site. The advantage of saving on the net is that you have access to your project even if yo
different computer, or a mobile device such as a tablet or smartphone. The advantage of savin
computer is that you have access to the saved project while on an airplane or otherwise not on
cloud projects are limited in size, but you can have all the costumes and sounds you like if you s
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 bottom right of the Save dialog allows you to enter project notes that are saved with the pro

A. Local Storage

Click on **Computer** and Snap!'s Save Project dialog window will be replaced by your operating s
save window. If your project has a name, that name will be the default filename if you don't giv
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! we r to
do this, you need an account with us. Click on the Cl d button ( ) in
Choose the "Signup..." option. This will show you a window that looks like
the right.

You must choose a user name that will identify you on the web site, such . If
you're a Scratch user, you can use your Scratch name for Snap! too. If you 't
pick a user name that includes your family name, but first names or initials n'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 email address or your parent's email address. (If you're a kid, you shouldn't sign up for anythin even Snap!, without your parent's knowledge.) We do not store your birthdate information on o used on your own computer only during this initial signup. We do not ask for your *exact* birthda 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 (b link) that it's really your email address. We keep your email address on file so that, if you forge we can send you a password-reset link. We will also email you if your account is suspended for Terms of Service. We do not use your address for any other purpose. You will never receive ma of any kind through this site, neither from us nor from third parties. If, nevertheless, you are wo 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 w else's use of the web site, and don't put copyrighted media or personally identifiable informatio you share with other users. And we're not responsible if something goes wrong. (Not that we e go wrong; since Snap! runs in JavaScript in your browser, it is strongly isolated from the rest of 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 Clou

Use the user name and password that you set up earlier. If you check the "Stay signed in" box, logged in automatically the next time you run Snap! from the same browser on the same compu box if you're using your own computer and you don't share it with siblings. *Don't* check the box 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 enter a project name, and optionally project notes; your project will be saved online and can be anywhere with net access. The project notes will be visible to other users if you publish your pr

D. Loading Saved Projects

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

1. If you saved the project in your online Snap! account, choose the "Open..." option from the F Choose the "Cloud" button, then select your project from the list in the big text box and click OK the "Computer" button to open an operating system open dialog. (A third button, "Examples,"

from example projects that we provide. You can see what each of these projects is about by cli
reading its project notes.)

2. If you saved the project as an XML file on your computer, choose “Import...” from the File me
give you an ordinary browser file-open window, in which you can navigate to the file as you wou
software. Alternatively, find the XML file on your desktop, and just drag it onto the Snap! windo

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

Snap! can also import projects created in BYOB 3.0 or 3.1, or (with some effort; see our web si
2.0 or 3.0. Almost all such projects work correctly in Snap!, apart from a small number of incom
If you saved projects in an earlier version of Snap! using the “Browser” option, then a **Browser** button will be sh
dialog to allow you to retrieve those projects. But you can save them only with the **Computer** and **Cloud** options

E. If you lose your project, do this first!

If you are still in Snap! and realize that you’ve loaded another project without saving the one
working on: ***Don’t edit the new project.*** From th[illegible] choose the **Restore unsaved project**
option.

Restore unsaved project will also work if you log out of Snap! and later log back in, as long
another project meanwhile. Snap! remembers only the most recent project that you’ve edited (
but actually changed in the project editor).

If your project on the cloud is missing, empty, or otherwise broken and isn’t the one y
most recently, or if **Restore unsaved project** fails: ***Don’t edit the broken project.*** In the **C**
your project name, then push the **Recover** button. *Do this right away,* because we save only th
most recent, and the latest before today. So don’t keep saving bad versions; **Recover** right aw
feature works only on a project version that you actually saved, so **Restore unsaved project**
you switch away from a project without saving it.

To help you remember to save your projects, when you’ve edited the project and haven’t yet s
displays a pencil icon to the left of the project name on the toolbar at the top of the window:

F. 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
available to others. If you **share** a project, you can give your friends a project URL (in your brow
after you open the project) they can use to read it. If you **publish** a project, it will appear on th
and the whole world can see it. In any case, nobody other than you can ever overwrite your pro
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 most important capability we added to Scratch. (The name was changed because a few teache of humor. ☹ You pick your battles.) Scratch 2.0 and later also has a partial custom block capabi

A. Simple Blocks

In every palette, at or near the bottom, is a button labeled "**Make a block**." Also, floating nea palette is a plus sign. Also, the menu you get by right-clicking on the background of the scriptin "make a block" option.

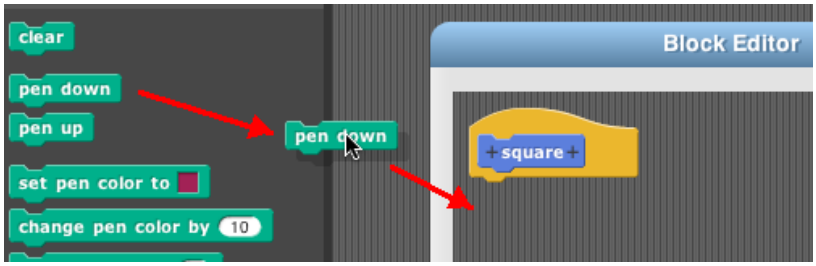
Clicking any of these will display a dialog window in which you choose the block's name, shape, palette/color. You also decide whether the block will be available to all sprites, or only to the cu its children.

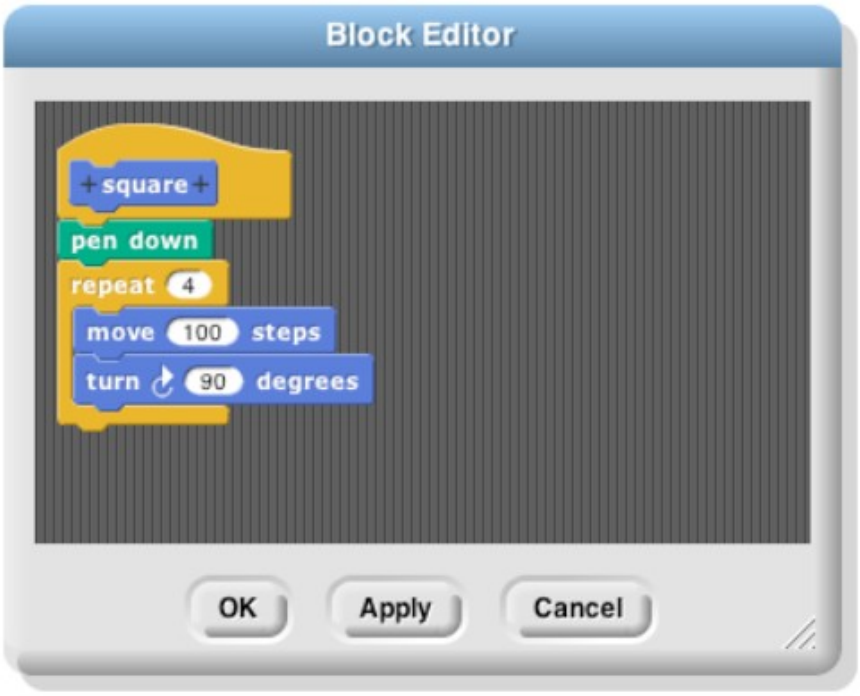
In this dialog box, you can choose the block's palette, shape, and name. With one exception, th per palette, e.g., all Motion blocks are blue. But the Variables palette includes the orange varia blocks and the red list-related blocks. Both colors are available, along with an "Other" option th blocks 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: puzzle-piece shaped blocks are Commands, and don't report a value. The oval blocks are Repo

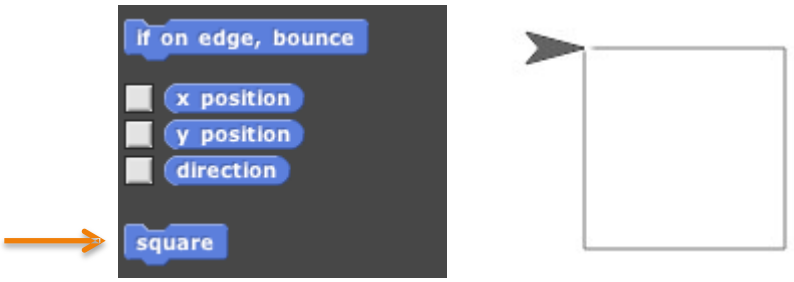
hexagonal blocks are Predicates, which is the technical term for reporters that report Boolean (t
values.

Suppose you want to make a block named “square” that draws a square. You would choose M
Command, and type “**square**” into the name field. When you click OK, you enter the Block Edit
works just like making a script in the sprite’s scripting area, except that the “hat” block at the t
saying something like “**when I am clicked**,” has a picture of the block you’re building. This ha
prototype[1] of your custom block. You drag blocks under the hat to program your custom block, then clic





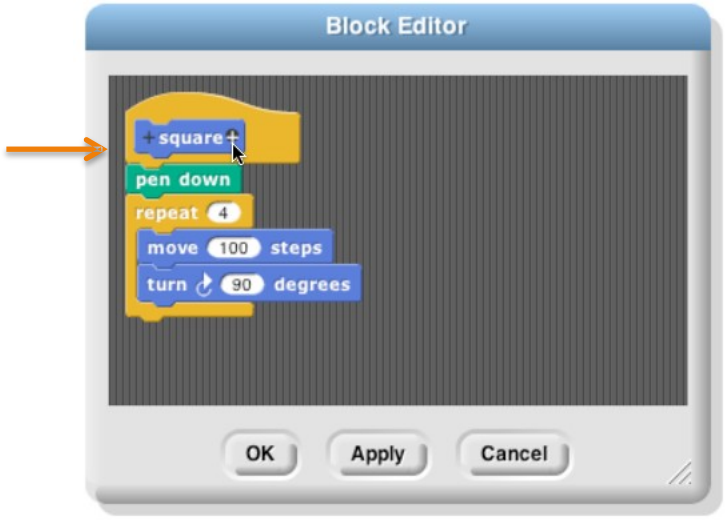
Your block appears at the bottom of the Motion palette. Here’s the block and the result of using



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

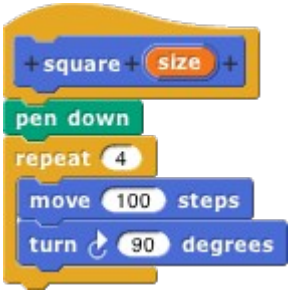
Custom Blocks with Inputs

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

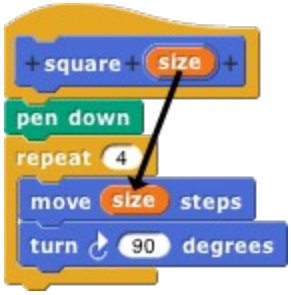


Click on the plus on the right. You will then see the "input name" dialog:

Type in the name "**size**" and click OK. There are other options in this dialog; you can choose "**t** you want to add words to the block name, so it can have text after an input (**steps**" like "**move** Or you can select a more extensive dialog with a lot of options about your input name. But we'l 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 

You can draw any size square by entering the length of its side in the box and running the block as usual, by clicking it or by putt

Editing Block Properties

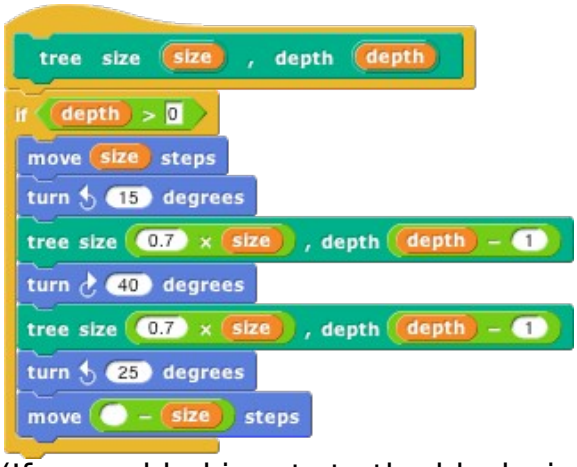
What if you change your mind about a block's color (palette) or shape (command, reporter, pre click in the hat block at the top that holds the prototype, but not in the prototype itself, you'll se which you can change the color, and *sometimes* the shape, namely, if the block is not used in a in a scripting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

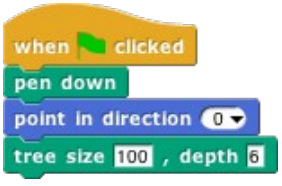
If you right-click/control-click the hat block, you get this menu:

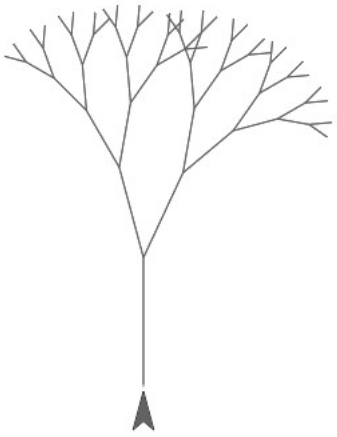
Script pic exports a picture of the script. (Many of the illustrations in this manual were made t **Translations** opens a window in which you can specify how your block should be translated if t language other than the one in which you are programming. **Block variables** lets you create a variables for this block: A script variable is created when a block is called, and it disappears wh finishes. What if you want a variable that's local to this block, as a script variable is, but doesn' between invocations? That's a block variable. If the definition of a block includes a block variab time that (custom) block is dragged from the palette into a script, the block variable is created. *copy* of the block is called, it uses the same block variable, which preserves its value between c of the block have their own block variables. The **in palette** checkbox determines whether or n be visible in the palette. It's normally checked, but you may want to hide custom blocks if you' writer creating a Parsons problem. To unhide blocks, choose "Hide blocks" from the File menu a the checkboxes. **Edit** does the same thing as regular clicking, as described earlier.

B. Recursion

Since the new custom block appears in its palette as soon as you *start* editing it, you can write (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 i drag the block from the top of the block editor rather than from the palette.)

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

Don't try to trace the exact sequence of steps that the computer follows in a recursive progra imagine that inside the computer there are many small people, and if Theresa is drawing a tree depth 6, she hires Tom to make a tree of size 70, depth 5, and later hires Theo to make another depth 5. Tom in turn hires Tammy and Tallulah, and so on. Each little person has his or her ow 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 it reports the value given; any further blocks in the script are not evaluated. Thus, the **if else** blo above could have been just an **if**, with the second block and the **report** block after it instead of inside it, and the result w be the same, because when the first **report** is seen in the base case, that finishes the block invocation, ar second **report** is ignored. There is also a **stop this block** block that has a similar purpose, ending the blo 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.)

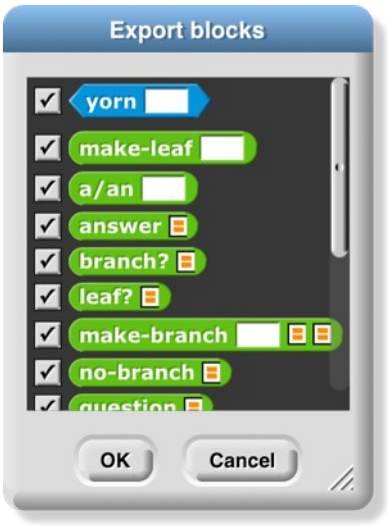
Here's a slightly more compact way to write the factorial function:

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 Chapter II above), any custom blocks you've made are saved wi sometimes you'd like to save a collection of blocks that you expect to be useful in more than on Perhaps your blocks implement a particular data structure (a stack, or a dictionary, etc.), or 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 windo



The window shows all of your global custom blocks. You can uncheck some of the checkboxes t
which blocks you want to include in your library. (You can right-click or control-click on the exp
a menu that lets you check or uncheck all the boxes at once.) Then press OK. An XML file cont
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 f
Snap! window.

Several block libraries are included with Snap!; for details about them, see page 25.

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 ste
custom block you must take these steps *in order:*

1. Turn on Visible Steppi
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 full editing capability. You can tell be
only one “OK” button at the bottom, not the usual three buttons. Use the button to close these
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 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 re give the list a name. You can't put the list into a variable, into an input slot of a block, or into a can't have lists of lists. None of the Scratch reporters reports a list value. (You can use a reduc into a text string as input to other blocks, but this loses the list structure; the input is just a text aggregate.)

A fundamental design principle in Snap! is that ***all data should be first class***. If it's in the la we should be able to use it fully and freely. We believe that this principle avoids the need for m tools, which can instead be written by Snap! users themselves.

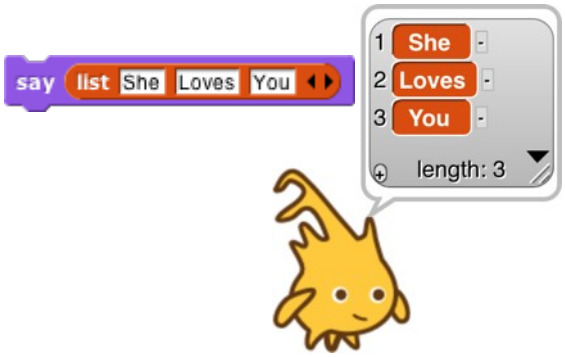
Note that it's a data *type* that's first class, not an individual value. Don't think, for example, th first class, while others aren't. In Snap!, lists are first class, period.

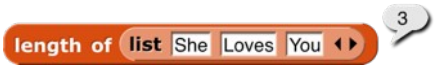
A. The list Block

At the heart of providing first class lists is the ability to make an "anonymous" list—to make a simultaneously giving it a name. The **list** block does that.

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

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





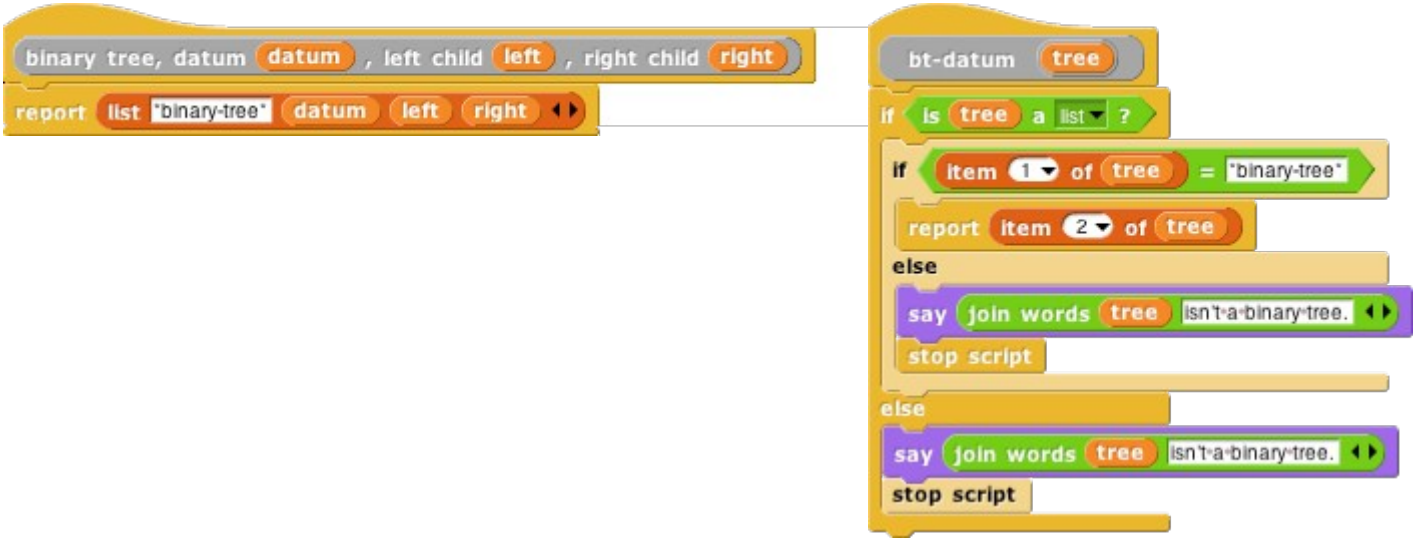
Snap! does not have a "**Make a list**" button like the one in Scratch. If you want a global "nam global variable and use the **set** block to put a list into the variable.

B. Lists of Lists

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

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

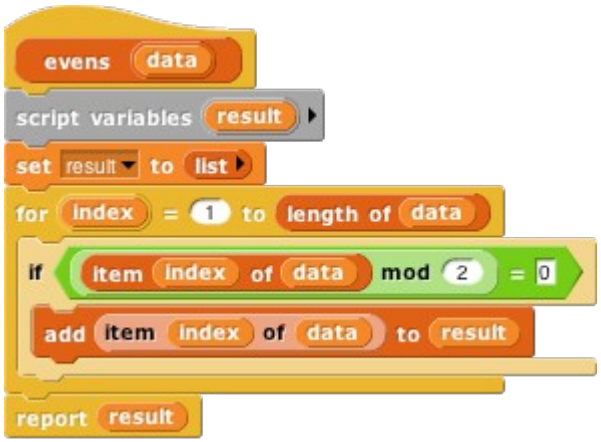
We can also build any classic computer science data structure out of lists of lists, by defining *c* to make an instance of the structure), *selectors* (blocks to pull out a piece of the structure), and change the contents of the structure) as needed. Here we create binary trees with selectors tha of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

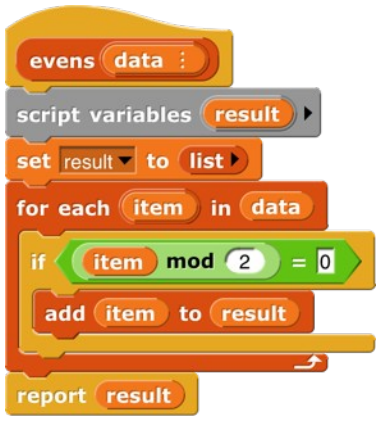


C. Functional and Imperative List Programming

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

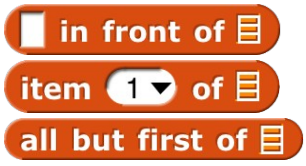
As an example, here are two blocks that take a list of numbers as input, and report a new list co even numbers from the original list:[1]





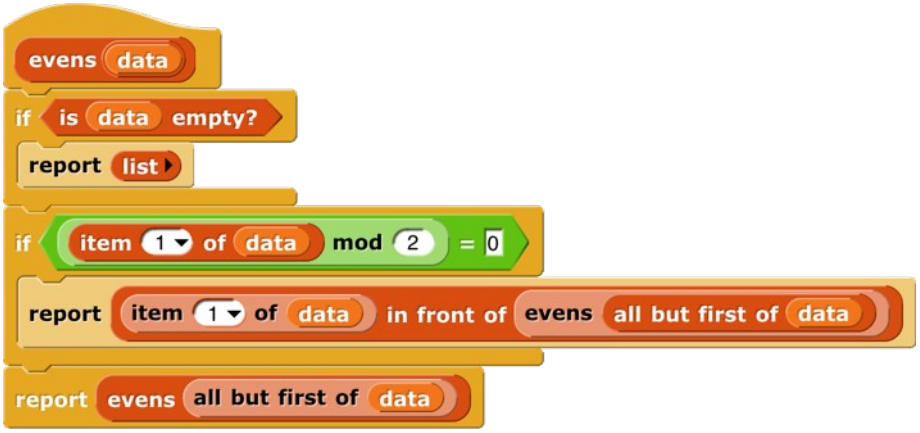
In this script, we first create a temporary variable, then put an empty list in it, then go through t input list using the **add … to (result)** block to modify the result list, adding one item at a time, report the result.

Functional programming is a different approach that is becoming important in "real world" pro because of parallelism, i.e., the fact that different processors can be manipulating the same dat time. This makes the use of mutation (changing the value associated with a variable, or the ite problematic because with parallelism it's impossible to know the exact sequence of events, so t mutation may not be what the programmer expected. Even without parallelism, though, functio programming is sometimes a simpler and more effective technique, especially when dealing wit 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 t as **for** (used in this example) and the higher order functions introduced on the next page as primitives, to be used primitives, and the desire to show how readily such tools can be implemented in Snap! itself. This is one instance pedagogic understanding that learners should both use abstractions and be permitted to see beneath the abstrac version 5.0, we used the uneasy compromise of a library of tools written in Snap! and easily, but not easily enoug project. By *not* loading the tools, users or teachers could explore how to program them. In 5.0 we made them tru because that's what some of us wanted all along and partly because of the increasing importance of fast performa "big data" and media computation. But this is not the end of the story for us. In a later version, after we get the intend to introduce "hybrid" primitives, implemented in high speed Javascript but with an "Edit" option that will op primitive implementation, but the version written in Snap!. The trick is to ensure that this can be done without dr users' projects.

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



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

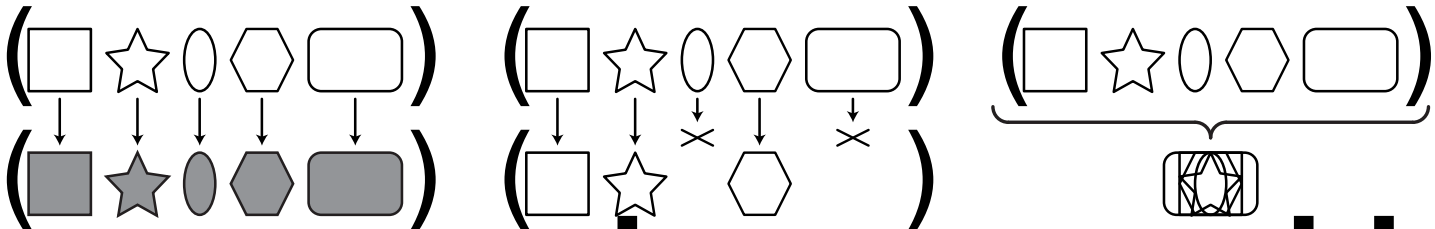
D. Higher Order List Operations and Rings

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

The **keep** block takes a Predicate expression as its first input, and a list as its second input. It r containing those elements of the input list for which the predicate returns **true**. Notice two thin predicate input: First, it has a grey ring around it. Second, the **mod** block has an empty input. item of its input list, one at a time, into that empty input before evaluating the predicate. (The supposed to remind you of the "box" notation for variables in elementary school: □+3=7.) The 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 int 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 rep itself. This allows **keep** to evaluate the **=** predicate repeatedly, once for each list item. A block another block as input is called a *higher order* block (or higher order procedure, or higher order



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 t
list of all the matching items. It's equi  but faster because it
stops looking as soon as it finds a match. If there are no matching items, it returns an empty st

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 lis

An *empty* gray ring represents the *identity function,* which just reports its input. Leaving the r
the most concise way to make a shallow copy of a list (that is, in the case of a list of lists, the re
toplevel list whose items are the same (uncopied) lists that are items of the toplevel input list).
copy of a list (that is, one in which all the sublists, sublists of sublists, etc. are copied), use the I
 block (one of the variants of the **sqrt of** block). This works because **id of** is a hyperbl

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

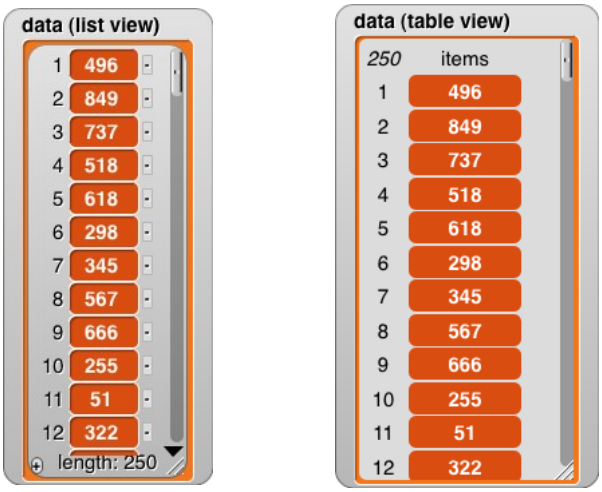
These blocks take the sum of the list items, take their product, string them into one word, comb
sentence (with spaces between items), see if all items of a list of Booleans are true, see if any o
the smallest, or find the largest.

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

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

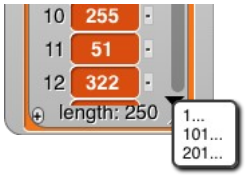
E. Table View vs. List View

We mentioned earlier that there are two ways of representing lists visually. For one-dimensiona
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 t
very versatile direct manipulation of the list through the picture: you can edit the individual iten
delete items by clicking the tiny buttons next to each item, and you can add new items at the e
tiny plus sign in the lower left corner. (You can just barely see that the item deletion buttons ha
them.) Even if you have several watchers for the same list, all of them will be updated when yo
anything. On the other hand, this versatility comes at an efficiency cost; a list view watcher for

be way too slow. As a partial workaround, the list view can only contain 100 items at a time; th
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 hundreds o
items and still scroll through them efficiently. The table view has flatter graphics for the items t
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
click menu also offers an **open in dialog...** option that opens an *offstage* table view watcher, b
can take up a lot of stage space that may make it hard to see what your program is actually doi
offstage dialog box is open, you can close the stage watcher. There's an OK button on the offst
close it if you want. Or you can right-click it to make *another* offstage watcher, which is useful i
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 any of the first ten items of th
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. Fo
lists, this display makes the content of the list very clear. A vertical display, with much of the sp
the "machinery" at the bottom of each sublist, would make it hard to show all the text at once.
cost is that the structure is no longer explicit; we can't tell just by looking that this is a list of ro
a list of column-lists or a primitive two-dimensional array type. But you can choose list view to s
structure.)

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

Snap! uses the simplest possible compromise between the two goals: It examines only the firs
list to decide on the format. If none of those are lists, or they're all lists of one item, and the ov
more than 100, list view is used. If the any of first ten items is a list, then table view is used, an
columns in the table is equal to the largest number of items among the first ten items (sublists)

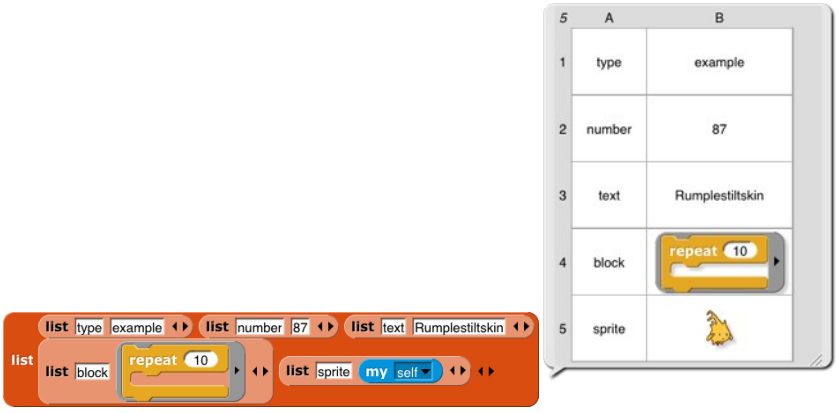
Table views open with standard values for the width and height of a cell, regardless of the actu
can change these values by dragging the column letters or row numbers. Each column has its o
changing the height of a row changes the height for all rows. (This distinction is based not on th

rows vs. columns, but on the fact that a constant row height makes scrolling through a large list
Shift-dragging a column label will change the width of that column.

If you tried out the adjustments in the previous paragraph, you may have noticed that a colum
into a number when you hover over it. Labeling rows and columns differently makes cell refere
"cell 4B" unambiguous; you don't have to have a convention about whether to say the row first
first. ("Cell B4" is the same as "cell 4B.") On the other hand, to extract a value from column B
program, you have to say **item 2 of**, not **item B of**. So it's useful to be able to find out a colun
hovering over its letter.

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

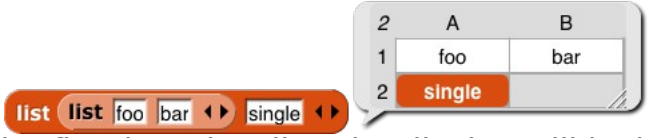
This display shows that the standard cell dimensions may not be enough for large value images
the entire speech balloon and then the second column and all the rows, we can make the result



But we make an exception for cases in which the value in a cell is a list (so that the entire table
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.
ways to see these inner sublists: You can switch to list view, or you can double-click on a list ico
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 l
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 disp

Comma-Separated Values

Spreadsheet and database programs generally offer the option to export their data as CSV (com values lists. You can import these files into Snap! and turn them into tables (lists of lists), and y tables in CSV format. Snap! recognizes a 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:

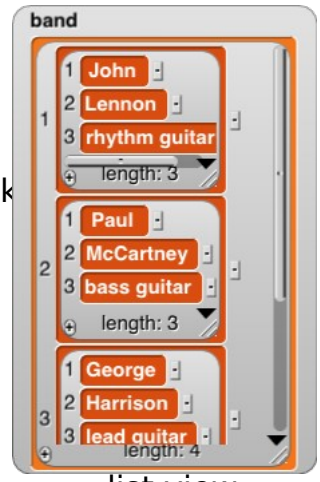
```
John,Lennon,rhythm guitar
Paul,McCartney,bass guitar
George,Harrison,lead guitar
Ringo,Starr,drums
```

Here's what the corresponding table look

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 or 

2. Right-click on the watcher and choose the "**import**" option. (If the variable's value is already click on the outside border of the watcher; there is a different menu if you click on the list itself. 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 someth 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 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 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 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 o Snap! will instead export your list as a JSON (JavaScript Object Notation) file. I modified my list:

and then exported again, getting this file:

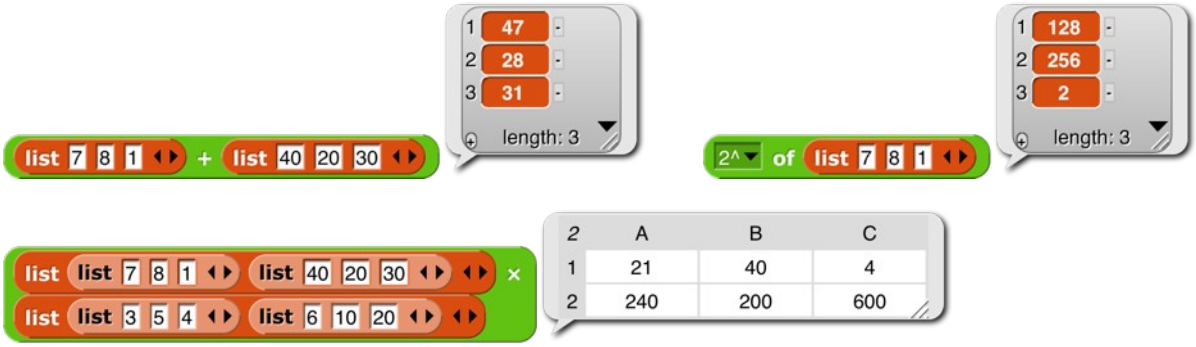
```
[["John","Lennon","rhythm guitar"],[["James","Paul"],"McCartney","bass
guitar"],["George","Harrison","lead guitar"],["Ringo","Starr","drums"]]
```

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

F. Hyperblocks

A *scalar* is anything other than a list. The name comes from mathematics, where it means a ma
direction, as opposed to a vector, which points toward somewhere. A scalar function is one whc
range are scalars, so all the arithmetic operations are scalar functions, but so are the text ones
the Boolean ones such as **not**.

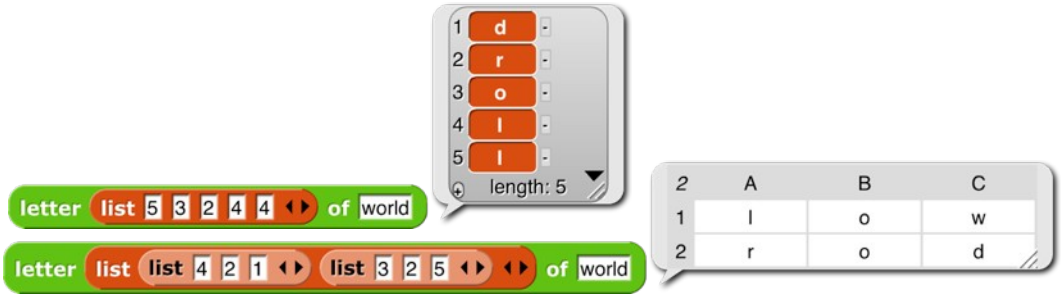
The major new feature in Snap! 6.0 is that the domain and range of most scalar function blocks
multi-dimensional lists, with the underlying scalar function applied termwise:



Mathematicians, note in the last example above that the result is just a termwise application of
function (7×3, 8×5, etc.), *not* matrix multiplication. See Appendix B for that. For a dyadic (two
if the lengths don't agree, the length of the result (in each dimension) is the length of the shorte

However, if the *number of dimensions* differs in the two inputs, then the number of dimensions
with the *higher*-dimensional input; the lower-dimensional one is used repeatedly in the missing

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



One important motivation for this feature is how it simplifies and speeds up media computation
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 (wit
transparency unchanged). By putting some sliders on the stage, you can play with colors dynar

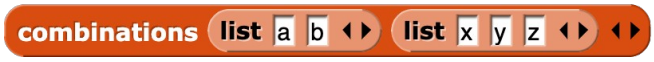
There are a few naturally scalar functions that have already had specific meanings when applie
therefore are not hyperblocks: **=** and **identical to** (because they compare entire structures, no
always reporting a single Boolean result), **and** and **or** (because they don't evaluate their secon
first input determines the result), **join** (because it converts non-scalar (and other non-text) inpu
form), and **is a** (type) (because it applies to its input as a whole). Blocks whose inputs are "nat

and

, are never hyperblocks.

The **reshape** block takes a list (of any depth) as its first input, and t
zero or more sizes along the dimensions of an array. In the example it will report a table (a mat
and three columns. If no sizes are given, the result is an empty list. Otherwise, the cells of the
are filled with the atomic values from the input list. If more values are needed than provided, th
again at the head of the list, using values more than once. If more values are provided than ne
are ignored; this isn't an error.

The **combinations** block takes any number of lists as input; it reports a list in which each item is a list whose length is the number of inputs; item *i* of a sublist is an item of input *i.* Every possible combination of items of the inputs is included, so the length of the reported list is the product of the lengths of the inputs.



6	A	B
1	a	x
2	a	y
3	a	z
4	b	x
5	b	y
6	b	z

The **item of** block has a special set of rules, designed to preserve its pre-hyper
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 inpu
may be a sublist, in which case the entire sublist is reported (the original meaning of **ite**

2. If the index is a list of numbers (no sublists), then **item of** reports a list of the indicated
(rows, in a matrix; a straightforward hyperization):

3	A	B
1	Paul	McCartney
2	John	Lennon
3	Paul	McCartney

3. If the index is a list of lists of numbers, then **item of** reports an array of only those scala
position in the list input matches the index input in all dimensions (changed in Snap! 6.6

1	A	B
1	Starr	Ringo

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

1	A	B
1	Ringo	Starr

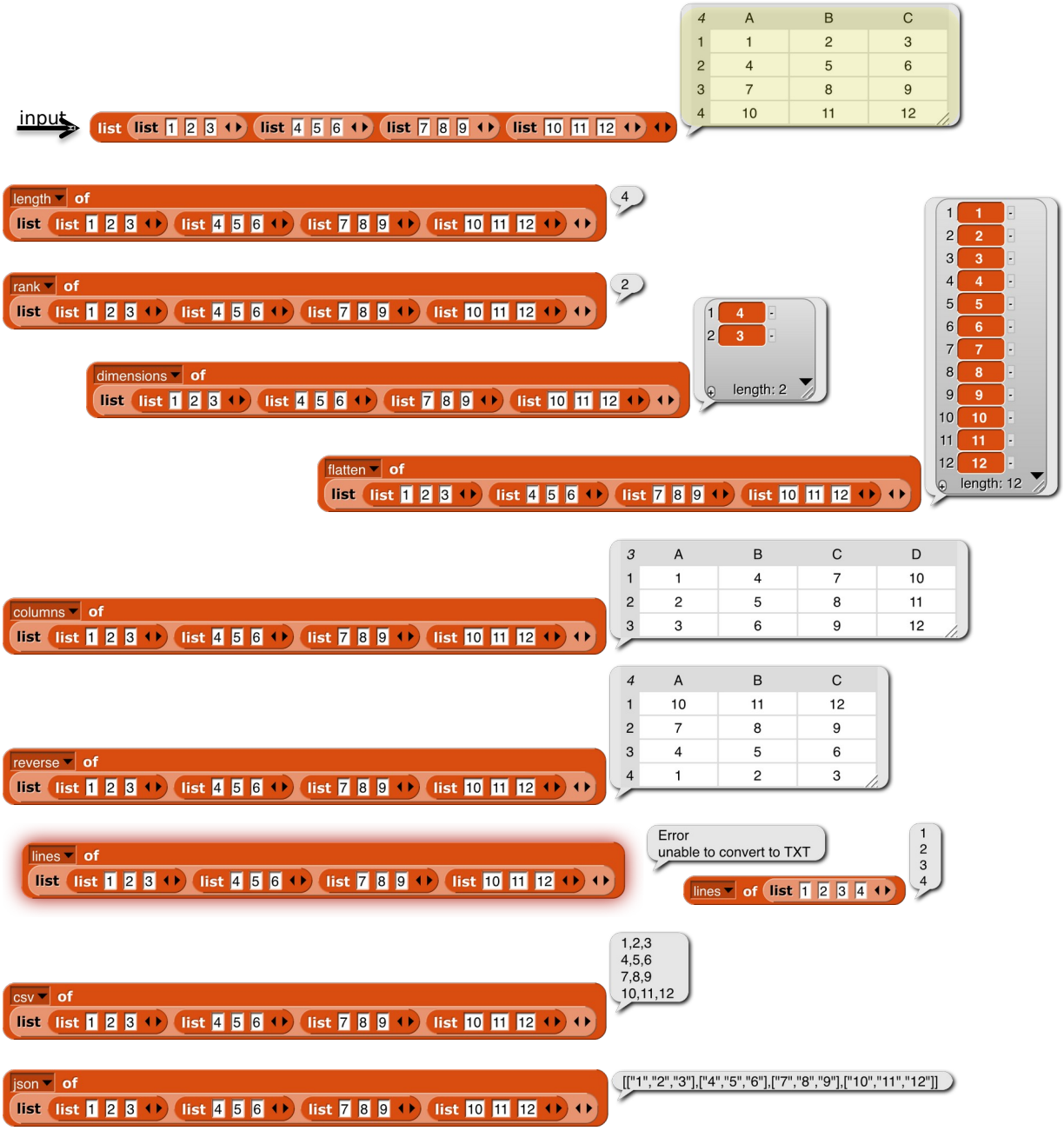
To get a column or columns of a spreadsheet, use an empty list in the row selector (changed i

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

The **length of** block is extended to provide various ways of looking at the shape and contents o
options other than **length** are mainly useful for *lists of lists,* to any depth. These new options w
hyperblocks and the APL library. (Examples are on the next page.)

length: reports the number of (toplevel) items in the list, as always.
rank: reports the number of *dimensions* of the list, i.e., the maximum depth
of lists of lists. (That example would be rank 4.)
dimensions: reports a list of numbers, each of which is the maximum lengt
dimension, so a spreadsheet of 1000 records, each with 4 fields, would repo
[1000 4].
flatten: reports a flat, one-dimensional list containing the *atomic* (non-list)
anywhere in the input list.
columns: reports a list in which the rows and columns of the input list are i
so the shape of the transpose of a shape **[1000 4]** list would be **[4 1000]**. This option works or
whose rank is at most 2. The name reflects the fact that the toplevel items of the reported tabl
of the original table.
reverse: reports a list in which the (toplevel) items of the input list are in reverse order.

The remaining three options report a (generally multi-line) text string. The input list may not in
atomic (non-list) data other than text or numbers. The **lines** option is intended for use with ran
strings; it reports a string in which each list item becomes a line of text. You can think of it as t
split by line block. The **csv** option (comma-separated values) is intended for rank-two lists tha
spreadsheet or other tabular data. Each item of the input list should be a list of atoms; the bloc
string in which each item of the big list becomes a line of text in which the items of that sublist
commas. The **json** option is for lists of any rank; it reports a text string in which the list structu
represented using square brackets. These are the opposites of **split by csv** and **split by json**



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

V. Typed Inputs

A. Scratch's Type Notation

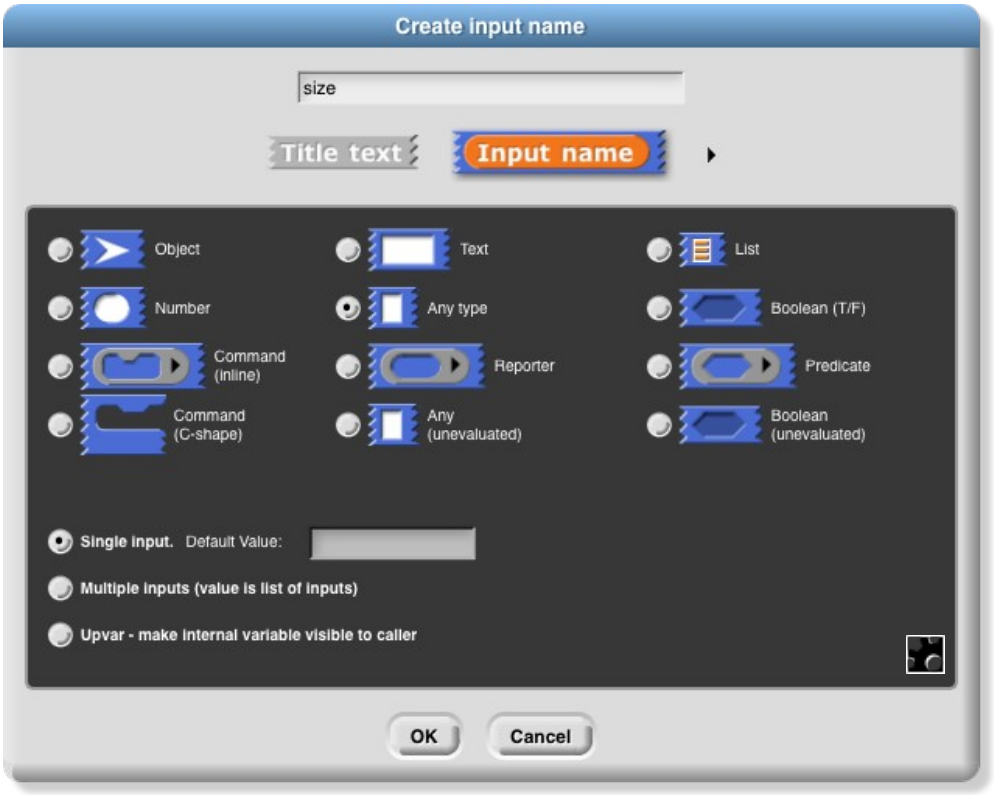
Prior to version 3, Scratch block inputs came in two types: Text-or-number type and Number typ
is indicated by a rectangular box, the latter by a rounded box: world . A third Scratch type,
Boolean (true/false), can be used in certain Control blocks with hexagonal slots.

The Snap! types are an expanded collection including Procedure, List, and Object types. Note
exception of Procedure types, all of the input type shapes are just reminders to the user of what
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**

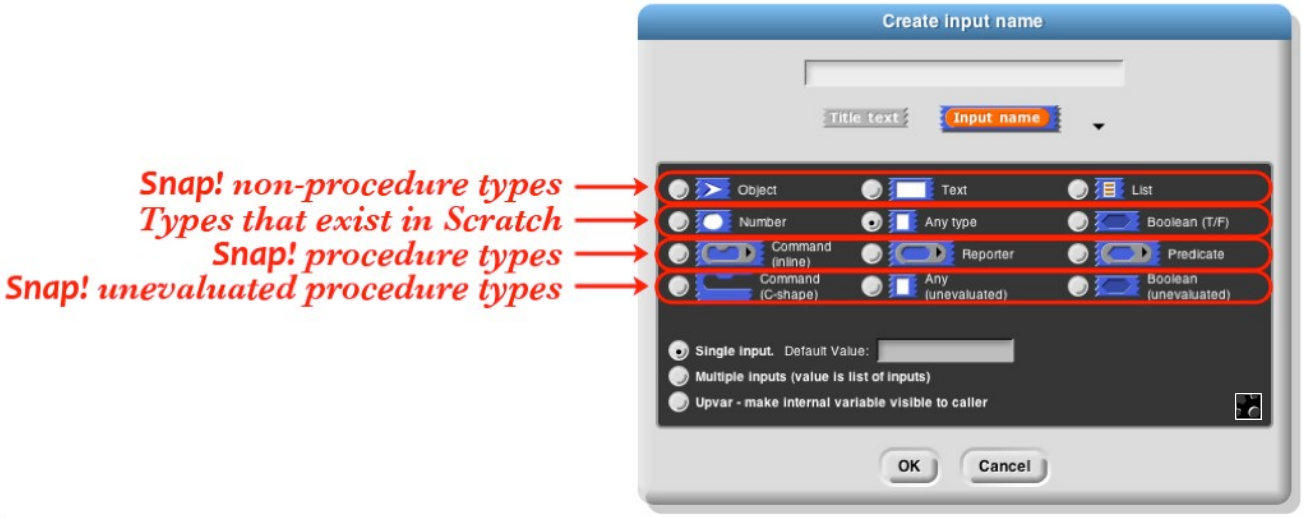
Clicking that arrowhead opens the "long" input name dialog:



There are twelve input type shapes, plus three mutually exclusive modifiers, listed in addition
choice between title text and an input name. The default type, the one you get if you don't cho

else, is “Any,” meaning that this input slot is meant to accept any value of any type. If the **size** 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 sh types is a category, and parts of each column form a category. Understanding the arrangement little easier to find the type you want.



The second row of input types contains the ones found in Scratch: Number, Any, and Boolean. these are in the second row rather than the first with the first one clear when we look at the column arrangeme The first row contains the new Snap! types other than procedures: Object, Text, and List. The las the types related to procedures, discussed more fully below.

The List type is used for first-class lists, discussed in Chapter IV above. The red rectangles inside the are meant to resemble the appearance of lists as Snap! displays them on the stage: each eleme 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 i[1]

Procedure Types

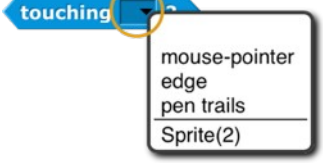
Although the procedure types are discussed more fully later, they are the key to understandin arrangement in the input types. Like Scratch, Snap! has three block shapes: jigsaw-piece for co oval for reporters, and hexagonal for predicates. (A *predicate* is a reporter that always reports Snap! these blocks are first class data; an input to a block can be of Command type, Reporter ty type. Each of these types is directly below the type of value that that kind of block reports, exc Commands, which don’t report a value at all. Thus, oval Reporters are related to the Any type, 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 that aren’t specifically about text either are of Number type or have no default value, so those rectangles are talle some of us (bh) thought that Text was a separate type that always had a wide input slot; it turns out that this isn’t the default text and the rectangle narrows), but we thought it a good idea anyway, so we allow Text-shaped boxes slots. (This is why Text comes just above Any in the input type selection box.)

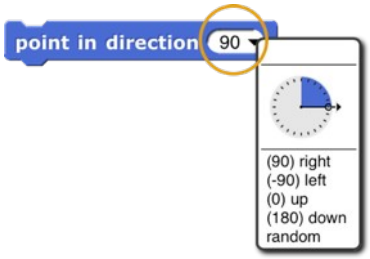
The unevaluated procedure types in the fourth row are explained in Section VI.E below. In or sentence, they combine the *meaning* of the procedure types with the *appearance* of the reporte rows higher. (Of course, this isn't quite right for the C-shaped command input type, since comm report values. 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** b



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



(indicated by the white input slot), which means that the user can type in an arbitrary input inst pulldown menu.

Custom blocks can also have such inputs. To make a pulldown input, open the long form input a text type (Any, Text, or Number) and click the bottom right corner, or control/right-click in dialog. You will see this menu:

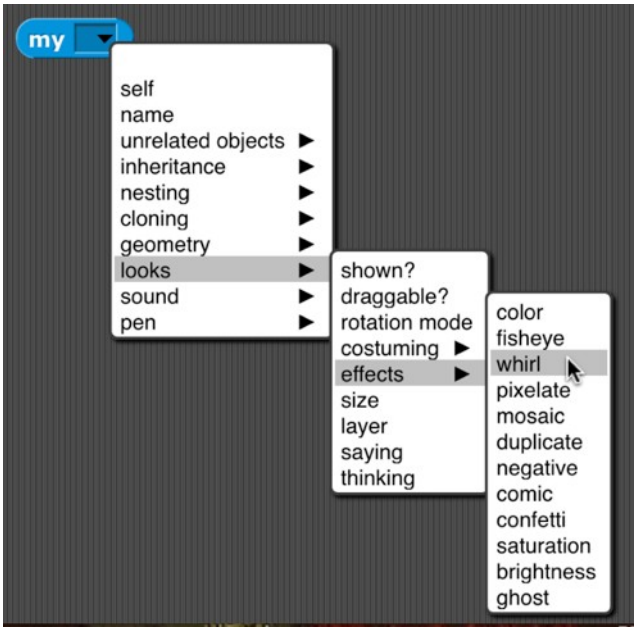
Click the **read-only** checkbox if you want a read-only pulldown input. Then from the same men
options... to get this dialog box:

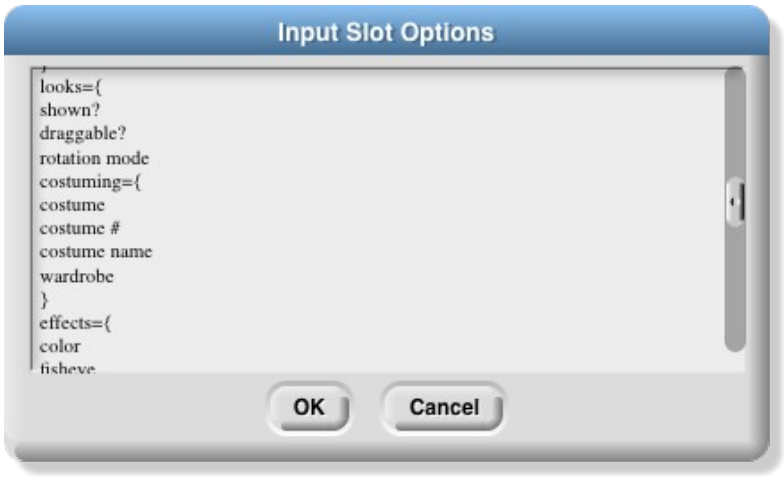
Each line in the text box represents one menu item. If the line does not contain any of the cha
then the text is both what's shown in the menu and the value of the input if that entry is chosen

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

If the line consists of a tilde **~**, then it represents a separator (a horizontal line) in the menu, u
menus into visible categories. There should be nothing else on the line. This separator is not cl
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 *sub*
before the equal sign is a name for the submenu, and will be displayed in the menu with an arro
end of the line. This line is not clickable, but hovering the mouse over it displays the submenu
original menu. A line containing a close brace **}** ends the submenu; nothing else should be on t
Submenus may be nested to arbitrary depth.



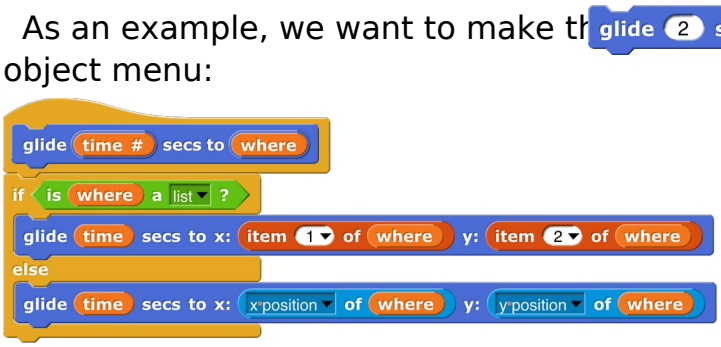


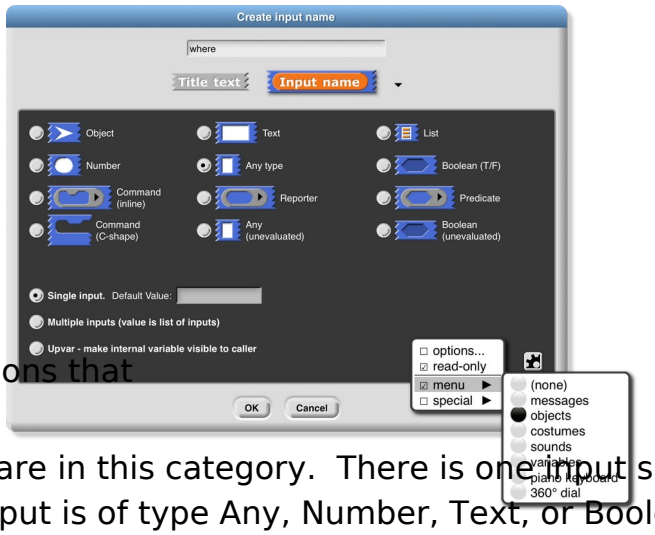
Alternatively, instead of giving a menu listing as described above, you can put a JavaScript fun
returns the desired menu in the textbox. This is an experimental feature and requires that Java
enabled in the Settings menu.

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

If the input type is something other than text, then clicking the button will inst ow this

As an example, we want to make th The second input must be a read-only object menu:

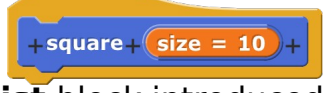




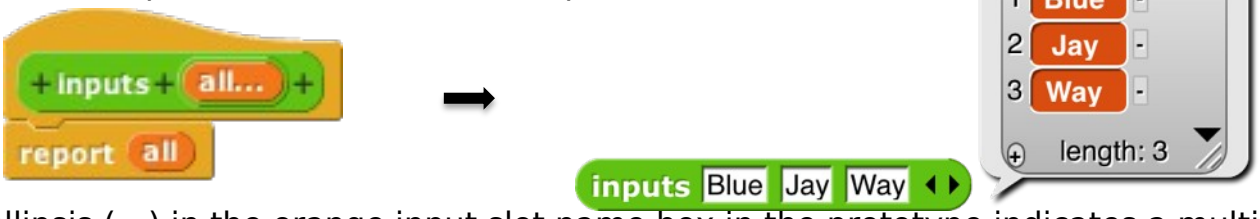
Input variants

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" i the **move (10) steps** block. In the prototype block at the top of the script in the Block Edit name "size" and default value 10 looks like this:



The "**Multiple inputs**" option: The **list** block introduced earlier accepts any number of inputs items of the new list. To allow this, Snap! introduces the arrowhead notation ( ) that expands and contracts the block, adding and removing input slots. (Shift-clicking on an arrowhead adds or removes th once.) Custom blocks made by the Snap! user have that capability, too. If you choose the "**Mu** button, then arrowheads will appear after the input slot in the block. More or fewer slots (as few used. When the block runs, all of the values in all of the slots for this input name are collected i 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 *variad*

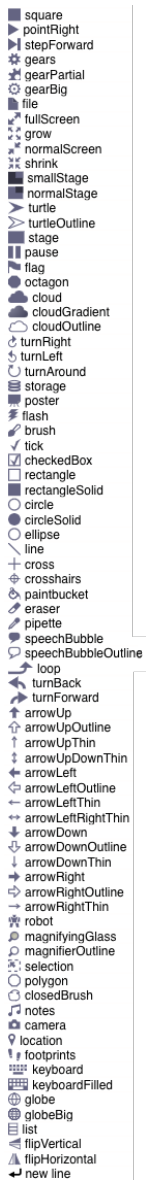
The third category, "**Upvar - make internal variable visible to caller**," isn't really an input sort of output from the block to its user. It appears as an orange variable oval in the block, rath input slot. Here's an example; the up arrow ( ) in the prototype indicates this kind of internal variable na

The variable **i** (in the block on the right above) can be dragged from the **for** block into the blocks shaped command slot. Also, by clicking on the orange **i**, the user can change the name of the v the calling script (although the name hasn't changed inside the block's definition). This kind of 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 se how it can be written in Snap!. Just give it a different name to avoid confusion, such as **my for**

Prototype Hints

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

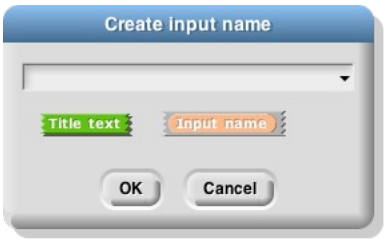


= default value ... multiple inputs ↑ upvar # number
λ procedure types: list ? Boolean object ¶ multi-lin

Title Text and Symbols

Some primitive blocks have symbols as part of the 

. Custom blocks can use symbols too. In the Block Editor, click the plus sign in the prototype at the po you want to insert the symbol. Then click the **title text** picture below the text box th 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 th Click it to see the menu shown here at the left.

Choose one of the symbols. The result will have the sym 

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

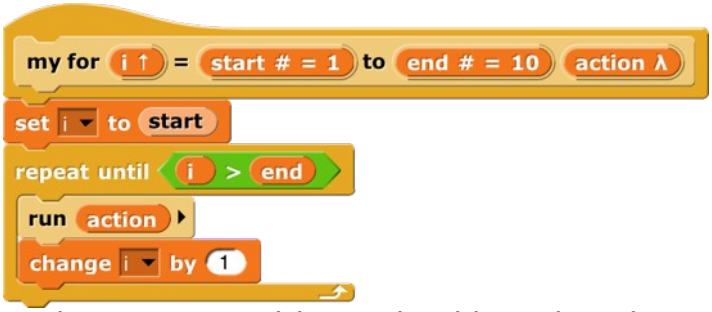
The size and color controls can also be used with text: **$foo-8-255-1** will make a huge orange "foo."

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

VI. Procedures as Data

A. Call and Run

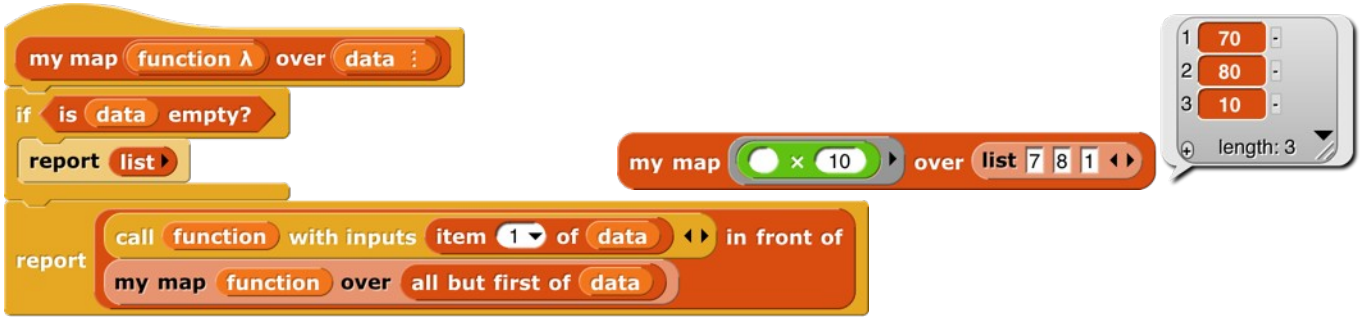
In the **for** block example above, the input named **action** has been declared as type "Command that's why the finished block is C-shaped. But how does the block actually tell Snap! to carry ou inside the C-slot? Here is a simple version of the block script:



This is simplified because it assumes, without checking, that the ending value is greater than th not, the block should (depending on the designer's purposes) either not run at all, or change the 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 comma takes a Command-type value (a script) as its input, and carries out its instructions. (In this exa the inpu action is the script that the user puts in the C-slot of the **my for** block.) There is a simila block for invoking a Reporter or Predicate block. The **call** and **run** blocks are at the heart of Sn procedure feature; they allow scripts and blocks to be used as data—in this example, as an inpu 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 50):



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

Call/Run with inputs

The **call** block (like the **run** block) has a right arrowhead at the end; clicking on it adds the phr 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. Th arrowhead can be clicked as many times as needed for the number of inputs required by the re being called.

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

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

An even more important thing to notice about these examples is the *ring* around the Reporter- in **call** and **map** above. This notation indicates that *the block itself,* not the number or other va would report when called, is the input. If you want to use a block itself in a non-Reporter-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), choices in the menu that appears is “**ringify**” and/or “**unringify**.” The ring indicating a Report Predicate-type input slot is essentially the same idea for reporters as the C-shaped input slot wi already familiar; with a C-shaped slot, it’s *the script* you put in the slot that becomes the input t block.

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

Variables in Ring Slots

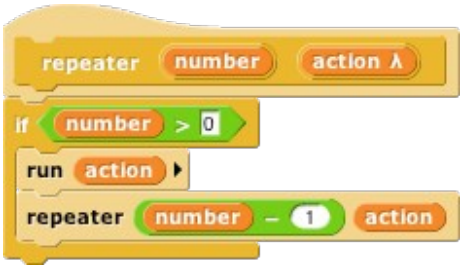
Note that the **run** block in the definition of the **my for** block (page 65) doesn’t have a ring aro variable **action**. When you drag a variable into a ringed input slot, you generally *do* want to us variable, which will be the block or script you’re trying to run or call, rather than the orange vari itself. So Snap! automatically removes the ring in this case. If you ever do want to use the vari rather than the value of the variable, as a Procedure-type input, you can drag the variable into t then control-click or right-click it and choose “**ringify**” from the menu that appears. (Similarly, to call a function that will report a block to use as the input, such as **item 1 of** applied to a list choose “**unringify**” from the menu. Almost all the time, though, Snap! does what you mean wi

B. Writing Higher Order Procedures

A *higher order procedure* is one that takes another procedure as an input, or that reports a proc document, the word “procedure” encompasses scripts, individual blocks, and nested reporters. otherwise, “reporter” includes predicates. When the word is capitalized inside a sentence, it me oval-shaped blocks. So, “nested reporters” includes predicates, but “a Reporter-type input” doe

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

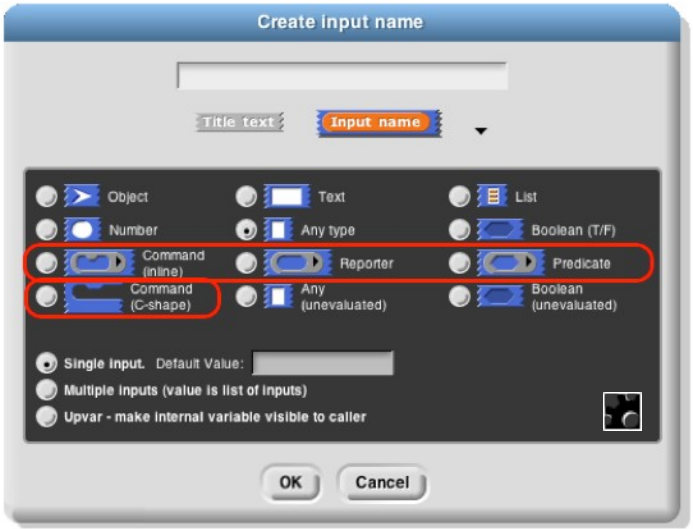
Why would you want a block to take a procedure as input? This is actually not an obscure thin
primitive conditional and looping blocks (the C-shaped ones in the Control palette) take a script
just don't usually think about it in those terms! We could write a custom block this way, if
Snap! didn't already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



The lambda (λ) next to **action** in the prototype indicates that this is a C-shaped block, and that the scr
enclosed by the C when the block is used is the input named **action** in the body of the script. T
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 th

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



Why would you ever choose an inline Command slot rather than a C shape? Other than the **run** discussed below, the only case I can think of is something like the C/C++/Java **for** loop, which a 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 commands) and **call** (for reporters). The **run** block's script input is an inline ring, not C-shaped anticipate that it will be rare to use a specific, literal script as the input. Instead, the input will g 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 procedures. How does Snap! know where to use those inputs? If the called procedure (block or 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! fil from left to right:

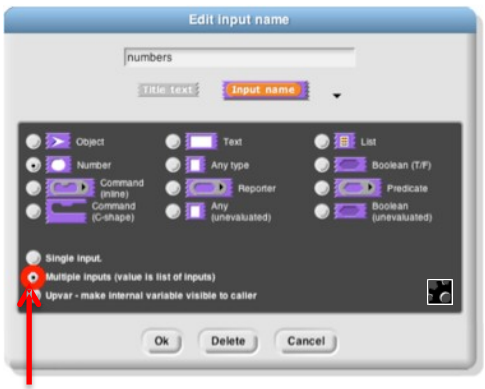
2. If exactly one input is provided, Snap! will fill any number of empty slots with it:

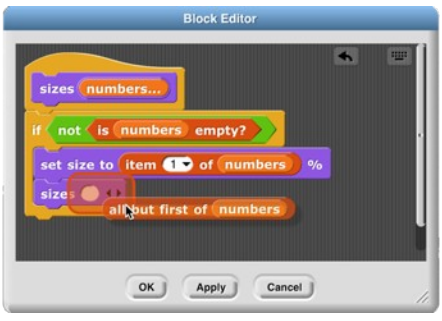
3. Otherwise, Snap! won't fill any slots, because the user's intention is unclear.

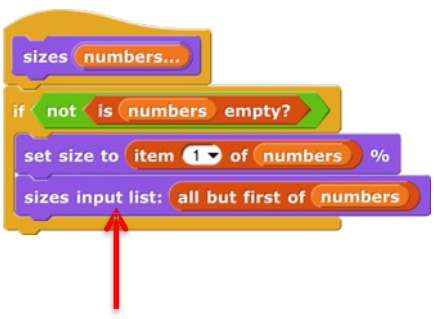
If the user wants to override these rules, the solution is to use a ring with explicit input names into the given block or script to indicate how inputs are to be used. This will be discussed more

Recursive Calls to Multiple-Input Blocks

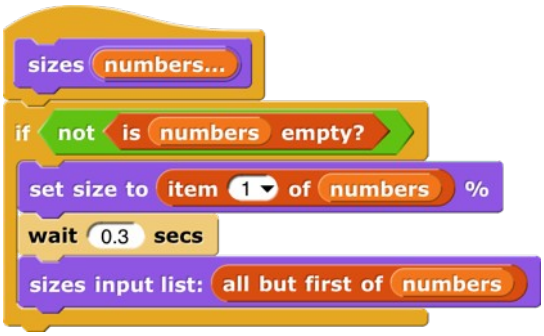
A relatively rare situation not yet considered here is the case of a recursive block that has a va inputs. Let's say the user of your project calls your block with five inputs one time, and 87 inpu How do you write the recursive call to your block when you don't know how many inputs to give answer is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







Note that the halo you see while dragging onto the arrowheads is red instead of white, and cove
as well as the arrowheads. And when you drop the expression onto the arrowheads, the words
added to the block text and the arrowheads disappear (in this invocation only) to remind you th
represents all of the multiple inputs, not just a single input. The items in the list are taken *indiv*
the script. Since **numbers** is a list of numbers, each individual item is a number, just what **size**
block will take any number of numbers as inputs, and will make the sprite grow and shrink acco



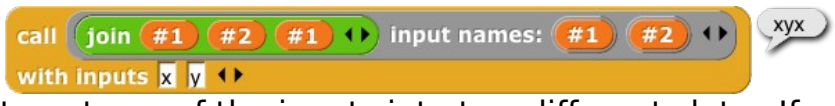
The user of this block calls it with any number of *individual numbers* as inputs. But inside the d
block, all of those numbers form *a list* that has a single input name, **numbers**. This recursive d
checks to make sure there are any inputs at all. If so, it reports (**item 1 of** the list), then it wan
to make a recursive call with all but the first **numbers**. But **sizes** doesn't take a list as input; it takes numb
inputs! So this would be wrong:

C. Formal Parameters

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

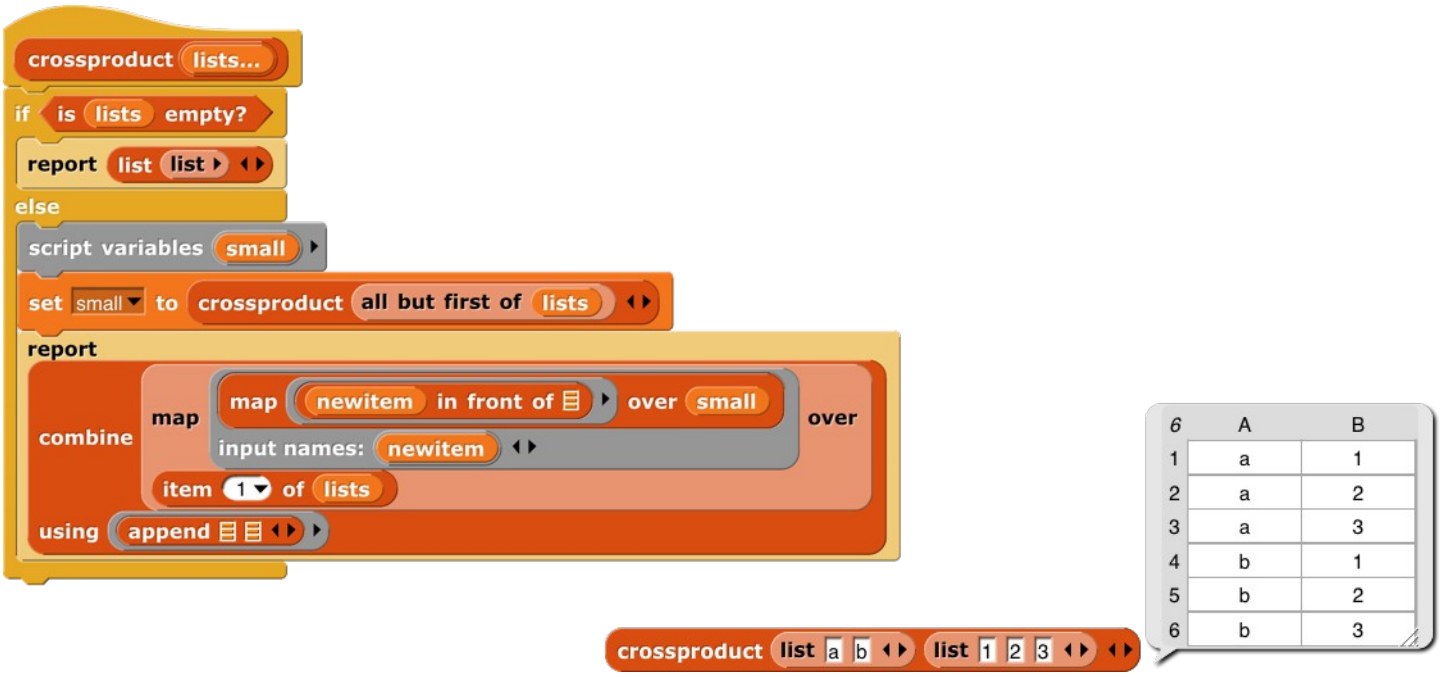
The names **#1**, **#2**, etc. are provided by default, but you can change a name by clicking on its c
the **input names** list. Be careful not to *drag* the oval when clicking; that's how you use the inp
The names of the input variables are called the *formal parameters* of the encapsulated procedu

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



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

Here is a more realistic, much more advanced example:

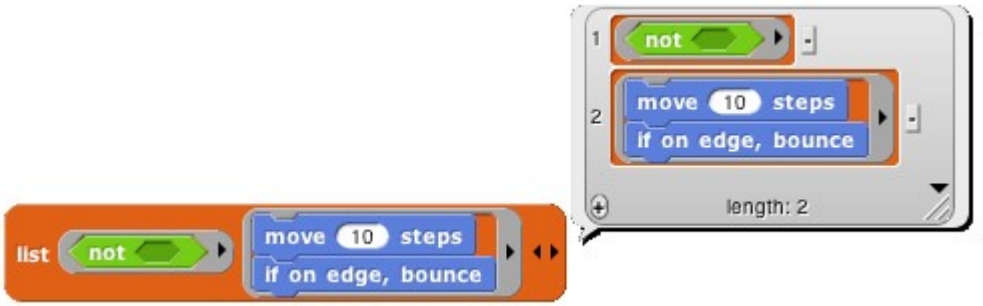


This is the definition of a block that takes any number of lists, and reports the list of all possible one item from each list. The important part for this discussion is that near the bottom there are to **map**, the higher order function that applies an input function to each item of an input list.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

By the way, once the called block provides names for its inputs, Snap! will not automatically fi the theory that the user has taken control. In fact, that's another reason you might want to nar 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 pu 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 in ringed. We must say explicitly that we want the block *itself* as the input, rather than whatever from evaluating the block.

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

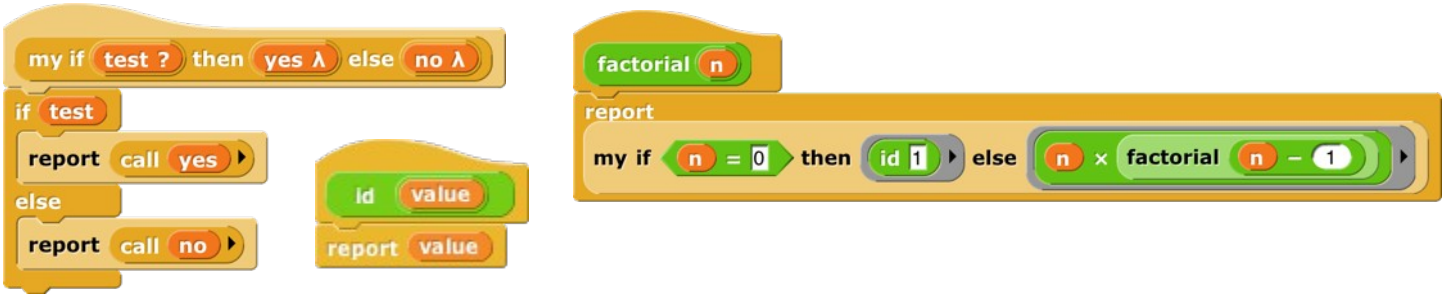
E. Special Forms

The primitive **if-else** block has two C-shaped command slots and chooses one or the other depen
Boolean test. Because Scratch doesn't emphasize functional programming, it lacks a correspon
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

The problem is that when any block is called, all of its inputs are computed (evaluated) before t
runs. The block itself knows only the values of its inputs, not what expressions were used to co
particular, all of the inputs to the **if-else** block are evaluated first. That means that even in the base
case, **factorial** will try to call itself recursively, causing an infinite loop. We need a block like **if-else**
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
Any. Then, when calling the block, those inputs will be enclosed in a ring so that the expression
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 repo
longer as intuitively obvious as the Scratch command-**if**. You have to know about procedures a
rings, and about a trick to get a constant value in a ringed slot. (The **id** block implements the id

which reports its input.[1] We need it because rings take only reporters as input, not numbers.) What
a reporter-**if** that *behaves* like this one, delaying the evaluation of its inputs, but *looks* like our f
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
evaluation of that input by wrapping an "invisible ring" around it (and, if necessary, an **id**-like tr
constant data into constant functions) is called a *special form*. To turn our **if** block into a specia
block's prototype, declaring the inputs **yes** and **no** to be of type "**Any (unevaluated)**" instead
The script for the block is still that of the second version, including the use of **call** to evaluate e
but not both. But the slots appear as white Any-type rectangles, not Reporter-type rings, and th
block will look like our first attempt.

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

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

Special Forms in Scratch

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

Also, the C-shaped slot familiar to Scratch users is an unevaluated procedure type; you don't h
to keep the commands in the C-slot from being run before the C-shaped block is run. Those co
themselves, not the result of running them, are the input to the C-shaped Control block. (This is
granted by Scratch users, especially because Scratchers don't think of the contents of a C-slot a
This is why it makes sense that "C-shaped" is on the fourth row of types in the long form input d
other unevaluated types.

[1] There is a primitive **id** function in the menu of the **sqrt of** block, but we think seeing its (very simple) implement
example easier to understand.

VII. Object Oriented Programming with Sprites

Object oriented programming is a style based around the abstraction *object:* a collection of *da* (procedures, which from our point of view are just more data) that you interact with by sending a name, maybe in the form of a text string, and perhaps additional inputs). The object respond by carrying out a method, which may or may not report a value back to the asker. Some people *data hiding* aspect of OOP (because each object has local variables that other objects can acces request messages to the owning object) while others emphasize the *simulation* aspect (in which abstractly represents something in the world, and the interactions of objects in the program mo interactions of real people or things). Data hiding is important for large multi-programmer indus but for Snap! users it's the simulation aspect that's important. Our approach is therefore less re 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 any object can send a message to another object. (2) *Local state:* Each object can remember th history of the computation it has performed. ("Important" means that it need not remember ev has handled, but only the lasting effects of those messages that will affect later computation.) would be impractical if each individual object had to contain methods, many of them identical to objects, for all of the messages it can accept. Instead, we need a way to say that this new obje old object except for a few differences, so that only those differences need be programmed exp

A. First Class Sprites

Like Scratch, Snap! comes with things that are natural objects: its sprites. Each sprite can own each sprite has its own scripts (methods). A Scratch animation is plainly a simulation of the inte characters in a play. There are two ways in which Scratch sprites are less versatile than the obj language. First, Scratch message passing is weak in three respects: Messages can only be **bro** addressed to an individual sprite; messages can't take inputs; and methods can't return values Second, and more basic, in the OOP paradigm objects are *data;* they can be the value of a varia 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 o sent messages individually. The children of a sprite can inherit sprite-local variables, methods (procedures), and other attributes (e.g., **x position**).

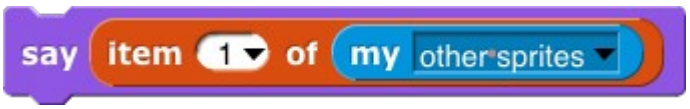
The fundamental means by which programs get access to sprites is the **my** reporter block. It h menu input slot that, when clicked, gives access to all the sprite 

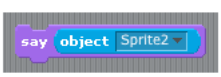
he stage. rep the one asking the ques 

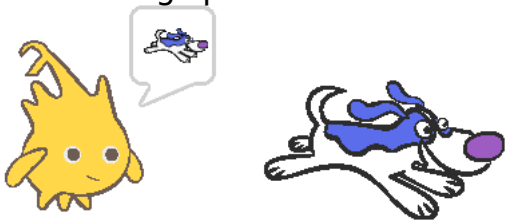
reports a list of all sprites other than the one asking the qu 

reports a list of all sprites that are *near* the one asking—the ones that are candidate collided with this one, for example. The **my** block has many other options, discussed below. If 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 type, such as **set**'s second input. If you **say** an object, the resulting speech balloon will contain 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. (Th
command version, for historical reasons.) There are two different kinds of situations in which cl
One is that you've made an example sprite and, when you start the project, you want a fairly la
essentially identical sprites that behave like the example. (Hereafter we'll call the example spri
and the others the "children.") Once the game or animation is over, you don't need the copies
we'll see, "copies" is the wrong word because the parent and the children *share* a lot of propert
we use the word "clones" to describe the children rather than "copies.") These are ***temporary***
are automatically deleted when the user presses either the green flag or the red stop sign. In Sc
later, all clones are temporary.

The other kind of situation is what happens when you want specializations of sprites. For exam
you have a sprite named Dog. It has certain behaviors, such as running up to a person who con
you decide that the family in your story really likes dogs, so they adopt a lot of them. Some are
who wag their tails when they see you. Others are rottweilers, who growl at you when they see
make a clone of Dog, perhaps rename it Cocker Spaniel, and give it a new costume and a script
when someone gets near. You make another clone of Dog, perhaps rename it Rottweiler, and g
costume, etc. Then you make three clones of Cocker Spaniel (so there are four altogether) and
Rottweiler. Maybe you hide the Dog sprite after all this, since it's no breed in particular. Each c
position, special behaviors, and so on. You want to save all of these dogs in the project. These
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
you're curious, one reason is that permanent clones appear in the sprite corral, where their pict
updated to reflect the clone's current costume, direction, and so on.) We have tried to anticipat
follows: When you make a clone in a script 

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

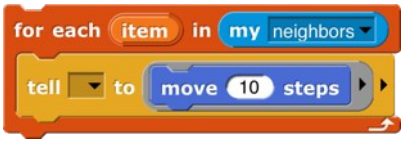
You can change a temporary sprite to permanent by right-clicking it and choosing "edit." (It's
rather than, say, "permanent" because it also shifts the scripting area to reflect that sprite, as i
button in the sprite corral.) You can change a permanent sprite to temporary by right-clicking it
"release."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

with **true** or
false as the second input.

C. Sending Messages to Sprites

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

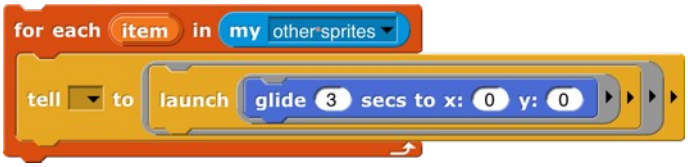
The way to send a message to a sprite (or the stage) is with the **tell** block (for command mess
block (for reporter messages).



A small point to note in the examples above: all dropdown menus include an empty entry at th
can be selected for use in higher order procedures like the **for each** and **map** examples. Each
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 co
has to be the case for **ask**, since we want to do something with the value it reports.) So **tell** is
broadcast and wait. Sometimes the other sprite's method may take a long time, or may ever
so you want the originating script to continue without waiting. For this purpose we have the **lau**



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 sp
That interface was designed back when we were trying hard to avoid adding new primitive bloc
to write **ask** and **tell** as tool procedures in Snap! itself. That technique still works, but is depre
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
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 bloc
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 diffe
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 **gr** Fido inherits the **greet** method from Dog, but Dog itself couldn't actually run that method, beca have **greet as friend** or **greet as enemy**. And perhaps only individual dogs such as Fido have Even though the **greet** method is defined in the Dog sprite, when it's running it remembers wha sprite called it, so it knows which **greet as friend** to use. Dog's **greet** block is called a *polymo* because it means different things to different dogs, even though they all share the same script.

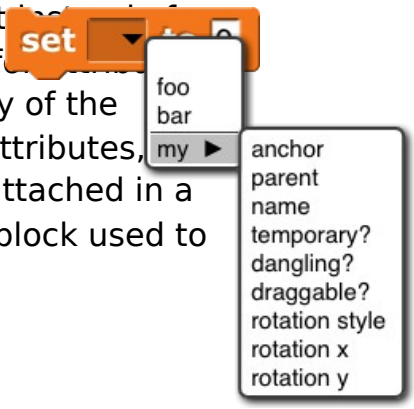
D. Local State in Sprites: Variables and Attributes

A sprite's memory of its own past history takes two main forms. It has *variables,* created explic with the "Make a variable" button; it also has *attributes,* the qualities every sprite has automati position, direction, and pen color. Each variable can be examined using its own orange oval blo **set** block to modify all variables. Attributes, however, have a less uniform programming interfa

- 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**, **turn**, and **if on edge, bounce**.
- There is no way for a script to examine a sprite's *pen color,* but the **pen color** block, **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 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 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 f The **my** block's menu (in Sensing; see page 78) includes many of the attributes of a sprite. It serves as a general getter for those attributes, **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 part and an instance is an *actual object* of that type. For example, there might be a Dog class, and s Fido, Spot, and Runt. The class typically specifies the methods shared by all dogs (RollOver, Sit Fetch, and so on), and the instances contain data such as species, color, and friendliness. Snap approach called *prototyping,* in which there is no distinction between classes and instances. Pr suited to an experimental, tinkering style of work: You make a single dog sprite, with both meth data (variables); you can actually watch it and interact with it on the stage; and when you like it prototype from which to clone other dogs. If you later discover a bug in the behavior of dogs, y method in the parent, and all of the children will automatically share the new version of the met Experienced class/instance programmers may find prototyping strange at first, but it is actually expressive system, because you can easily simulate a class/instance hierarchy by hiding the pr

Prototyping is also a better fit with the Scratch design principle that everything in a project shou
and visible on the stage; in class/instance OOP the programming process begins with an abstrac
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 '
at the bottom right corner of the window, you get a menu that includes "clone" as one of the ch
an **a new clone of** block in the Control palette that creates and reports a child sprite. And spri
"parent" attribute that can be set, like any attribute, thereby *changing* the parent of an existing

F. Inheritance by Delegation

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

When a new clone is created, by default it shares only its methods, wardrobe, and jukebox wit
other properties are copied to the clone, but not shared. (One exception is that a new *permane*
random position. Another is that *temporary* clones share the scripts in their parent's scripting a
sprite-local variables that the parent creates *after* cloning are shared with its children.) If the va
property is changed in the parent, then the children see the new value. If the value of a shared
changed in the *child*, then the sharing link is broken, and a new private version is created in tha
the mechanism by which a child chooses not to share a property with its parent.) "Changed" in
means using the **set** or **change** block for a variable, editing a block in the Block Editor, editing
sound, or inserting, deleting, or reordering costumes or sounds. To change a property from uns
the child uses the **inherit** command block. The pulldown menu in the block lists all the things t
inherit from its parent (which might be nothing, if this sprite has no parent) and is not already in
that would prevent **tell**ing a child to inherit, so if the **inherit** block is inside a ring, its pulldown
the things a child could inherit from this sprite. Right-clicking on the scripting area of a perman
a menu option to share the entire collection of scripts from its parent, as a temporary clone doe

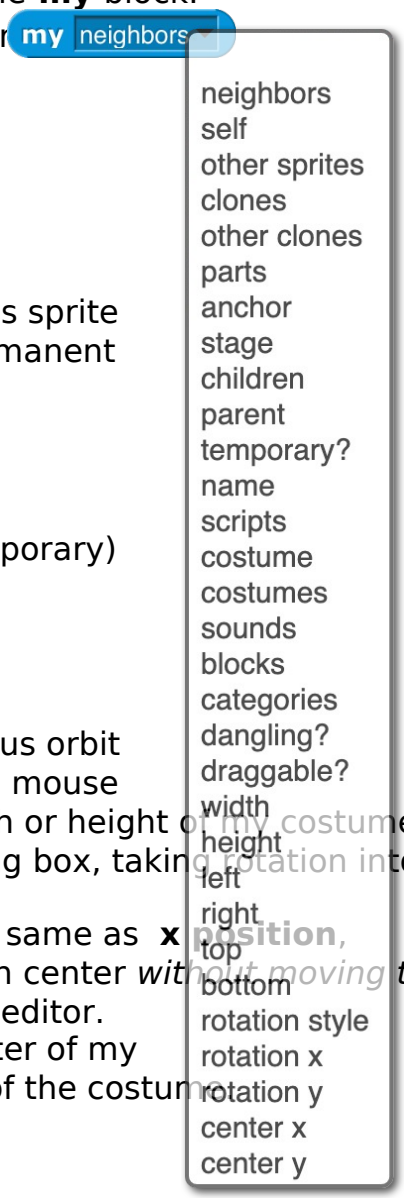
[1] Some languages popular in the "real world" today, such as JavaScript, claim to use prototyping, but their object s
complicated than what we are describing (we're guessing it's because they were designed by people too familiar
programming); that has, in some circles, given prototyping a bad name.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

The rules are full of details, but the basic idea is simple: Parents can change their children, bu
directly change their parents. That's what you'd expect from the word "inherit": the influence j
direction. When a child changes some property, it's declaring independence from its parent (wi
one property). What if you really want the child to be able to make a change in the parent (and
itself and all its siblings)? Remember that in this system any object can **tell** any other object to

When a sprite gets a message for which it doesn't have a corresponding block, the message is
sprite's parent. When a sprite does have the corresponding block, then the message is not dele
script that implements a delegated message uses **(self)** to mean the child to which the message wa
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 attr



- **self**: this sprite
- **neighbors**: a list of *nearby*[1] 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
- **blocks**: a list of the blocks visible in this sprite
- **categories**: a list of all the block category names
- **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.

[1] *Neighbors* are all other sprites whose bounding boxes intersect the doubled dimensions of the requesting sprite's

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 you can make a list of them, put them in variables, use them as input to a procedure, and so on and **my [sounds]** report lists of them.

Media Computation with Costumes

The components of a costume are its name, width, height, and pi 

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

But the really interesting part of a costume is its bitmap, a list of *pixels*. (A pixel, short for "pic represents one dot on your display.) Each pixel is itself a list of four items, the red, green, and b of its color (in the range 0-255) and what is standardly called its "transparency" but should be c also in the range 0-255, in which 0 means that the pixel is invisible and 255 means that it's fully can't see anything from a rearward layer at that point on the stage. (Costume pixels typically h 0 only for points inside the bounding box of the costume but not actually part of the costume; p interior of a costume typically have an opacity of 255. Intermediate values appear mainly at th costume, or at sharp boundaries between colors inside the costume, where they are used to rec stairstep-like shape of a diagonal line displayed on an array of discrete rectangular screen coord the opacity of a *sprite* pixel is determined by combining the costume's opacity with the sprite's 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. pixel list has to be combined with the dimensions to produce a costume. This choice partly refle bitmaps are stored in the computer's hardware and operating system, but also makes it easy to transformations of a costume with **map**:

In this simplest possible transformation, the red value of all the pixels have been changed to a c Colors that were red in the original (such as the logo printed on the t-shirt) become closer to bla color components being near zero); the blue jeans become purple (blue plus red); perhaps coun white t-shirt, which has the maximum value for all three color components, loses some of its red cyan, the color opposite red on the color wheel. In reading the code, note that the function that to **map** is applied to a single pixel, whose first item is its red component. Also note that this pro on bitmap costumes; if you call **pixels of** on a vector costume (one with "svg" in the corner of i 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 block creates a costume by combining a bitmap, a width, and a height. But, as in the example **costume** will accept a bitmap as input and will automatically use the width and height of the cu Note that there's no **name** input; costumes computed in this way are all named **costume**. Not of **switch to costume** does *not* add the computed costume to the sprite's wardrobe; to do that

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 cartoonis Alternatively, you can do the computation taking advantage of hyperblocks:

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

It's the li 

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

The **stretch** block takes a costume as its first input, either by selecting a costume from the m dropping a costume-valued expressio 

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

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, regardl given. The camera remains on until you click the red stop button, your program runs the **stop** turn it off explicitly wit 

block. The video image on the stage is partly ghosted, extent determined by the **set video transparency** block, whose input really is transparency a

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

block.

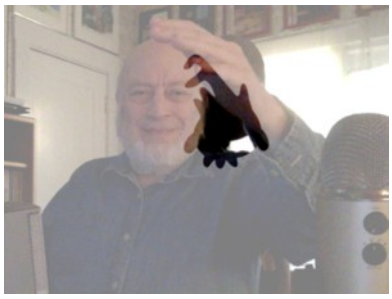
The **video snap on** block then takes a still picture from the camera, and trims it to fit on the s
(**Video snap on stage** means to use the entire stage-sized rectangle.) For example, here's a c
trimmed to fit Alonzo:

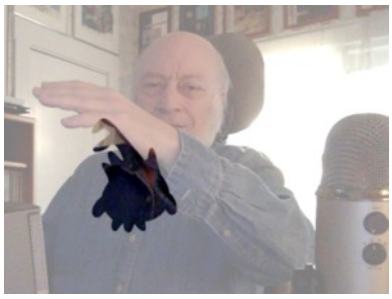
The "Video Capture" project in the Examples collection repeatedly takes such trimmed snapsh
Alonzo sprite use the current snapshot as its costume, so it looks like this:

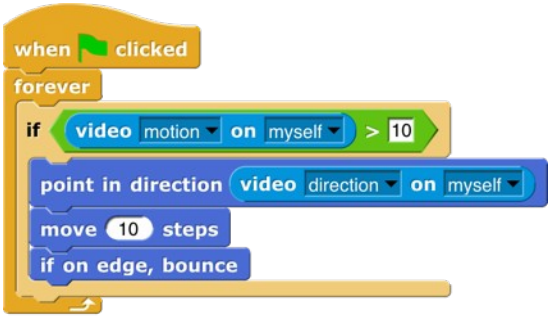
(The picture above was actually taken with transparency set to 50, to make the background mo
printing.) Because the sprite is always still in the place where the snapshot was taken, its costu
with the rest of the full-stage video. If you were to add a **move 100 steps** block after the **swit
you'd see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

This time, the sprite's costume was captured at one position, and then the sprite is shown at a c
(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:

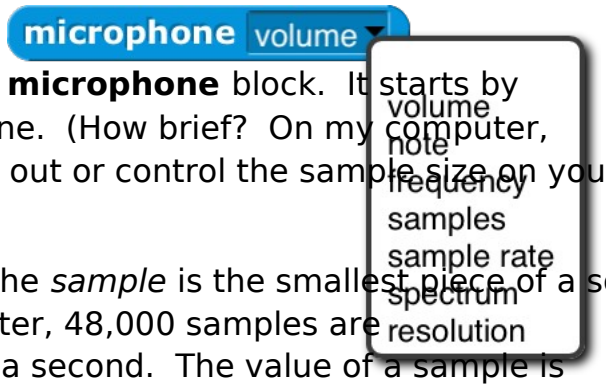






(Really these should be Jens's picture; it's his project. But he's [illegible]) **Video motion** compares two snapshots a moment apart, looking only at the part within the given trim (here **myself**, meanin sprite, not the person looking into the camera), to detect a difference between them. It reports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

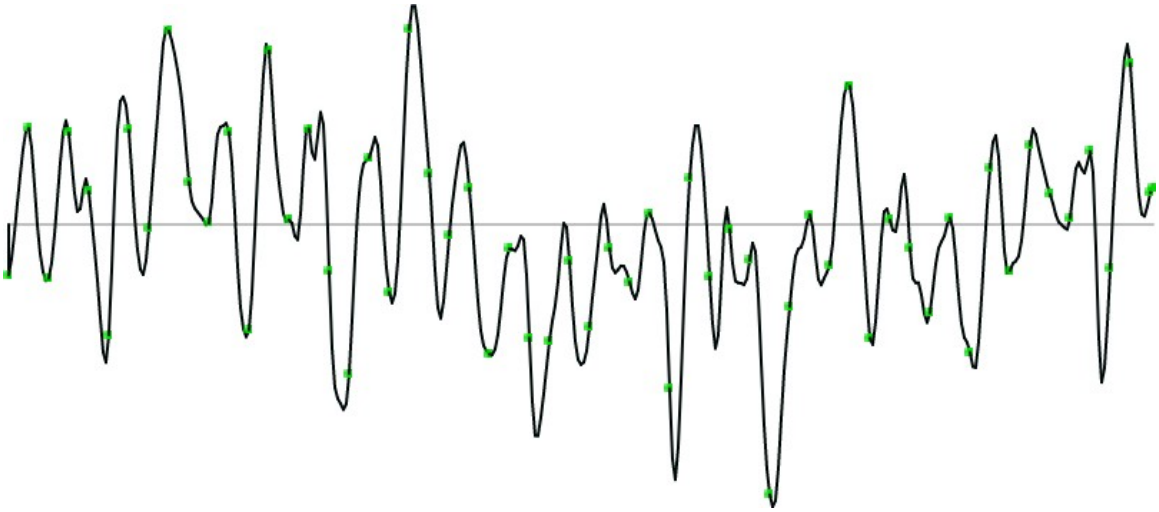
Media Computation with Sounds



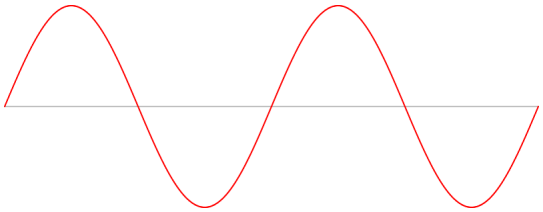
The starting point for computation with sound is the **microphone** block. It starts by recording a brief burst of sound from your microphone. (How brief? On my computer, 0.010667 seconds, but you'll see shortly how to find out or control the sample size on your computer.)

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

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 p instant. (You can skip the next page or so if you know about Fourier analysis.) Here's a picture



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



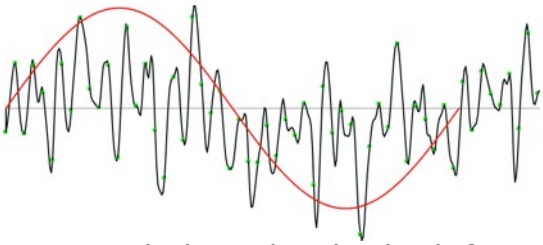
Every periodic function (more or less, any sample that sounds like music rather than sounding li
composed of a sum of sine waves of different frequencies.

Look back at the graph of our sampled sound. There is a green dot every seven samples. There
magic about the number seven; I tried different values until I found one that looked right. What
is that, for the first few dots at least, they coincide almost perfectly with the high points and low
graph. Near the middle (horizontally) of the graph, the green dots don't seem anywhere near th
points, but if you find the very lowest point of the graph, about 2/3 of the way along, the dots st
almost perfectly again.

The red graph above shows two *cycles* of a sine wave. One cycle goes up, then down, then up a
amount of time taken for one cycle is the *period* of the sine function. If the green dots match be
in the captured sound, then two dots—14 samples, or 14/48000 of a second—represent the peri
cycle and a half of the graph looks like it could be a pure sine wave, but after that, the tops and
line up, and there are peculiar little jiggles, such as the one before the fifth green dot. This hap
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 a
per second. Another name for "cycles per second" is "Hertz," abbreviated Hz, so our sound has
3249 Hz. As a musical note, that's about an A (a little flat), four octaves above middle C. (Don'
much about the note being a little off; remember that the 14-sample period was just eyeballed a
be exactly right.)

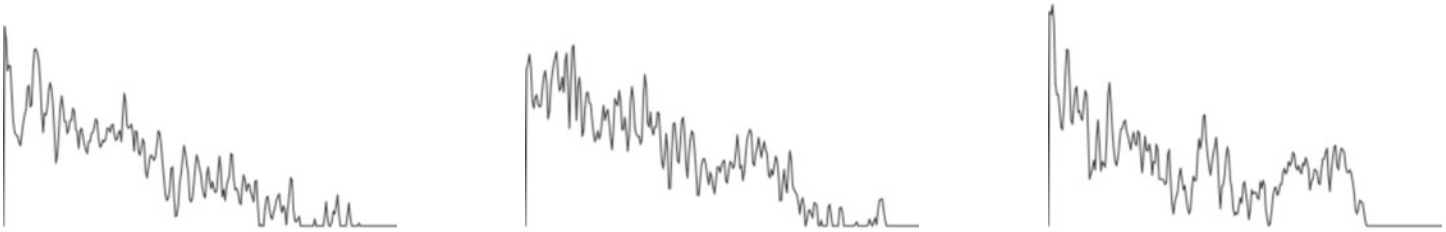
Four octaves above middle C is really high! That would be a shrill-sounding note. But remembe
complex waveform is the sum of multiple sine waves at different frequency. Here's a different u
regularity:



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

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

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



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

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

Samples and **spectrum** are the two most detailed representations of a sound. But the **micro**
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** 

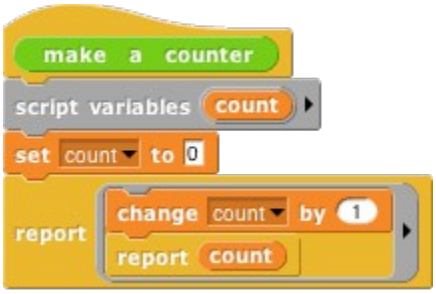
Its first input is a list of samples, and its second input specifies how many samples occupy one s

VIII. OOP with Procedures

The idea of object oriented programming is often taught in a way that makes it seem as if a sp
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 procedur*
message as input and reports the corresponding method. In this section we start with a strippe
to show how local state works, and build up to full implementations of class/instance and protot

A. Local State with Script Variables



This script implements an object *class*, a type of object, namely the counter class. In this first s
there is only one method, so no explicit message passing is necessary. When the **make a cou**
it reports a procedure, the ringed script inside its body. That procedure implements a specific c
instance of the counter class. When invoked, a counter instance increases and reports its count
counter has its own local count:

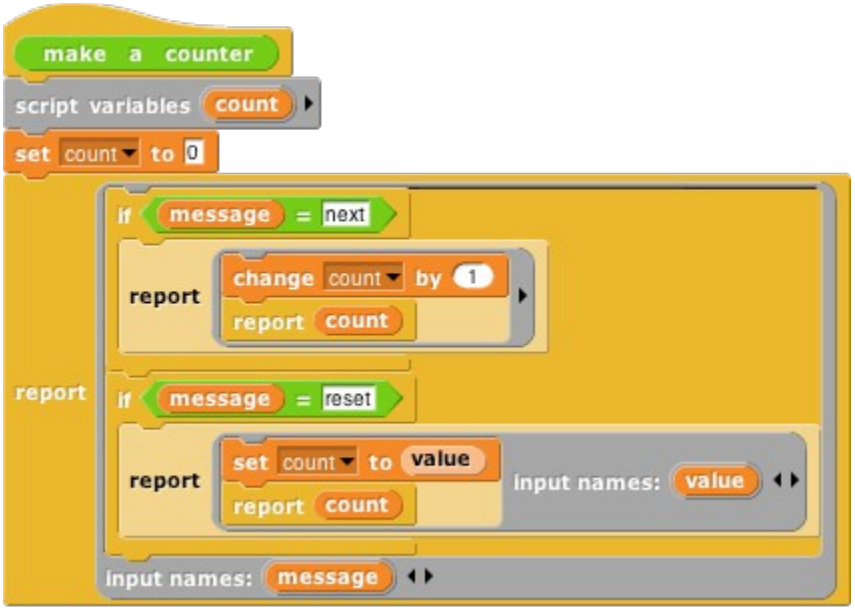
This example will repay careful study, because it isn't obvious why each instance has a separa
the point of view of the **make counter** procedure, each invocation causes a new **count** variable to b
Usually such *script variables* are temporary, going out of existence when the script ends. But th
because **make counter** returns *another script* that makes reference to the **count** variable, so it re

(The **script variables** block makes variables local to a script. It can be used in a sprite's script ar 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 pro**a counter**. The **r** block represents the class, while each instance is represented by a nameless script **c**reated eac **counter** is called. The script variables created inside the **make a counter** block but outside the ring are *insta 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 jus to carry out its one method. Here is a more refined version that uses message passing:



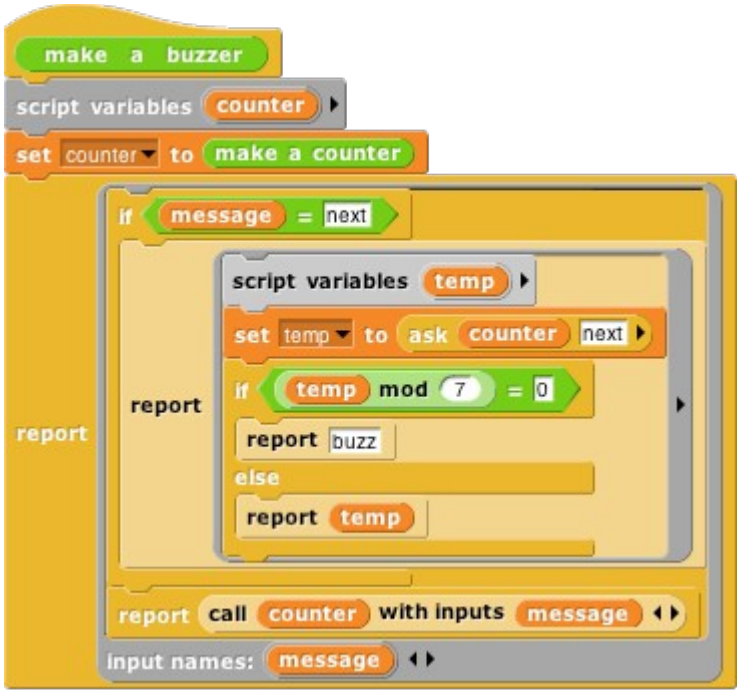
Again, the **make a counter** block represents the **counter** class, and again the script creates a local va each time it is invoked. The large outer ring represents an instance. It is a *dispatch procedure:* (just a text word) as input, and it reports a method. The two smaller rings are the methods. Th **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 instanc a method, which must then be called to finish the job. We can provide a block to do both proced

The **ask** block has two required inputs: an object and a message. It also accepts optional additi which Snap! puts in a list; that list is named **args** inside the block. **Ask** has two nested **call** blo calls the object, i.e., the dispatch procedure. The dispatch procedure always takes exactly one message. It reports a method, which may take any number of inputs; note that this is the situa drop a list of values onto the arrowheads of a multiple input (in the outer **call** block). Note also of the rare cases in which we must unringify the inner **call** block, whose *value when called* give

C. Inheritance via Delegation

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

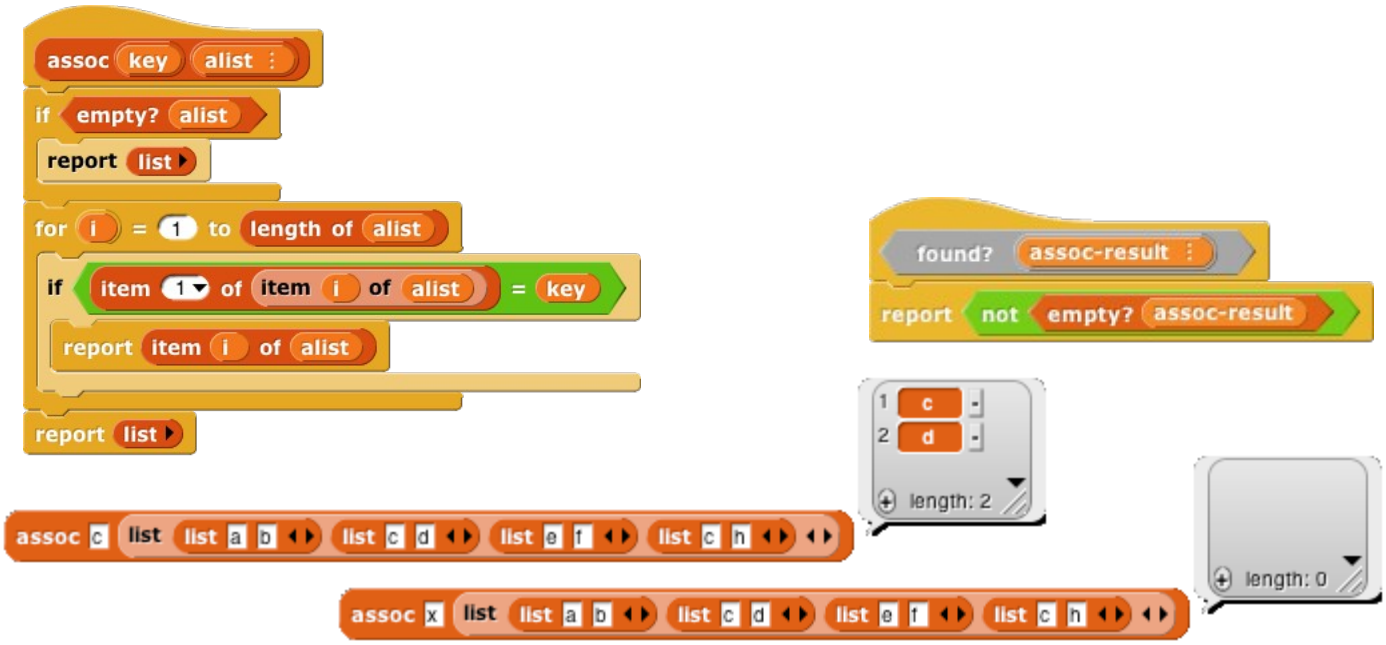


This script implements the **buzzer** class, which is a child of **counter**. Instead of having a cour local state variable, each buzzer has a **counter** (an object) as a local state variable. The class s method, reporting what the counter reports unless that result is divisible by 7, in which case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

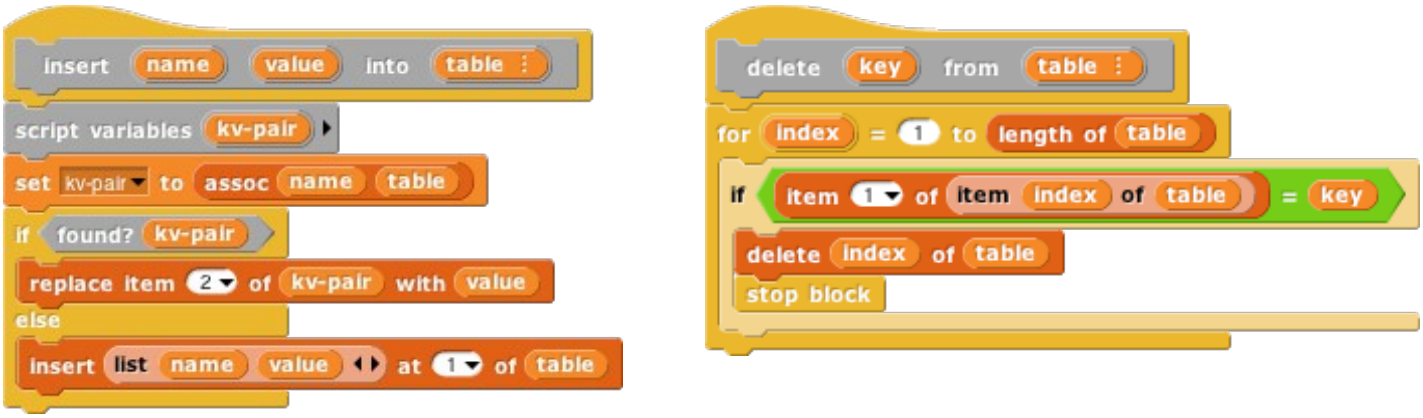
D. An Implementation of Prototyping OOP

In the class/instance system above, it is necessary to design the complete behavior of a class b
make any instances of the class. This is okay for top-down design, but not great for experiment
sketch the implementation of a *prototyping* OOP system: You make an object, tinker with it, ma
and keep tinkering. Any changes you make in the parent are inherited by its children. In effect
is both the class and an instance of the class. In the implementation below, children share prop
and local variables) of their parent unless and until a child changes a property, at which point th
private copy. (If a child wants to change something for its entire family, it must ask the parent

Because we want to be able to create and delete properties dynamically, we won't use Snap! v
an object's variables or methods. Instead, each object has two *tables,* called **methods** and **dat**
an *association list:* a list of two-item lists, in which each of the latter contains a *key* and a corres
provide a lookup procedure to locate the key-value pair corresponding to a given key in a given



There are also commands to insert and delete entries:

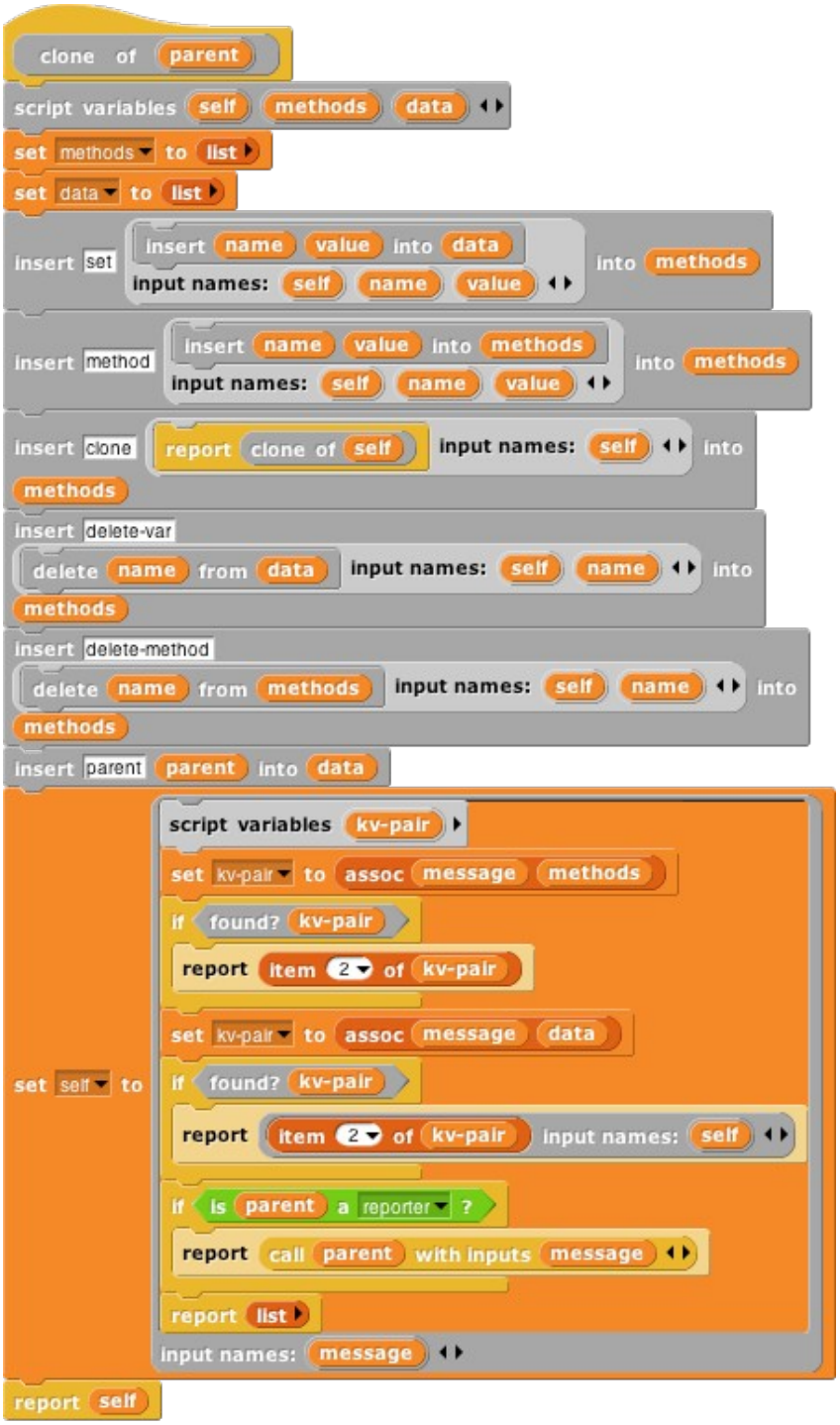


As in the class/instance version, an object is represented as a dispatch procedure that takes a
input and reports the corresponding method. When an object gets a message, it looks for that keyword
in its **methods** table. If it's found, the corresponding value is the method we want. If not, the
data table. If a value is found there, what the object returns is *not* that value, but rather a repo
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 pare
parent (that is, the parent's dispatch procedure) is invoked with the message as its input. Even
match is found, or an object with no parent is found; the latter case is an error, meaning that th
the object a message not in its repertoire.

Messages can take any number of inputs, as in the class/instance system, but in the prototypir
method automatically gets the object to which the message was originally sent as an extra first
do this so that if a method is found in the parent (or grandparent, etc.) of the original recipient,
refers to a variable or method, it will use the child's variable or method if the child has its own v

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



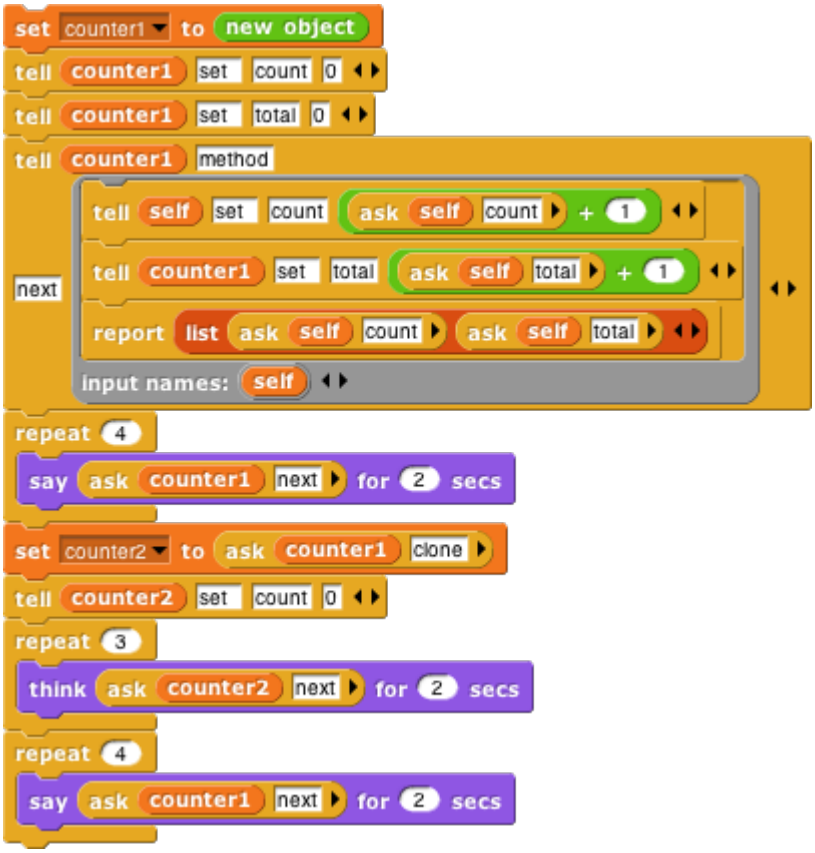
Every object is created with predefined methods for **set**, **method**, **delete-var**, **delete-metho** one predefined variable, **parent**. Objects without a parent are created by calling **new object**:

As before, we provide procedures to call an object's dispatch procedure and then call the meth version, we provide the desired object as the first 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 in over the arrowheads rather than over the input slot.)

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

[11] [22] [33] [44] (15) (26) (37) [58] [69] [710] [811]



IX. The Outside World

The facilities discussed so far are fine for projects that take place entirely on your computer's so may want to write programs that interact with physical devices (sensors or robots) or with the W 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 repo of the Web server's response (minus HTTP header), *without interpretation.* This means that in r response is a description of the page in HTML (HyperText Markup Language) notation. Often, es commercial web sites, the actual information you're trying to find on the page is actually at ano included in the reported HTML. The Web page is typically a very long text string, and so the pri 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 line 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 downarrow near the bottom right corner of the speech balloon in the picture presents a menu o ranges. (This may seem unnecessary, since the scroll bar should allow for any number of items 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 p
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 w
There is an official workaround for this limitation called the CORS protocol (Cross-Origin Resourc
but the target site has to allow `snap.berkeley.edu` explicitly, and of course most don't. To get around t
problem, you can use third-party sites ("cors proxies") that are not limited by JavaScript and tha
requests.

B. Hardware Devices

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

The idea is that you run a separate program that both interfaces with the device and provides
server that Snap! can use to make requests to the device. *Unlike* Snap! *itself, these programs l*
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 Pa
robots; the Nintendo Wiimote and Leap Motion sensors, the Arduino microcomputer, and Super-
Sylvia's Water Color Bot. The same server technique can be used for access to third party softw
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 repositor
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
reports one component of the date or time, so you will probably combine several calls, like this:

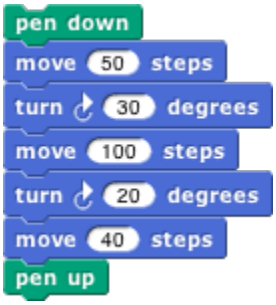
for Americans, or like this:

for Europeans.

X. Continuations

Blocks are usually used within a script. The *continuation* of a block within a particular script is t computation that remains to be completed after the block does its job. A continuation can be re ringed script. Continuations are always part of the interpretation of any program in any langua these continuations are implicit in the data structures of the language interpreter or compiler. I continuations explicit is an advanced but versatile programming technique that allows users to 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 af 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 10** following script?

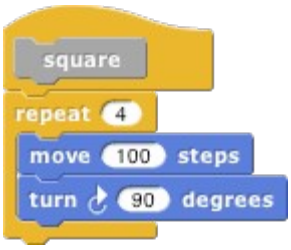
That's a trick question; the **move** block is run four times, and it has a different continuation eac time, its continuation is

Note that there is no **repeat 3** block in the actual script, but the continuation has to represent t 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 done.

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

When a block is used inside a custom block, its continuation may include parts of more than one 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**. Nev 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. Bu continuation of a *reporter* block has to do something with the value reported by the block, so it 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 a empty-slot notation for input substitution.

A. Continuation Passing Style

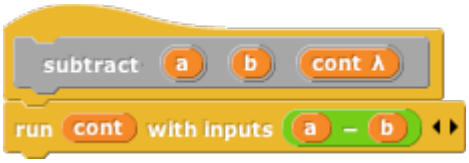
Like all programming languages, Snap! evaluates compositions of nested reporters from the in example, in the expre 

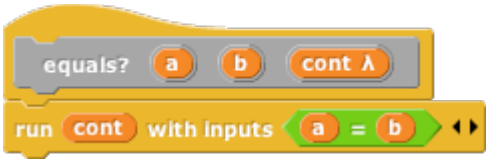
Snap! first adds 4 and 5, then multiplies 3 by th means that the order in which the operations are done is backwards from the order in which the expression: When reading the above expression you say "times" before you say "plus." In Eng saying "three times four plus five," which actually makes the order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

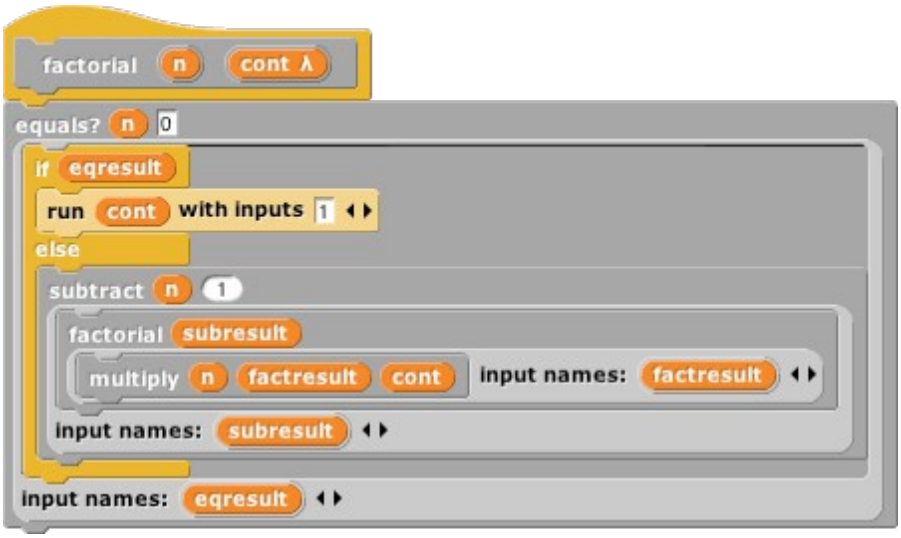
That may seem like overkill in a simple expression, but suppose you're trying to convey the ex

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

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

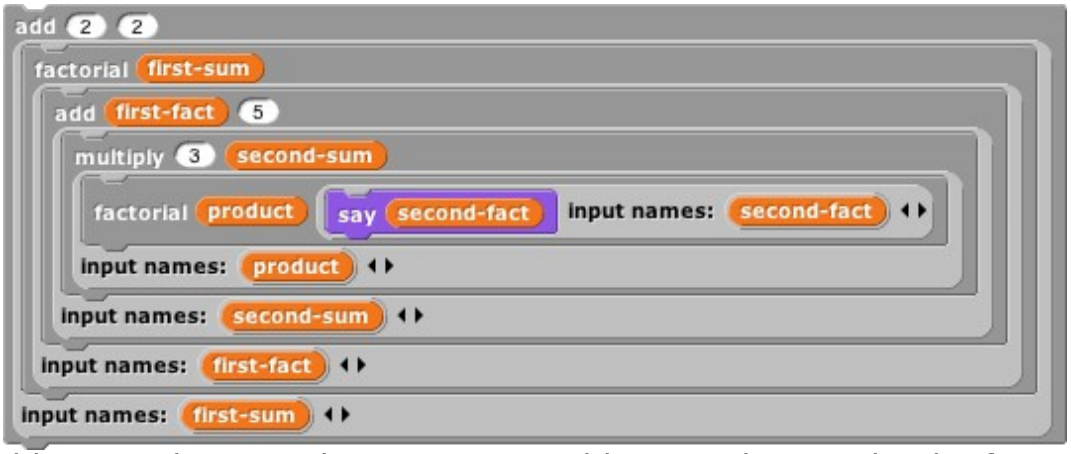






We can check that these blocks give the results we

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 fiv multiply three by that, and take the factorial of the result"? Just what we wanted! This way of w which every block is a command that takes a continuation as one of its inputs, is called *continua (CPS).* Okay, it looks horrible, but it has subtle virtues. One of them is that each script is just on (with the rest of the work buried in the continuation given to that one block), so each block does remember what else to do—in the vocabulary of this section, the (implicit) continuation of each Instead of the usual picture of recursion, with a bunch of little people all waiting for each other, happens is that each little person hands off the problem to the next one and goes to the beach, one active little person at a time. In this example, we start with Alfred, an **add** specialist, who c value 4 and then hands off the rest of the problem to Francine, a **factorial** specialist. She com 24, then hands the problem off to Anne, another **add** specialist, who computes 29. And so on, a **say** specialist, says the value 2.107757298379527×10[illegible] a very large number!

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

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

B. Call/Run w/Continuation

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

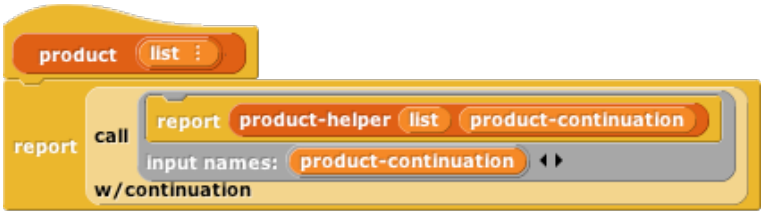
Here's the classic example. We want to write a recursive block that takes a list of numbers as reports the product of all the numbers:

But we can improve the efficiency of this block, in the case of a list that includes a zero; as soo 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 fi the third recursive call (the fourth call altogether), as the first item of the sublist 0,4,5. What is of the **report** 0 block? It's

Even though we already know that **result** is zero, we're going to do three unnecessary multiplic 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** exa
calls that script with *the continuation of the* call-with-continuation *block itself* as its input. In thi
continuation is

reporting to whichever script called **product**. If the input list doesn't include a zero, then nothi
with that continuation, and this version works just like the original **product**. But if the input list
then three recursive calls are made, the zero is seen, and **product-helper** *runs the continuatio*
The continuation immediately reports that 0 to the caller of **product**, *without* unwinding all the
and without the unnecessary multiplications.

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

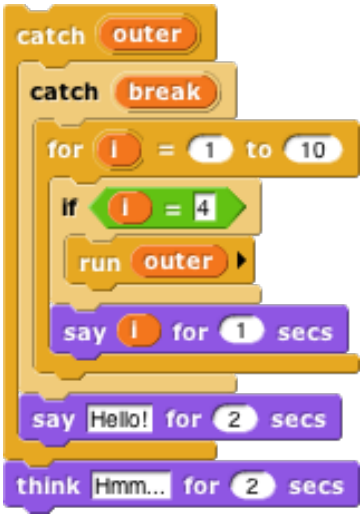
but it's customary to use a script to represent the input to **call w/continuation** because very c
the form
so that the continuation is saved permanently and can be called from anywhere in the project.
input slot in **call w/continuation** has a Command ring rather than a Reporter ring.

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

Nonlocal exit

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

The upvar **break** has as its value a continuation that can be called from anywhere in the progra
immediately to whatever comes after the **catch** block in its script. Here's an example with two
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 variable **break** is used instead of **outer**, then the script will say 1, 2, 3, and "Hello!" before

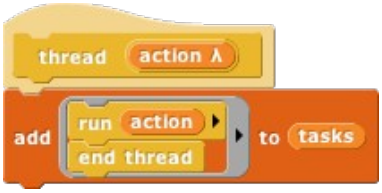
There are corresponding **catch** and **throw** blocks for reporters. The **catch** block is a reporter
expression as input instead of a C-shaped slot. But the **throw** block is a command; it doesn't re
own continuation, but instead reports a value (which it takes as an additional input, in addition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

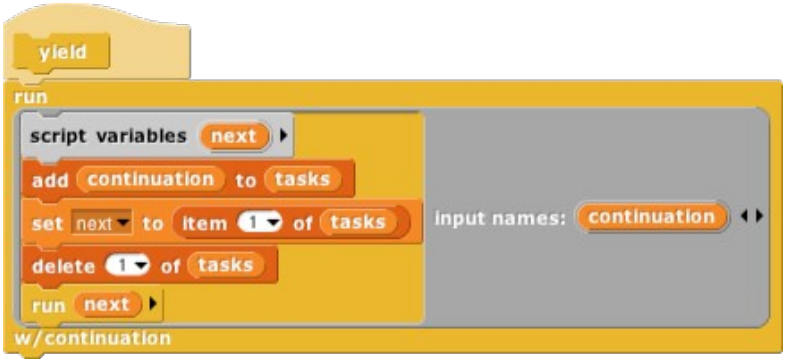
Without the **throw**, the inner **call** reports 5, the **+** block reports 8, so the **catch** block reports 8
reports 80. With the **throw**, the inner **call** doesn't report at all, and neither does the **+** block.
input of 20 becomes the value reported by the **catch** block, and the **×** block multiplies 10 and 2

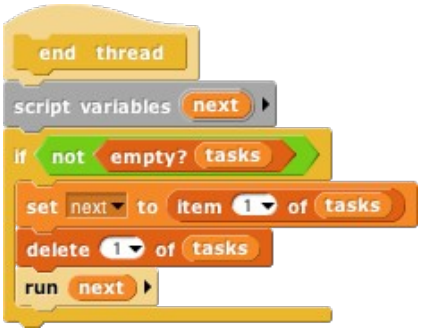
Creating a Thread System

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

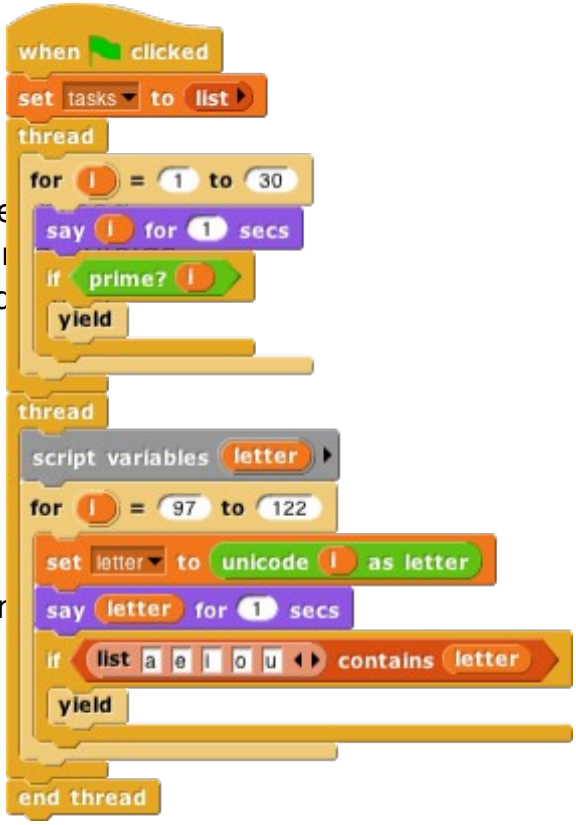
Since this all happens automatically, there is generally no need for the user to think about thre show that this, too, is not magic, here is an implementation of a simple thread system. It uses a named **tasks** that initially contains an empty list. Each use of the C-shaped **thread** block adds (the ringed script) to the list. The **yield** block uses **run w/continuation** to create a continuatio thread, adds it to the task list, and then runs the first waiting task. The **end thread** block (whic 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 **say**s numbers; the other **say**s letters. The number thr after every prime number, while the letter thread yield 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.

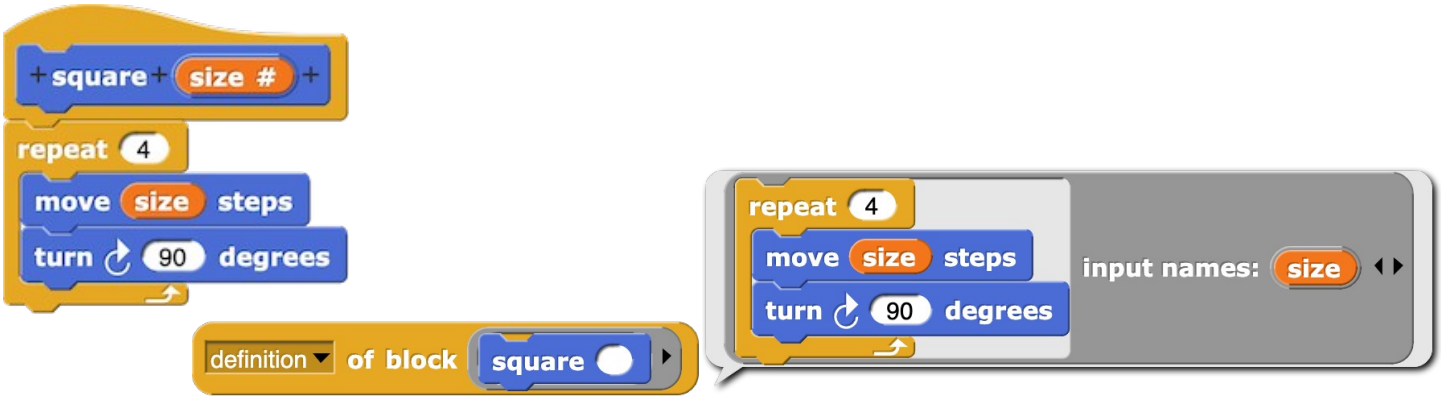


If we wanted this to behave exactly like Snap!'s own we'd define new versions of **repeat** and so on that run each repetition.

XI. Metaprogramming

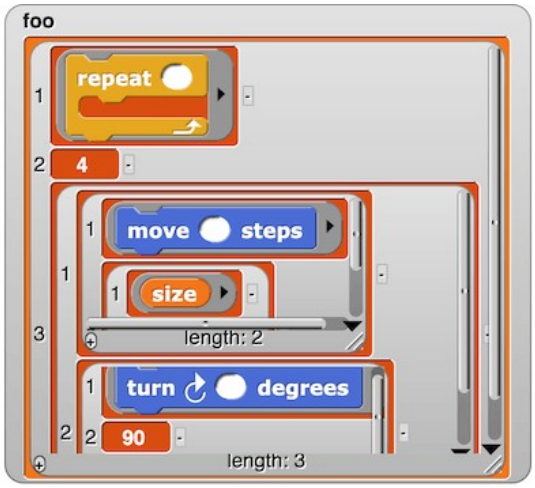
The scripts and custom blocks that make up a program can be examined or created by the pro

A. Reading a block

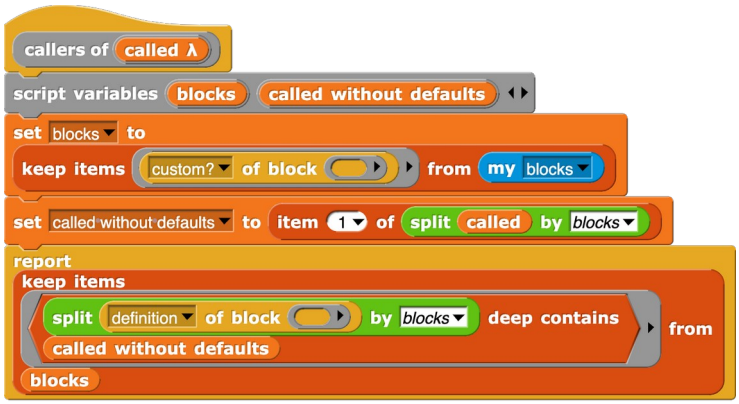


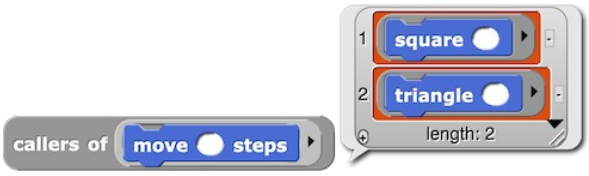
The **definition of** block takes a custom block (in a ring, since it's the block itself that's the inp of calling the block) as input and reports the block's definition, i.e., its inputs and body, in the fo named inputs corresponding to the block's input names, so that those input names are bound in

The **split by blocks** block takes any expression or script as input (ringed) and reports a list re *tree* for the script or expression, in which the first item is a block with no inputs and the remaini input values, which may themselves be syntax trees.



Using **split by blocks** to select custom blocks whose definitions contain another block gives u aid:

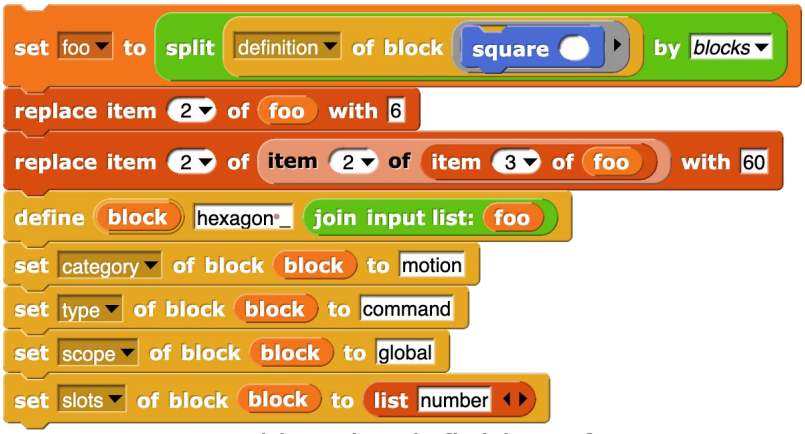


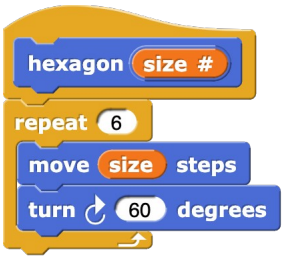


Note in passing the **my blocks** block, which reports a list of all visible blocks, primitive and cust
a **my categories** block, which reports a list of the names of the palette categories.) Also note
which reports True if its input is a custom block.

B. Writing a block

The inverse function to **split by blocks** is provided by the **join** block, which when given a syn
reports the corresponding expression or script.





Here we are taking the definition of **square**, modifying the repetition count (to 6), modifying th
(to 60), using **join** to turn the result back into a ringed definition, and using the **define** block to
hexagon block.

The **define** block has three "input" slots. The quotation marks are there because the first slot
a way for **define** to provide information to its caller, rather than the other way around. In this c
block is the new block itself (the **hexagon** block, in this example). The second slot is where yo
the new block. In this example, the label is "**hexagon** _" in which the underscore represents an
here are a few examples of block labels:

set pen _ to _
for _ = _ to _ _
ask _ and wait
_ of _

Note that the underscores are separated from the block text by spaces. Note in the case of the
that the upvar (the **i**) and the C-slot both count as inputs. Note also that the label is not meant
symbol that represents only this block. Fo 

, and 

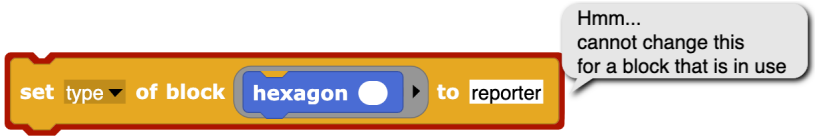
 both have the label
_ of _. The label does not give the input slots names (that's done in the body, coming next) or
in the **set _ of block _ to _** block, coming in two paragraphs).

The third slot is for the *definition* of the new block. This is a (ringed) script whose input names
parameters) will become the formal parameters of the new block. And the script is its script.

So far we know the block's label, parameters, and script. There are other things to specify abo
and one purpose of the **block** upvar is to allow that. In the example on the previous page, ther
set _ of block _ to _ blocks, reproduced below for your convenience:

The **category** of the block can be set to any primitive or custom category. The default is **oth** **command**, **reporter**, or **predicate**. **Command** is the default, so this setting is redundant, bu the choices in the **set** block. The **scope** is either **global** or **sprite**, with **global** as the default. **set slots** is a list of length less than or equal to the number of underscores in the label. Each i type name, like the ones in the **is (5) a (number)?** block. If there is only one input, you can u instead of putting it in a list. An empty or missing list item means type **Any**.

It's very important that these **set** blocks appear in the same script as the **define** that creates the **block** upvar is local to that script. You can't later say, for example,



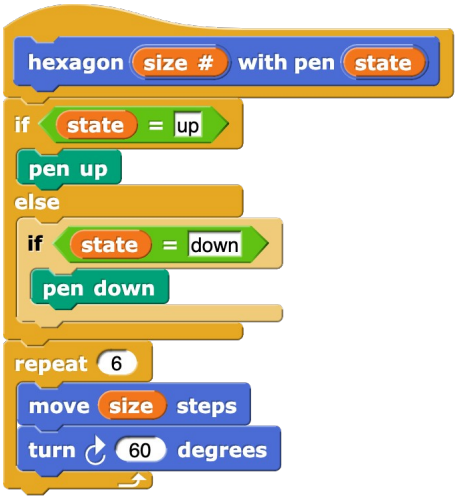
because the copy of the **hexagon** block in this instruction counts as "using" it.

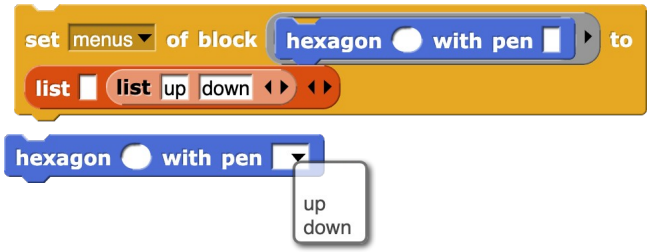
The **of block** reporter is useful to copy attributes from one block to another, as we copied the **square**, modified it, and used it to define **hexagon**. Some of the values this block reports are

"1"? Yes, this block reports *numbers* instead of names for **category**, **type**, and **scope**. The re someday we'll have translations to other languages for custom category names, as we already categories, types, and scopes; if you translate a program using this block to another language, outputs won't change, simplifying comparisons in your code. The **set** block accepts these num alternative to the names.

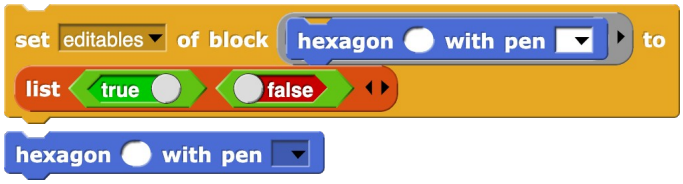
There are a few more attributes of a block, less commonly used.

The list input is just like the one for **set slots** except for default values instead of types. Now f menu input:



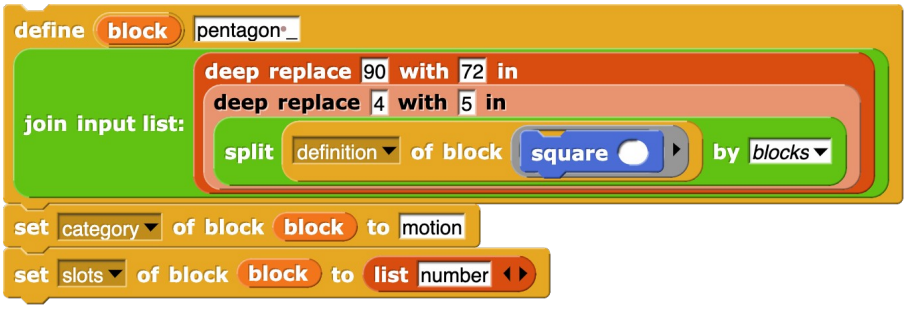


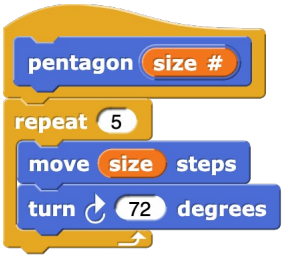
Prefer a read-only menu?



We passed too quickly over how the script turned the **square** block into a **hexagon** block:

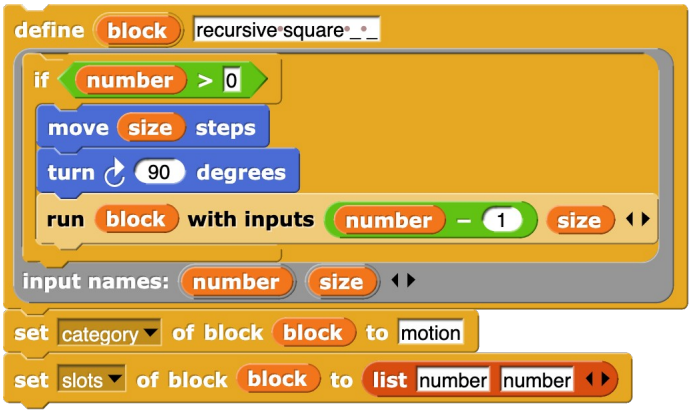
Those **replace item** blocks aren't very elegant. I had to look at **foo** by hand to figure out wher wanted to change are. This situation can be improved with a little programming:

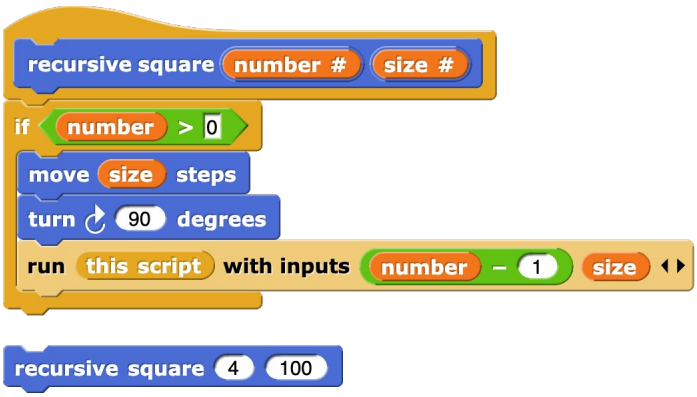




Exercise for the reader: Implement this.

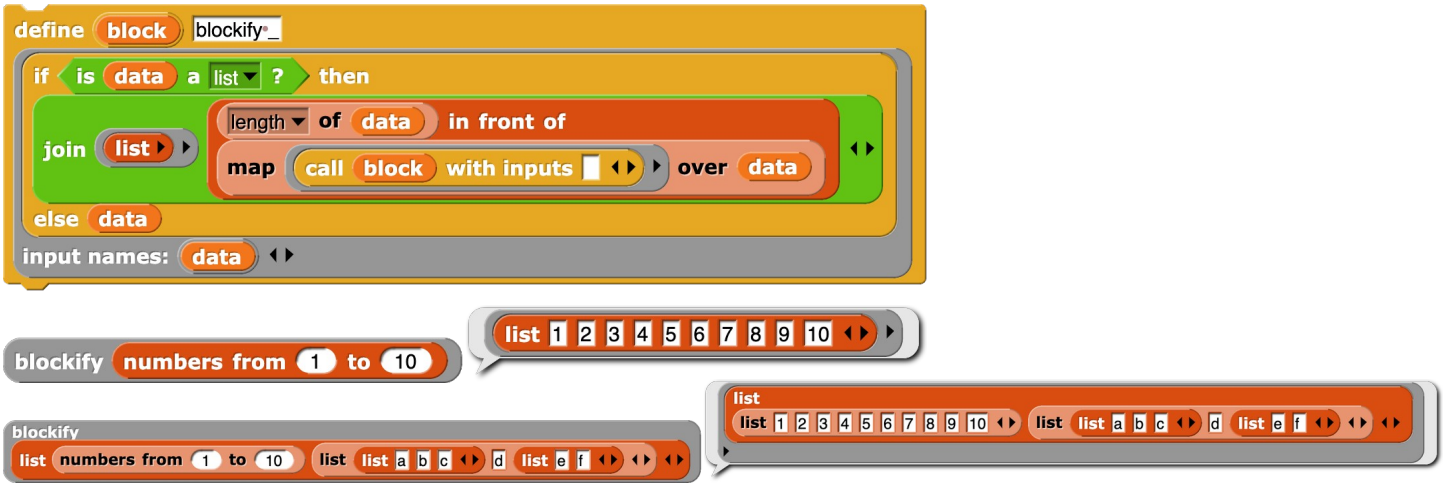
Returning to the **define** block, there's another reason for the **block** upvar: It's helpful in defin procedure using **define**. For a procedure to call itself, it needs a name for itself. But in the def the **define** block, **define** itself hasn't been called yet, so the new block isn't in the palette yet.





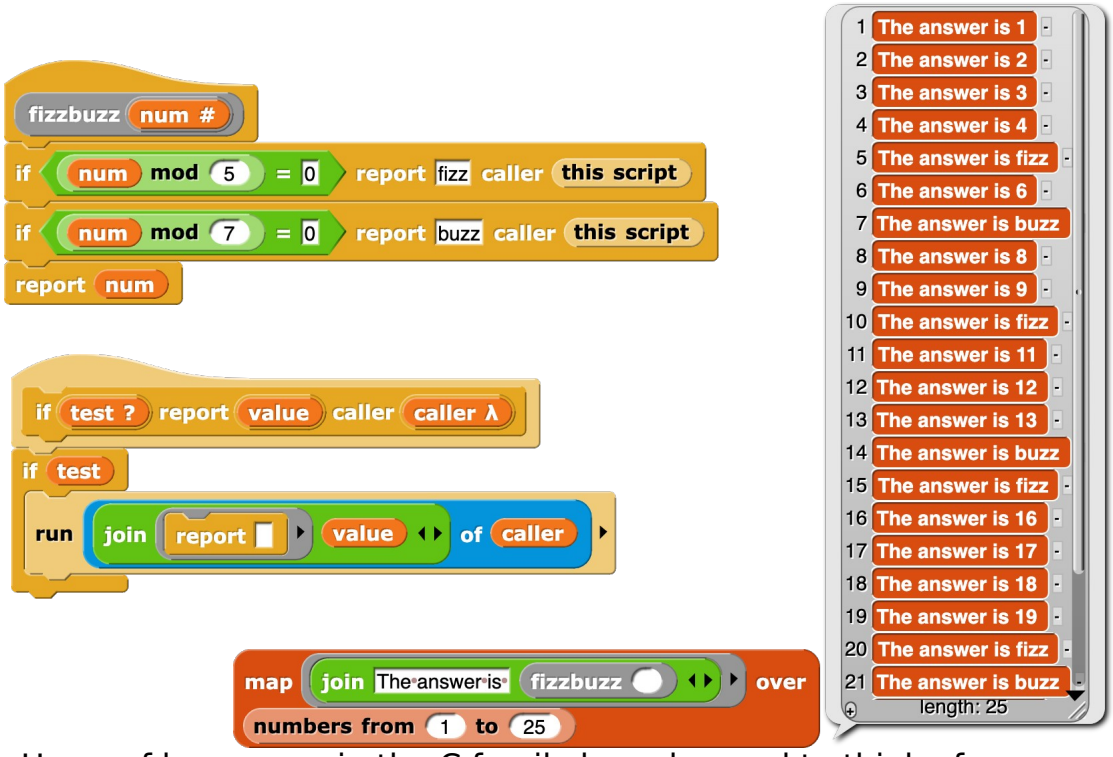
Yes, you put **block** in the **define**, but it gets changed into **this script** in the resulting definition **this script** directly in a simple case like this, but in a complicated case with a recursive call ins

the one giving the block definition, **this script** always means the innermost ring. But the upva
ring:



It’s analogous to using explicit formal parameters when you nest calls to higher order functions.

C. Macros



Users of languages in the C family have learned to think of macros as entirely about text string
relation to the syntax of the language. So you can do things like

```
#define foo baz)
```

with the result that you can only use the macro after an open parenthesis.

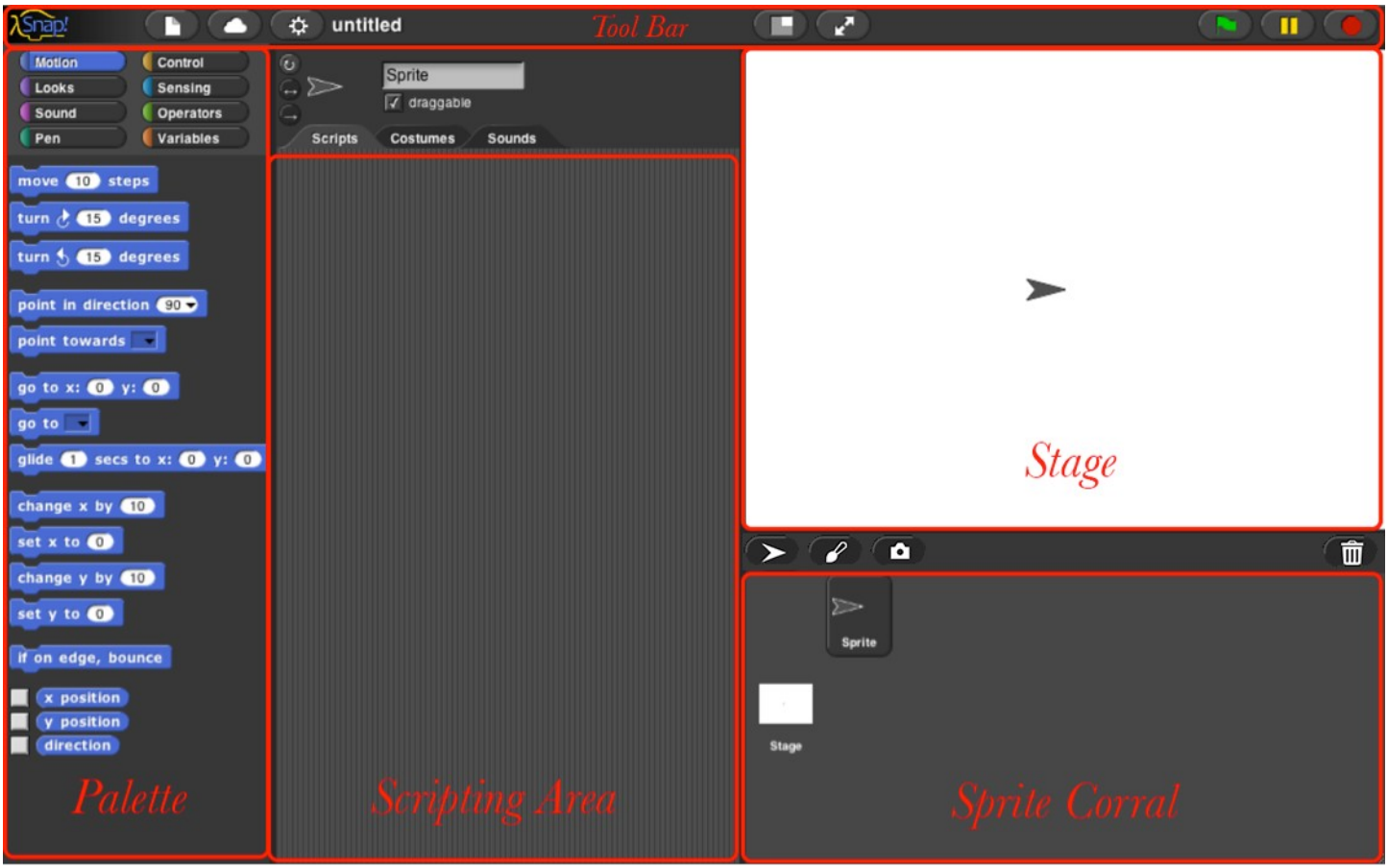
In the Lisp family of languages we have a different tradition, in which macros are syntactically
procedure calls, except that the “procedure” is a macro, with different evaluation rules from ord
procedures. Two things make a macro different: its input expressions are not evaluated, so a m
establish its own syntax (but still delimited by parentheses, in Lisp, or still one block, in Snap!);
macro call is a new expression that is evaluated *as if it appeared in the caller* of the macro, with
variables and, implicitly, its continuation.

Snap! has long had the first part of this, the ability to make inputs unevaluated. In version 8.0 ability to run code in the context of another procedure, just as we can run code in the context o using the same mechanism: the **of** block. In the example on the previous page, the **if _ report** a **report** block, but not in its own context; it causes *the* fizzbuzz *block* to report “fizz” or “buzz” (Yes, we know that the rules implemented here are simplified compared to the real game.) It d out of the entire toplevel script; you can see that **map** is able to prepend “The answer is” to ea

This macro capability isn’t fully implemented. First, we shouldn’t have to use the calling script input to the macro. In a later release, this will be fixed; when defining a block you’ll be able to s macro, and it will automatically get its caller’s context as an invisible input. Second, there is a p confusion between the variables of the macro and the variables of its caller. (What if the macro to a variable **value** in its caller?) The one substantial feature of Scheme that we don’t yet impl *macros,* which make it possible to keep the two namespaces separate.

XII. User Interface Elements

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

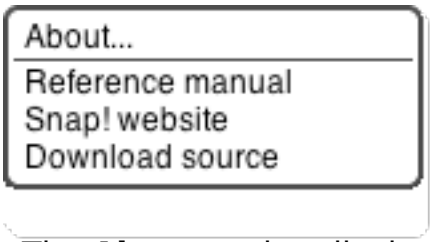


A. Tool Bar Features

Holding down the Shift key while clicking on any of the menu buttons gives access to an extend options, shown in red, that are experimental or for use by the developers. We're not listing thos 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 Sna



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

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

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 Snap!. At the bottom of the page are links to download the latest official release. Or you can n the site to find the current development version. You can read the code to learn how Snap! is i a copy on your own computer (this is one way to keep working while on an airplane), or make a version with customized features. (However, access to cloud accounts is limited to the official v Berkeley.)

The File Menu

The file icon shows a menu mostly about saving and loading projects. You may not see all th you don't have multiple sprites, scenes, custom blocks, and custom categories.

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

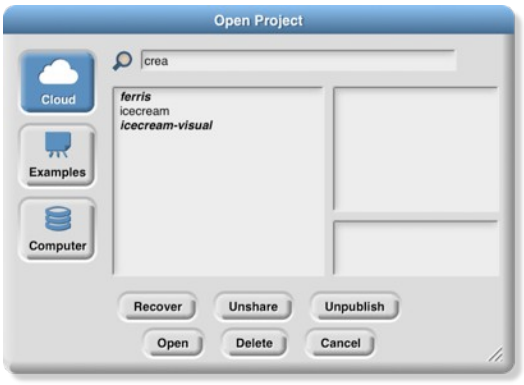
The **New** option starts a new, empty project. Any project you were working on before disappea asked to confirm that this is really what you want. (It disappears only from the current working you should save the current project, if you want to keep it, before using **New**.)

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

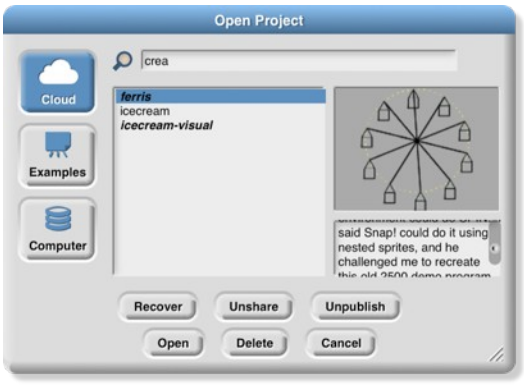
The **Open...** option shows a project open dialog box in which you can choose a project to open

In this dialog, the three large buttons at the left select a source of projects: **Cloud** means your S cloud storage. **Examples** means a collection of sample projects we provide. **Computer** is for p your own computer; when you click it, this dialog is replaced with your computer's system dialo files. The text box to the right of those buttons is an alphabetical listing of projects from that so project by clicking shows its thumbnail (a picture of the stage when it was saved) and its projec right.

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



I was looking for my ice cream projects and typed “crea” in the search bar, then wondered why 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 ro in your cloud account for older versions of the chosen project. ***If your project is damaged, d saving broken versions! Use*** Recover ***first thing.*** You will see a list of saved versions; cho it. Typically, you’ll see the most recent version before the last save, and the newest version sa Then come buttons **Share**/**Unshare** and **Publish**/**Unpublish**. The labelling of the buttons dep project’s publication status. If a project is neither shared nor published (the ones in lightface ty list), it is private and nobody can see it except you, its owner. If it is shared (**boldface** in the pr when you open it you’ll see a URL like this one:

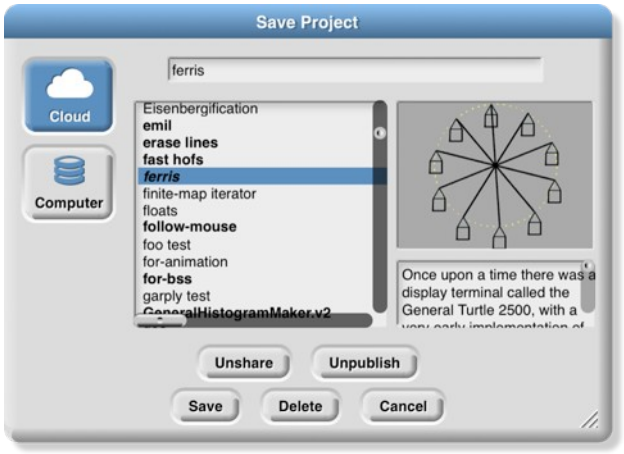
https://snap.berkeley.edu/snapsource/snap.html#present:Username=bh&ProjectNam

but with your username and project name. (“**%20**” in the project name represents a space, wh of a URL.) Anyone who knows this URL can see your project. Finally, if your project is published in the list), then your project is shown on the Snap! web site for all the world to see. (In all of t the only one who can *write* to (save) your project. If another user saves it, a separate copy will user’s account. Projects remember the history of who created the original version and any othe versions along the way.

In the second row, the first button, **Open**, loads the project into Snap! and closes the dialog b button (if **Cloud** is the source) is **Delete**, and if clicked it deletes the selected project. Finally, t closes the dialog box without opening a project. (It does not undo any sharing, unsharing, or de done.)

Back to the File menu, the **Save** menu option saves the project to the same source and same used when opening the project. (If you opened another user's shared project or an example pro 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 pr



This is much like the **Open** dialog, except for the horizontal text box at the top, into which you t the project. You can also publish, unpublish, share, unshare, and delete projects from here. Th **Recover** button.

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

Depending on your browser, the **Export project…** option either directly saves to your disk or browser tab containing your complete project in XML notation (a plain text format). You can 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 alter project to your cloud account: keeping it on your own computer. It is equivalent to choosing **Co** the Save dialog described earlier.

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

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

The **Unused blocks…** option presents a listing of all the global custom blocks in your project t anywhere, and offers to delete them. As with Export blocks, you can choose a subset to delete This option is shown only if you have defined custom blocks.

The **Hide blocks…** option shows *all* blocks, including primitives, with checkboxes. This option remove any blocks from your project, but it does hide selected block in your palette. The purpo is to allow teachers to present students with a simplified Snap! with some features effectively r hiddenness of primitives is saved with each project, so students can load a shared project and s blocks. But users can always unhide blocks by choosing this option and unclicking all the check click in the background of the dialog box to get a menu from which you can check all boxes or u boxes.)

The **New category…** option allows you to add your own categories to the palette. It opens a which you specify a name *and a color* for the category. (A lighter version of the same color will zebra coloring feature.)

The **Remove a category…** option appears only if you've created custom categories. I easy-to-miss menu of category names just under the file icon in the menu bar. If you remove a blocks in it, all those blocks are also removed.

The next group of options concern the *scenes* feature, new in Snap! 7.0. A scene is a complete own stage, sprites, and code, but several can be merged into one 

block to bring another scene onscreen. The **Scenes…** option presents a menu of all the scenes in your the File menu was before you clicked it. The **New scene** option creates a new, empty scene, w rename as you like from its context menu. **Add scene…** is like **Import…** but for scenes. (A co can be imported as a scene into another project, so you have to specify that you're importing th *scene* rather than replacing the current project.)

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

The library menu is divided into five broad categories. The fir is, broadly, utilities: blocks that might well be primitives. The might be useful in all kinds of projects.

The second category is blocks related to media computation: ones that help in dealing with costumes and sounds (a/k/a Je libraries). There is some overlap with "big data" libraries, for dealing with large lists of lists.

The third category is, roughly, specific to non-media applicati (a/k/a Brian libraries). Three of them are imports from other programming languages: words and sentences from Logo, arr functions from APL, and streams from Scheme. Most of the others are to meet the needs of the BJC curriculum.

The fourth category is major packages provided by users.

The fifth category provides support for hardware devices such robots, through general interfaces, replacing specific hardwar libraries in versions before 7.0.

When you click on the one-line description of a library, you are shown the actual blocks in the li
longer explanation of its purpose. You can browse the libraries to find one that will satisfy your
libraries are described in detail in Section I.H, page 25.

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
import more than one costume by double-clicking each one, and then clicking Cancel when don
some costumes are tagged with “svg” in this picture; those are vector-format costumes that are
within Snap!.

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

The costume and background libraries include both bitmap (go jagged if enlarged) and vector
smoothly) images. Thanks to Scratch 2.0/3.0 for most of these images! Some older browsers r
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.

Finally, the **Undelete sprites...** option appears only if you have deleted a sprite; it allows you sprite that was deleted by accident (perhaps intending to delete only a costume).

The Cloud Menu

The cloud ico shows a menu of options relating to your Snap! cloud account. If you are n in, you see the outline and get this menu:

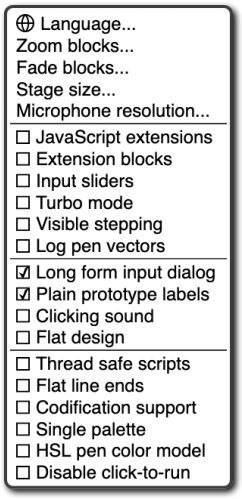
Choose **Login...** if you have a Snap! account and remember your password. Choose **Signup...** an account. Choose **Reset Password...** if you've forgotten your password or just want to chan then get an email, at the address you gave when you created your account, with a new tempora Use that password to log in, then you can choose your own password, as shown below. Choose **Verification Email...** if you have just created a Snap! account but can't find the email we sent verify that it's really your email. (If you still can't find it, check your spam folder. If you are usi address, your school may block incoming email from outside the school.) The **Open in Commu** appears only if you have a project open; it takes you to the community site page about that pro

If you are already logged in, you'll see the cloud in get this menu:

Logout is obvious, but has the additional benefit of showing you who's logged in. **Change pas** for your old password (the temporary one if you're resetting your password) and the new passw entered twice because it doesn't echo. **Open in Community Site** is the same as above.

The Settings Menu

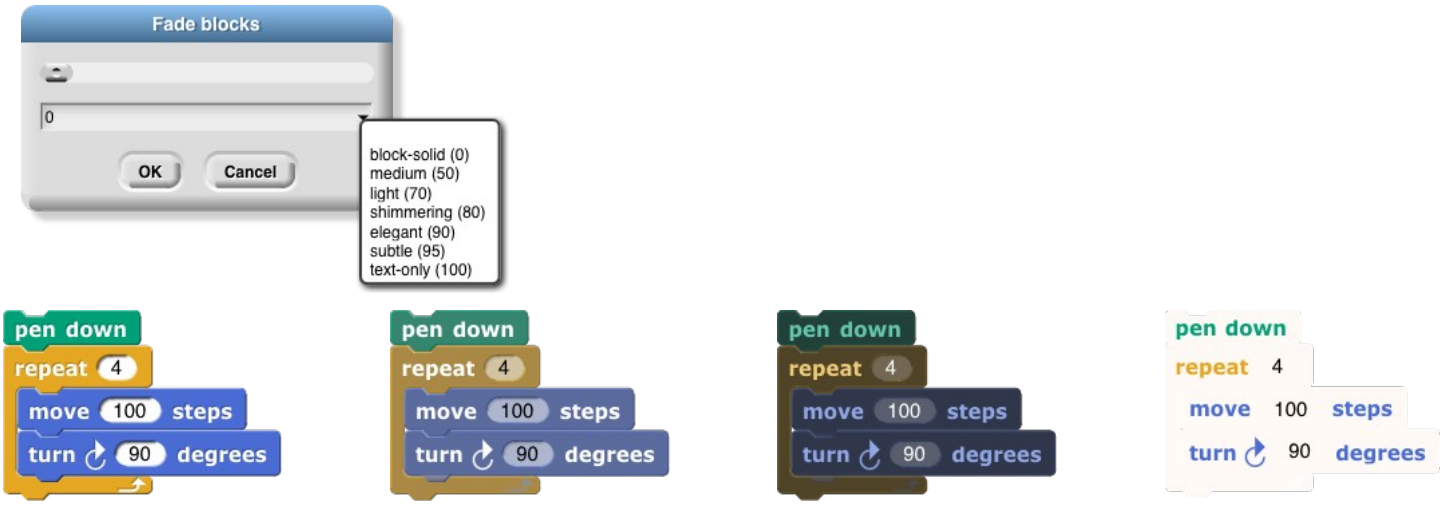
The settings ic shows a menu of Snap! options, either for the current project or for you perm
depending on the option:



The **Language...** option lets you see the Snap! user interface (blocks and messages) in a langu
English. (Note: Translations have been provided by Snap! users. If your native language is mis
email!)

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

The **Fade blocks...** option opens a dialog in which you can change the appearance of blocks:



Mostly this is a propaganda aid to use on people who think that text languages are somehow be
grown up than block languages, but some people do prefer less saturated block colors. You can
pulldown menu for preselected fadings, use the slider to see the result as you change the fading
a number into the text box once you've determined your favorite value.

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

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

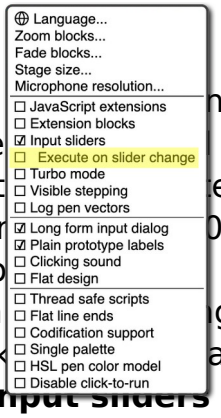
The remaining options let you turn various features on and off. There are three groups of chec
first is for temporary settings not saved in your project nor in your user preferences.

The **JavaScript extensions** option enables the use of the **JavaScript function** block. Becau
projects could use JavaScript to collect private information about you, or to delete or modify you
projects, you must enable JavaScript *each time* you load a project that uses it.

The **Extension blocks** option adds two blocks t 

These blocks provide assorted capabilities to official libraries that were formerly implemented w
JavaScript function block. This allows these libraries to run without requiring the **JavaScript**
option. Details are subject to change.

Input sliders provides an alternate way to put values in numeric input slots; if you click in suc
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 n the
current value. If you want to make a bigger change than that, you can slide the
way to either end, then click on the input slot again, getting a new slider wit er
point. But you won’t want to use this technique to change the input value fr 00,
and it doesn’t work at all for non-integer input ranges. This feature was imp
because software keyboard input on phones and tablets didn’t work at all in g,
and still doesn’t work perfectly on Android devices, so sliders provide a work as
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**
option is also checked, then changing a slider in the scripting area automatically runs the script
input appears. The project **live-tree** in the **Examples** collection shows how this can be used; i
tree custom block with several inputs, and you can see how each input affects the picture by m

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

Visible stepping enables the slowed-down script evaluation described in Chapter I. Checking
equivalent to clicking the footprint button above the scripting area. You don’t want this on exce
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 tha pixels that the drawing leaves on the stage. This remembered vector picture can be used in tw right-clicking on a **pen trails** block gives an option to relabel it into a **pen vectors** block which reports the logged lines as a vector (svg) costume. Second, right-clicking on the stage when th vectors shows an extra option, **svg…**, that exports a picture of the stage in vector format. Only not color regions made with the **fill** block.

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

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

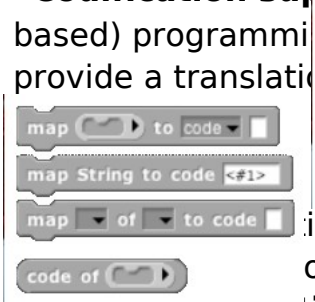
Clicking sound causes a really annoying sound effect whenever one block snaps next to anot Certain very young children, and our colleague Dan Garcia, like this, but if you are such a child bear in mind that driving your parents or teachers crazy will result in you not being allowed to u might, however,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

Flat design changes the "skin" of the Snap! window to a really hideous design with white an background, rectangular rather than rounded buttons, and monochrome blocks (rather than the somewhat 3D-looking normal blocks). The monochrome blocks are the reason for the "flat" in t option. The only thing to be said for this option is that, because of the white background, it ma with the rest of a web page when a Snap! project is run in a frame in a larger page. (I confess the picture of blocks faded all the way to just text two pages ago, though.)

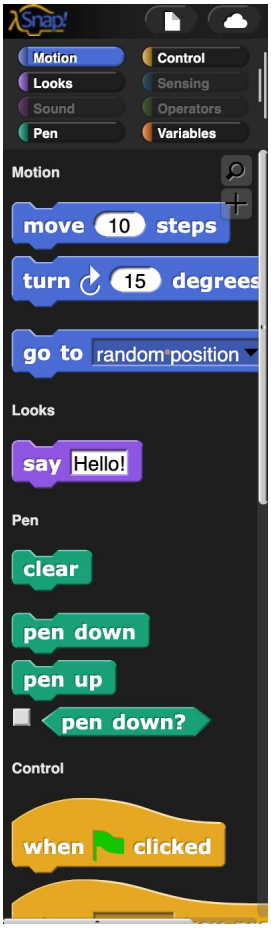
The final group of settings change the way Snap! interprets your program; they are saved wit anyone who runs your project will experience the same behavior. **Thread safe scripts** change responds when an event (clicking the green flag, say) starts a script, and then, while the script i same event happens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

Flat line ends affects the drawing of thick lines (large pen width). Usually the ends are round best when turning corners. With this option selected, the ends are flat. It's useful for drawing a filled rectangle.

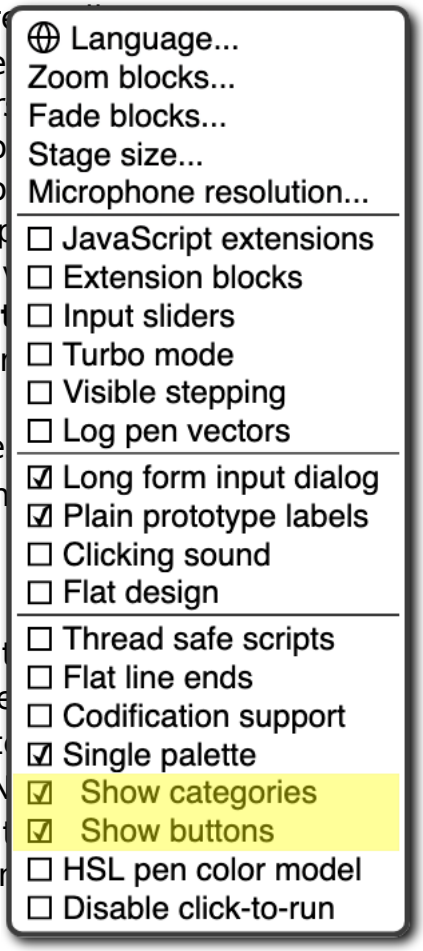
Codification support enables a feature that can translate a Snap! project to a text-based (ra based) programmi feature doesn't know about any particular other language; provide a translati e block using these special blocks:



ive blocks, you can build a block library to translate into any programming lan o be added to our library collection (or contribute one). To see some example project "**Codification**" in the **Examples** project list. Edit the blocks **map to Smalltalk**, **map t** to see examples of how to provide translations for blocks.



The **Single palette** option puts all blocks, r of category, into a single palette. It's intende for use by curriculum developers building *Par problems:* projects in which only a small set o provided, and the task is to arrange those blo achieve a set goal. In that application, this op combined with the hiding of almost all primiti blocks. (See page 119.) When **Single palett** on, two additional options (initially on) appear settings menu; the **Show categories** option the appearance of the palette category name Motion and Looks, while the **Show buttons** option controls the appearance Make a variable and Make a block buttons in the palette.



The **HSL pen color model** option changes t **pen**, **change pen**, and **pen** blocks to provide options **hue**, **saturation**, and **lightness** inst **saturation**, and **brightness** (a/k/a value). N name "saturation" means something different from in HSV! See Appendix A for all the inform you need about colors.

The **Disable click-to-run** option tells Snap! to ignore user mouse clicks on blocks and scripts ordinarily run the block or script. (Right-clicking and dragging still work, and so does clicking in edit it.) This is another Parsons problem feature; the idea is that there will be buttons displayed only in teacher-approved ways. But kids can uncheck the checkbox.

Visible Stepping Controls

After the menu buttons you'll see the project name. After that comes th speed button 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 o
first is the **shrink/grow** button. Normally it lo ik Clicking
the button displays the stage at hal
normal size horizontally and vertically (so it takes up ¼ of its usual area). When the stage is ha
looks like th and clicking it returns the stage to normal size. The main reason you'd want a
stage is during the development process, when you're assembling scripts with wide input expre
normal scripting area isn't wide enough to show the complete script. You'd typically then switc
normal size to try out the project. The next **presentation mode** button nor y looking
Clicking like thi
the button makes the stage double size in both dimensions and eliminates most of the other us
elements (the palette, the scripting area, the sprite corral, and most of the tool bar). When you
project using a link someone has sent you, the project starts in presentation mode. While in pre
the button looks like Clicking it returns to normal (project development) mode.

Project Control Buttons

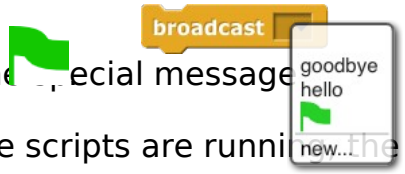
Above the right edge of the stage are three buttons that control the running of the project.

Technically, the **gree g** is no more a project control than anything else that can trigger a ha
typing on the keyboard or clicking on a sprite. But it's a convention that clicking the flag shoul
of the project from the beginning. It's only a convention; some projects have no flag-controlled
respond to keyboard controls instead. Clicking the green flag also deletes temporary clones.

Whenever any script is running (not necessarily in the current sprite), t reen flag is lit:

Shift-clicking the button enters Turbo mode, and the button then looks lik ig Shifting bolt:
clicking again turns Turbo mode off.

Scripts can simulate clicking the green flag by **broadcast**ing the ecial message



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

The **stop** butt stops all scripts, like the **stop all** block. It does *not* prevent a script from st
in response to a click or keystroke; the user interface is always active. There is one exception:
block will not fire after a stop until some non-generic event starts a script. The **stop**
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 categ
Motion, Looks, Sound, Pen, Control, Sensing, Operators, and Variables (which also includes the L
blocks). There are no menus behind these buttons.

Buttons in the Palette

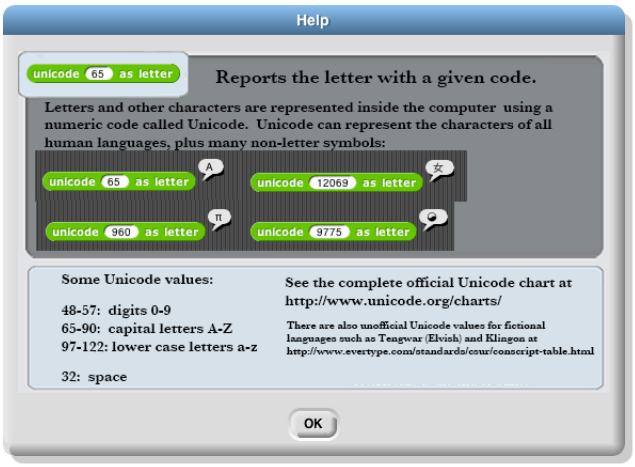
Under the eight palette selector buttons, at the top of the actual palette, are two semi-transpare
first is the *search* button, which is equivalent to typing control-F: It replaces the palette with a s
into which you can type part of the title text of the block you're trying to find. To leave this sea
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 th
has the current palette (color) preselected.

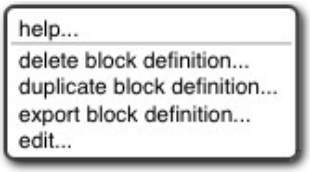
Context Menus for Palette Blocks

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

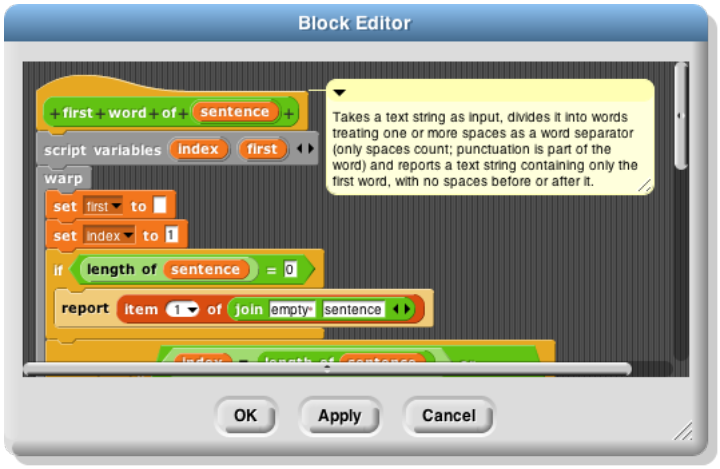
The **help...** option displays a box with documentation about the block. Here's an example:

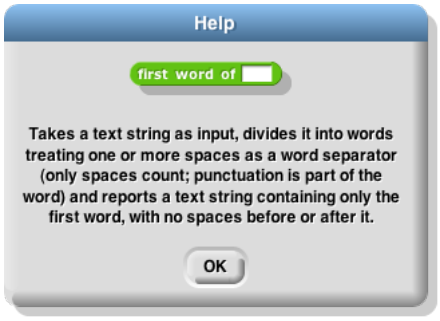


If you control-click/right-click a *custom* (user-defined) block in the palette, you see this menu:



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





If the help text includes a URL, it is clickable and will open the page in a new tab.

The **delete block definition…** option asks for confirmation, then deletes the custom block ar any scripts in which it appears. (The result of this removal may not leave a sensible script; it's l correct such scripts *before* deleting a block.) Note that there is no option to *hide* a custom bloc in the Block Editor by right-clicking on the hat block.

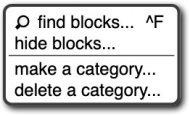
The **duplicate block definition…** option makes a *copy* of the block and opens that copy in th Since you can't have two custom blocks with the same title text and input types, the copy is cre (or a higher number if necessary) at the end of the block prototype.

The **export block definition…** option writes a file in your browser's downloads directory cont definition of this block and any other custom blocks that this block invokes, directly or indirectly resulting file can be loaded later without the risk of red **Undefined!** blocks because of missing

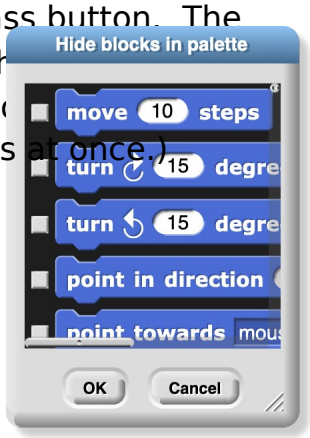
The **edit…** option opens a Block Editor with the definition of the custom block.

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 blocks…** option opens a dialog box in which you can choose wh (custom as well as primitive) should be hidden. (Within that dialog bo menu of the background allows you to check or uncheck all the boxes at once.)

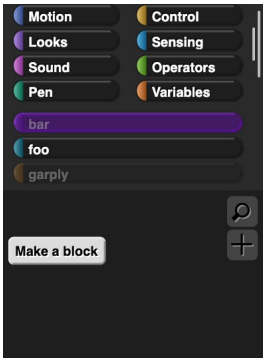


The **make a category…** option, which is intended mainly for authors of snap extensions, lets y
categories to the palette. It opens a small dialog window in which you specify a name *and a co*
category:

Pick a dark color, because it will be lightened for zebra coloring when users nest blocks of the sa
Custom categories are shown below the built-in categories in the category selector:

This example comes from Eckart Modrow's SciSnap*!* library. Note that the custom category list
scroll bar, which appears if you have more than six custom categories. Note also that the butto
custom category occupy the full width of the palette area, unlike the built-in categories, which c
of the width. Custom categories are listed in alphabetical order; this is why Prof. Modrow chose
category name with a number, so that he could control their order.

If there are no blocks visible in a category, the category name is dimmed in the category selec



Here we see that category foo has blocks in it, but categories bar and garply are empty. The bu
are also subject to dimming, if all of the blocks of a category are hidden.

Palette Resizing

At the right end of the palette area, just to the left of the scripting area, is a r
handle that can be dragged rightward to increase the width of the palette are
useful if you write custom blocks with very long names. You can't reduce the
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
the appearance and behavior of a sprite. There is always a *current sprite,* whose scripts are sho
area. A dark grey rounded rectangle in the sprite corral shows which sprite (or the stage) is cur
it's only the visible *display* of the scripting area that is "current" for a sprite; all scripts of all spr
running at the same time. Clicking on a sprite thumbnail in the sprite corral makes it current. T
can be selected as current, in which case the appearance is different, with some primitives not s

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 contr

1. The three circular buttons in a column at the left control the sprite's *rotation* behavior. Sprite
designed to be right-side-up when the sprite is facing toward the right (direction = 90). If the to
lit, the default as shown in the picture above, then the sprite's costume rotates as the sprite cha
the middle button is selected, then the costume is reversed left-right when the sprite's direction
leftward (direction between 180 and 359, or equivalently, between -180 and -1). If the bottom b
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 stag
dragging it. The common use of this feature is in game projects, in which some sprites are mea
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
clicking on the script runs it even if the event in the hat block doesn't happen. (This is a useful

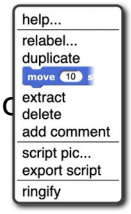
technique when you have a dozen sprites and they each have five scripts with green-flag hat bl
want to know what a single one of those scripts does.) The script will have a green "halo" arou
running. If the script is shared with clones, then while it has the green halo it will also have a c
many instances of the script are running. Clicking a script with such a halo *stops* the script. (If
a **warp** block, which might be inside a custom block used in the script, then Snap! may not res
to clicks.)

If a script is shown with a *red* halo, that means that an error was caught in that script, such as
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 a
and/or *redo* button that can be used to undo or redo block and script drops. When you un
an input slot, whatever used to be in the slot is restored. The redo button appears once you've

The third button starts keyboard editing mode (Section D, page 130).

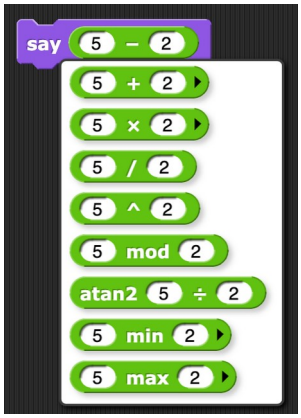
Control-click/right-clicking a primitive block within a script shows a menu like this one:

command bloc 

reporter block 

The **help...** option shows the help screen for the block, just as in the palette. The other options
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 repla
similar block, keeping the input expressions in place. For example, here's what happens when
relabel... for an arithmetic operator:



Note that the inputs to the existing **–** block are displayed in the menu of alternatives also. Click
menu to choose it, or click outside the menu to keep the original block. Note that the last three
available in the palette; you must use the relabel feature to access them.

Not every reporter has a **compile** option; it exists only for the higher order functions. When s
lightning bolt appears before the block map over and Snap! tries to compile the function in
the ring to JavaScript, so it runs at primitive speed. This works only for simple functions (but the
function still works even if the compilation doesn't). The function to be compiled must be quick
be uninterruptable; in particular, if it's an infinite loop, you may have to quit your browser to re

Therefore, **save your project before** you experiment with the compilation feature. The right-
a compiled higher order function will have an **uncompile** option. This is an experimental featu

The **duplicate** option for a command block makes a copy of the *entire script* starting from the
For a reporter, it copies only that reporter and its inputs. The copy is attached to the mouse, ar
it to another script (or even to another Block Editor window), even though you are no longer hol
mouse button. Click the mouse to drop the script copy.

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

The **extract** option removes the selected block from the script and leaves you holding it with t
other words, it's like the block picture option, but it doesn't leave a copy of the block in the orig
block is at the end of its script, this option does not appear. (Just grab the block with the mouse
for this operation is to *shift-click* and drag out the block.

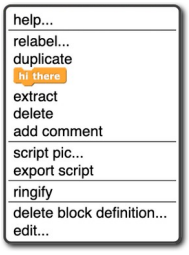
The **delete** option deletes the selected block from the script.

The **add comment** option creates a comment, like the same option in the background of the s
attaches it to the block you clicked.

The **script pic…** option saves a picture of the entire script, not just from the selected block to
your download folder; or, in some browsers, opens a new browser tab containing the picture. In
you can use the browser's Save feature to put the picture in a file. This is a super useful feature
be writing a Snap! manual! (If you have a Retina display, consider turning off Retina support b
script pictures; if not, they end up huge.) For reporters not inside a script, there is an additiona
option that calls the reporter and includes a speech balloon with the result in the picture. Note:
downloaded file is a "smart picture": It also contains the code of the script, as if you'd exported
you later drag the file into the costumes tab, it will be loaded as a costume. But if you drag it in
it will be loaded as a script, which you can drop wherever you want it in the scripting area.

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

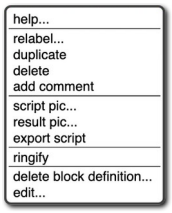
Clicking a *custom* block in a script gives a similar but different menu:



The **relabel…** option for custom blocks shows a menu of other same-shape custom blocks with
At present you can't relabel a custom block to a primitive block or vice versa. The two options a

for custom blocks only, are the same as in the palette. The other options are the same as for pr
commands.

If a reporter block is in the scripting area, possibly with inputs included, but not itself serving as
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
with the result of calling the block.

Broadcast and **broadcast and wait** blocks in the scripting area have an additional option: **r**
clicked, it causes a momentary (be looking for it when you click!) halo around the picture in the
those sprites that have a **when I receive** hat block for the same message. Similarly, **when I r**
senders… option that light up the sprite corral icons of sprites that **broadcast** the same mess

Scripting Area Background Context Menu

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

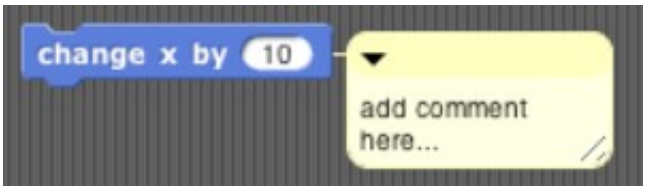
The **undrop** option is a sort of "undo" feature for the common case of dropping a block somew
where you meant it to go. It remembers all the dragging and dropping you've done in this sprit
(that is, other sprites have their own separate drop memory), and undoes the most recent, retu
its former position, and restoring the previous value in the relevant input slot, if any. Once you'
something, the **redrop** option appears, and allows you to repeat the operation you just undid.
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
margin, and with uniform spacing between scripts. This is a good idea if you can't read your ow

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

You can drag the bottom right corner of the comment box to resize it. Clicking the arrowhead
changes the box to a single-line comp add comment I , 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

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



Comments have their own context menu, with obvious meanings:

Back to the options in the menu for the background of the scripting area (picture on the previc

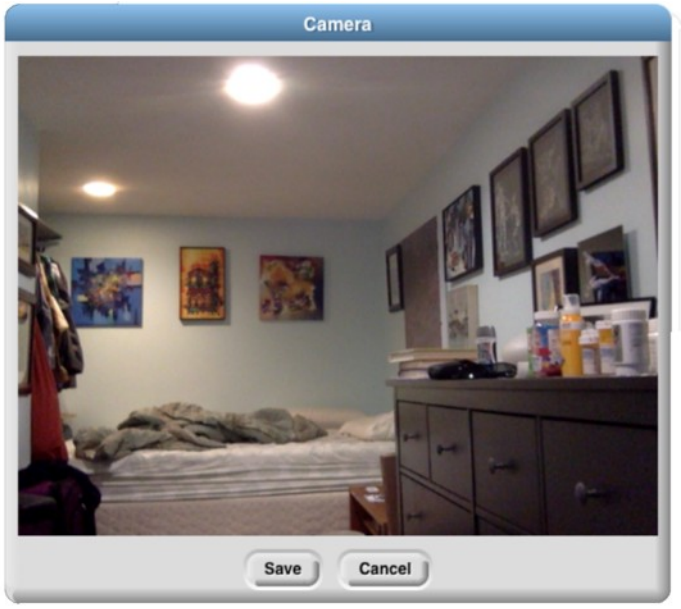
The **scripts pic...** option saves, or opens a new browser tab with, a picture of *all* scripts in the just as they appear, but without the grey striped background. Note that "all scripts in the script just the top-level scripts of the current sprite, not other sprites' scripts or custom block definitio "smart picture"; if you drag it into the scripting area, it will *create a new sprite* with those script area.

Finally, the **make a block...** option does the same thing as the "Make a block" button in the p 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 within Snap! or imported from files or other browser tabs if your browser supports that. Clickin selects it; that is, the sprite will look like the selected costume. Clicking on the paintbrush icon *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 stage unless you shrink it in the Paint Editor). This works only if you give Snap! permission camera, and maybe only if you opened Snap! in secure (HTTPS) mode, and then only if your br



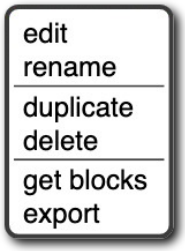
Brian's bedroom when he's staying at Paul's house.

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 two pictures below show what the stage looks like after drawing a square in each mode; **tip** (ot "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 actually the middle of the four vertices, the concave one. (If the shape were a simple isosceles a fancier arrowhead, it would mean the midpoint of the back edge.) The advantage of **tip** mod sprite is less likely to obscure the drawing. The advantage of **middle** mode is that the rotation rarely at a tip, and students are perhaps less likely to be confused about just what will happen i turtle to turn 90 degrees from the position shown. (It's also the traditional rotation point of the which originated this style of drawing.)

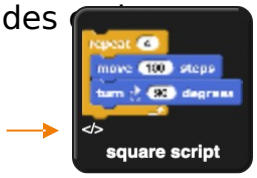
Costumes other than the turtle have a different context menu:



The **edit** option opens the Paint Editor on this costume. The **rename** option opens a dialog box can rename the costume. (A costume's initial name comes from the file from which it was impo something like **costume5**.) **Duplicate** makes a copy of the costume, in the same sprite. (Pres that because you intend to edit one of the copies.) **Delete** is obvious. The **get blocks** option smart costume, and brings its script to the scripting area. The **export** option saves the costume computer, in your usual downloads folder.

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

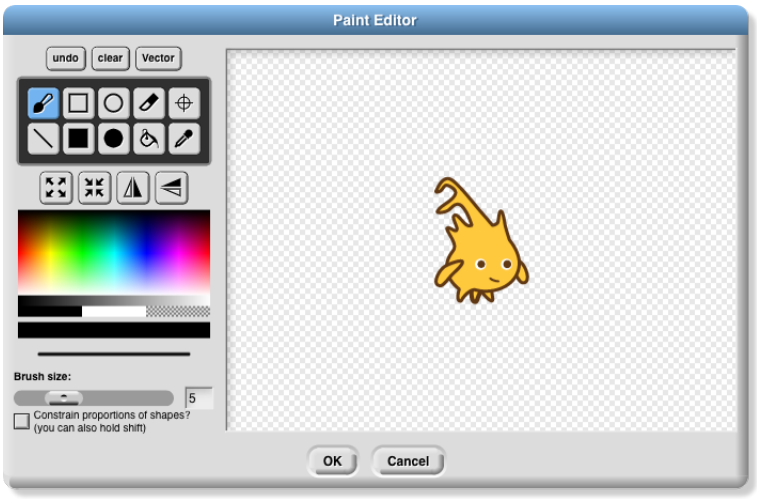
If you drag a *smart picture* of a script into the Costumes tab, its icon will display the text "</> remind you that it includes



Its right-click menu will have an extra **get blocks** option that switches to the Scripts tab with th be dropped there.

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 yo be edited only if they are in a bitmap format (png, jpeg, gif, etc.). There is a vector editor, but i creating a costume, not editing an imported vector (svg) picture. Unlike the case of the Block E 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 to to right, are the paintbrush tool, the outlined rectangle tool, the outlined ellipse tool, the eraser rotation point tool. The bottom row tools are the line drawing tool, the solid rectangle tool, the tool, the floodfill tool, and the eyedropper tool. Below the tools is a row of four buttons that imm the picture. The first two change its overall size; the next two flip the picture around horizontall Below these are a color palette, a greyscale tape, and larger buttons for black, white, and transp Below these is a solid bar displaying the currently selected color. Below that is a picture of a lin brush width for painting and drawing, and below that, you can set the width either with a slider

number (in pixels) into the text box. Finally, the checkbox constrains the line tool to draw horiz vertically, the rectangle tools to draw squares, and the ellipse tools to draw circles. You can get temporarily by holding down the shift key, which makes a check appear in the box as long as yo (But the Caps Lock key doesn't affect it.)

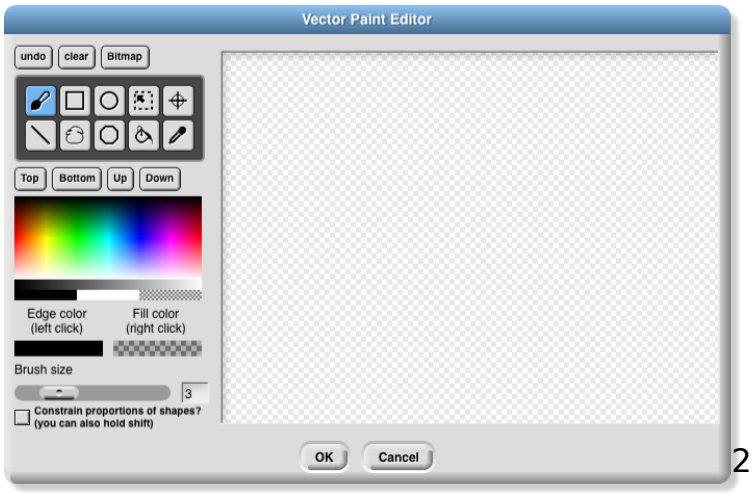
You can correct errors with the **undo** button, which removes the last thing you drew, or the **cl** which erases the entire picture. (Note, it does *not* revert to what the costume looked like before editing it! If that's what you want, click the **Cancel** button at the bottom of the editor.) When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

Note that the ellipse tools work more intuitively than ones in other software you may have use dragging between opposite corners of the rectangle circumscribing the ellipse you want, so that your dragging have no obvious connection to the actual shape, in Snap! you start at the center want and drag out to the edge. When you let go of the button, the mouse cursor will be on the drag out from the center at 45 degrees to the axes, the resulting curve will be a circle; if you dr horizontally or vertically, the ellipse will be more eccentric. (Of course if you want an exact circ down the shift key or check the checkbox.) The rectangle tools, though, work the way you expe 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 Pain tool will select the color at the mouse cursor for use in the Paint Editor. You can only do this on Paint Editor automatically selects the paintbrush when you choose a color. (Of course you can c eyedropper tool button again.)

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

The vector editor's controls are much like those in the bitmap editor. One point of difference is bitmap editor has two buttons f solid a outline rectangles, and similarly for ellipses, editor there is always an edge color and a fill color, even if the latter is "transparent paint," and button per shape is needed. Since each shape that you draw is a separate layer (like sprites on are controls to move the selected shape up (frontward) or down (rearward) relative to other sha selection to 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
(The sprite always has an appearance unless hidden, but it doesn't sing unless explicitly asked.)
menu for sounds has only **rename, delete**, and **export** options, and it has a clickable button la
Stop as appropriate. There is a sound *recorder,* which appears if you click th d record butto

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

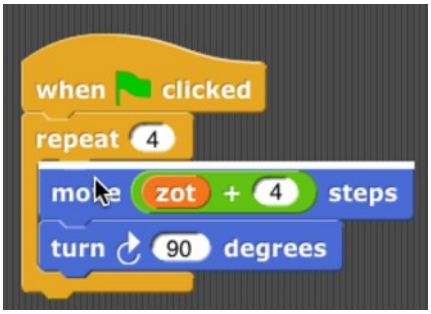
D. Keyboard Editing

An ongoing area of research is how to make visual programming languages usable by people wi
motoric disabilities. As a first step in this direction, we provide a keyboard editor, so that you ca
edit scripts without tracking the mouse. So far, not every user interface element is controllable
we haven't even begun providing *output* support, such as interfacing with a speech synthesizer
which we know we have a long way to go! But it's a start. The keyboard editor may also be use
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 are
editor at that point: either editing an existing script or, if you shift-click on the background of th
editing a new script at the mouse position. Alternatively, typing **shift-enter** will start the edito
script, and you can use the tab key to switch to another script. Or you can click the keyboard b
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 script

Navigating in the keyboard editor

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

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

You can rearrange scripts within the scripting area from the keyboard. Typing **shift-arrow** ke
up, or down) will move the current script. If you move it onto another script, the two won't snap
one you're moving will overlap the one already there. This means that you can move across an
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* t
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* (a
bar is deleted. If the focus is on an input slot, the reporter in that slot is deleted. (If that input s
value, it will appear in the slot.) If the focus is on a *variadic* input (one that can change the num
clicking on arrowheads), then *one* input slot is deleted. (When you right-arrow into a variadic in
first covers the entire thing, including the arrowheads; another right-arrow focuses on the first s
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
input, the enter key adds one more input slot. If the focus is on a white input slot (one that doe
reporter in it), then the enter key selects that input slot for *editing;* that is, you can type into it,
clicked on the input slot. (Of course, if the focus is on an input slot containing a reporter, you ca
backspace key to delete that reporter, and then use the enter key to type a value into it.) When
the value, type the enter key again to accept it and return to navigation, or the escape key if yo
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
is on a single input slot. If the focus is on a slot with a pulldown menu of options, then the spac
menu. (If it's a block-colored slot, meaning that only the choices in the menu can be used, the
the same thing. But if it's a white slot with a menu, such as in the **turn** blocks, then enter lets y
while space shows the menu.) Otherwise, the space key shows a menu of variables available at
script. In either case, use the up and down arrow keys to navigate the menu, use the enter key
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 oth
generating a character) activates the *block search palette.* This palette
accessible by typing control-F or command-F outside the keyboard edit
clicking the search button floating at the top of the palette, has a text e
top, followed by blocks whose title text includes what you type. The ch
typed to start the block search palette is entered into the text field, so
palette of blocks containing that character. Within the palette, blocks w
with the text you type come first, then blocks in which *a word* of the tit
text you type, and finally blocks in which the text appears inside a word
Once you have typed enough text to see the block you want, use the arrow keys to navigate to
palette, then enter to insert that block, or escape to leave the block search palette without inse
(When not in the keyboard editor, instead of navigating with the arrow keys, you drag the block
the script, as you would from any other palette.)

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

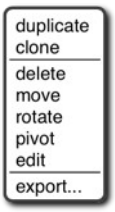
As the example shows, you can also use **parentheses** for grouping, and non-numeric operands
variables or primitive functions. (A variable name entered in this way may or may not already e
Only **round** and the ones in the pulldown menu of the **sqrt** block can be used as function name

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.

Sprites
Most sprites can be moved by clicking and dragging them. (If you have unchecked the **dragg**
a sprite, then dragging it has no effect.) Control-clicking/right-clicking a sprite shows this conte



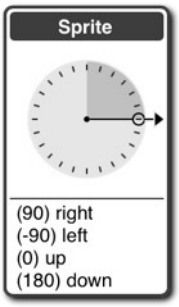
The **duplicate** option makes another sprite with copies of the same scripts, same costumes, e
The new sprite starts at a randomly chosen position different from the original, so you can see
which. The new sprite is *selected:* It becomes the current sprite, the one shown in the scripting
option makes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

The **delete** option deletes the sprite. It's not just hidden; it's gone for good. (But you can unc clicking the wastebasket just below the right edge of the stage.) The **edit** option selects the sp actually change anything about the sprite, despite the name; it's just that making changes in th will change this sprite.

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

You can ordinarily just grab and move the sprite without this option, but there are two reasons y it: First, it works even if the "draggable" checkbox above the scripting area is unchecked. Secc 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 i** 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 the sprite, then turning the sprite will revolve as well as rotate it around the pivot.) When done stage not on the crosshair. Note that, unlike moving the pivot point in the Paint Editor, this tech visibly move the sprite on the stage. Instead, the values of **x position** and **y position** will cha

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

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

Variable watchers
Right-clicking on a variable watcher shows this menu:

The first section of the menu lets you choose one of three visualizations of the watcher:

The first (normal) visualization is for debugging. The second (large) is for displaying informatio a project, often the score in a game. And the third (slider) is for allowing the user to control the behavior interactively. When the watcher is displayed as a slider, the middle section of the men control the range of values possible in the slider. It will take the minimum value when the slide the left, the maximum value when all the way to the right.

The third section of the menu allows data to be passed between your computer and the variab **import...** option will read a computer text file. Its name must end with `.txt`, in which case the text is read int the variable as is, or `.csv` or `.json`, in which case the text is converted into a list structure, which will be a two-dimensional array for csv (comma-separated values) data, but can be any shape for js **data...** option prevents that conversion to list form. The **export...** option does the opposite co a text-valued variable value into a file unchanged, but converting a list value into csv format if the one- or two-dimensional, or into json format if the list is more complicated. (The scalar values w must be numbers and/or text; lists of blocks, sprites, costumes, etc. cannot be exported.)

An alternative to using the **import...** option is simply to drag the file onto the Snap! window, i variable will be created if necessary with the same name as the file (but without the extension).

If the value of the variable is a list, then the menu will include an additional **blockify** option; c generate an expression with nested **list** blocks that, if evaluated, will reconstruct the list. It's u imported a list and then want to write code that will construct the same list later.

The stage itself

Control-clicking/right-clicking on the stage background (that is, anywhere on the stage except watcher) 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 t Note that when the stage is selected, some blocks, especially the Motion ones, are not in the pa the stage can't move.

The **show all** option makes all sprites visible, both in the sense of the **show** block and by brin 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 b drawn with the pen, and any visible sprites. What you see is what you get. (If you want a pictu background, select the stage, open its **costumes** tab, control-click/right-click on a background,

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

If you previously turned on the **log pen vectors** option, and there are logged vectors, the me extra option, **svg…**, that exports a picture of the stage in vector format. Only lines are logged, made with the **fill** block.

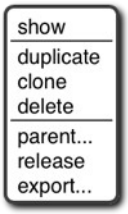
F. The Sprite Corral and Sprite Creation Buttons

Between the stage and the sprite corral at the bottom right of the Snap! window is a dark grey three buttons at the left and one at the right. The first three are used to create a new sprite. T makes a sprite with just the turtle costume, with a randomly chosen position and pen color. down the Shift key while clicking, the new sprite's direction will also be random.) The second bu makes a sprite and opens the Paint Editor so that you can make your own costume for it. (Of co click the first button and then click the paint button in its **costumes** tab; this paint button is a s that.) Similarly, the third b uses your camera, if possible, to make a costume for the new sp

The trash can but at the right has two uses. You can drag a sprite thumbnail onto it from th corral to delete that sprite, or you can click it to undelete a sprite you deleted by accident.

In the sprite corral, you click on a sprite's "thumbnail" picture to select that sprite (to make it t scripts, costumes, etc. are shown in the scripting area). You can drag sprite thumbnails (but no reorder them; this has no special effect on your project, but lets you put related ones next to ea 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:

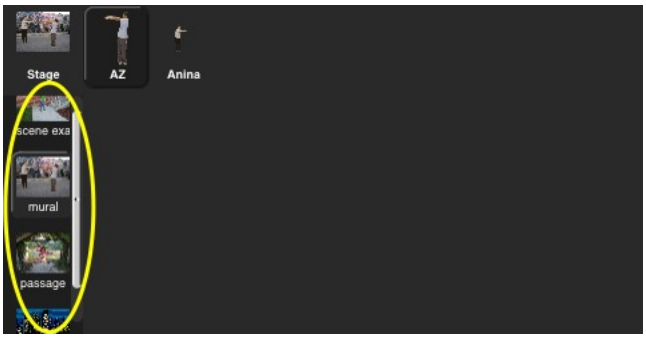


The **show** option makes the sprite visible, if it was hidden, and also brings it onto the stage, if it the stage boundary. The next three options are the same as in the context menu of the actual stage, discussed above.

The **parent...** option displays a menu of all other sprites, showing which if any is this sprite's p allowing you to choose another sprite (replacing any existing parent). The **release** option is sh sprite is a (permanent, or it wouldn't be in the sprite corral) clone; it changes the sprite to a tem (The name is supposed to mean that the sprite is released from the corral.) The **export...** optio 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 the stage, just like the same option in the context menu of the stage background. If pen trails a there will also be an **svg...** option.

If your project includes scenes, then under the stage icon in the sprite corral will be the *scene*



Clicking on a scene will select it; right-clicking will present a menu in which you can rename, de the scene.

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 orde project automatically. You can think of such a URL as just running the project rather than as ru especially if the URL says to start in presentation mode and click the green flag. The general fo

`https://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 project in the Snap! cloud; the first two are for projects that have been exported and made available a Internet.

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

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

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

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,
`&lang=fr`	Set language to (in this example) French.
`&noCloud`	Don't allow cloud operations from this project (for running projects from unkn sources that include JavaScript code)
`&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.
`&blocksZoom=n`	Like the **Zoom blocks** option in the Settings menu.

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:

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

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

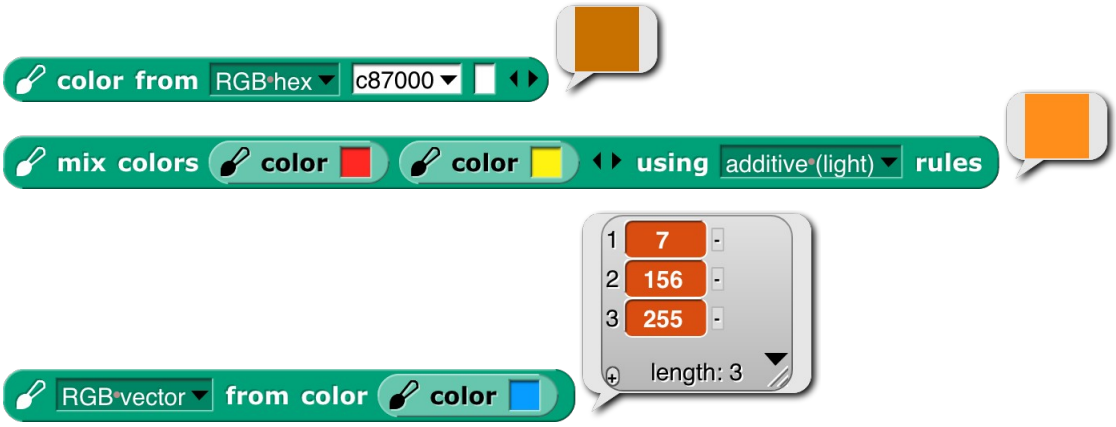
H. Mirror Sites

If the site `snap.berkeley.edu` is ever unavailable, you can load Snap! at the following mirror site

- `https://bjc.edc.org/snapsource/snap.html`
- `https://cs10.org/snap`

Appendix A. Snap*!* color library

The Colors and Crayons library provides several tools for manipulating color. Although its main
controlling a sprite's pen color, it also establishes colors as a first class data type:



For people who just want colors in their projects without having to be color experts, we provide
mechanisms: a *color number* scale with a broad range of continuous color variation and a set of
organized by color family (ten reds, ten oranges, etc.) The crayons include the block colors:

For experts, we provide color selection by RGB, HSL, HSV, X11/W3C names, and variants on tho

Introduction to Color

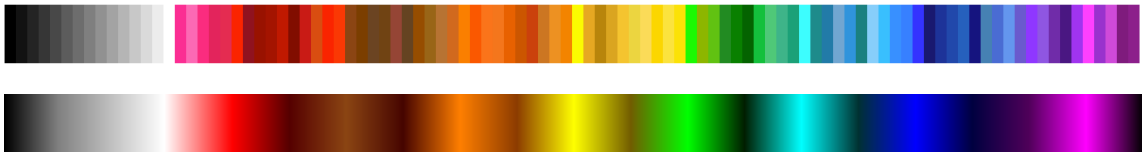
Your computer monitor can display millions of colors, but you probably can't distinguish that ma
example, here's red 57, green 180, e 200: And here's red 57, gr n *182,* blue 200:
able to tell them apart if you see them si side: ... but maybe not even then.

Color space—the collection of all possible colors—is three-dimensional, but there are many way
dimensions. RGB (red-green-blue), the one most commonly used in computers, matches the wa
displays produce color. Behind every dot on the screen are three tiny lights: a red one, a green
one. But if you want to print colors on paper, your printer probably uses a different set of three
(cyan-magenta-yellow). You may have seen the abbreviation CMYK, which represents the comr
of adding black ink to the collection. (Mixing cyan, magenta, and yellow in equal amounts is su
in black ink, but typically it comes out a muddy brown instead, because chemistry.) Other syste
mimic human perception are HSL (hue-saturation-lightness) and HSV (hue-saturation-value). Th
many more, each designed for a particular purpose.

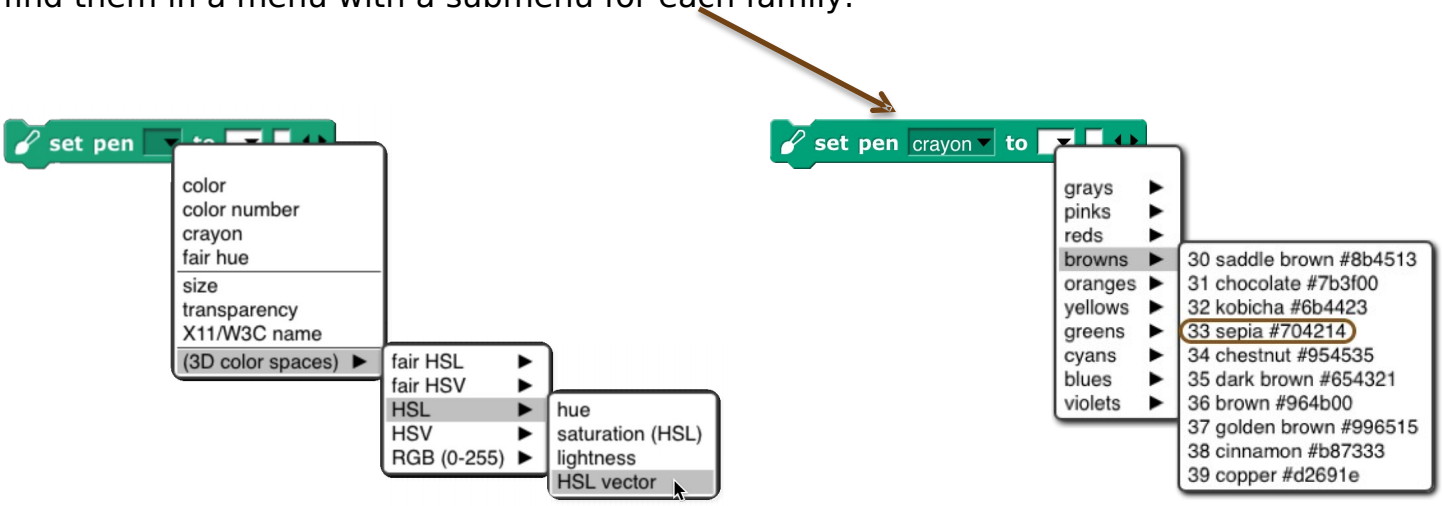
If you are a color professional—a printer, a web designer, a graphic designer, an artist—then yo
understand all this. It can also be interesting to learn about. For example, there are colors that
your computer display can't generate. If that intrigues you, look up color theory in Wikipedia.

Crayons and Color Numbers

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



The first row shows 100 distinct colors. They have names; this is pumpkin, and this is supposed to think of them as a big box of 100 crayons. They're arranged in families: grays, pinks, oranges, etc. But they're not consistently ordered within a family; you'd be unlikely to say "next project(But look at the crayon spiral on page 142.) Instead, you'd think "I want this to look like a really old-fashioned photo" and so you'd find that is crayon number 33. You don't have to memorize the numbers! You find them in a menu with a submenu for each family.



Or, if you know the crayon name: 

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

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

The set of *color numbers* is arranged so that each color number is visually near each of its neighbors, dark colors alternate for each family. Color numbers range from 0 to 99, like crayon numbers, but 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.)

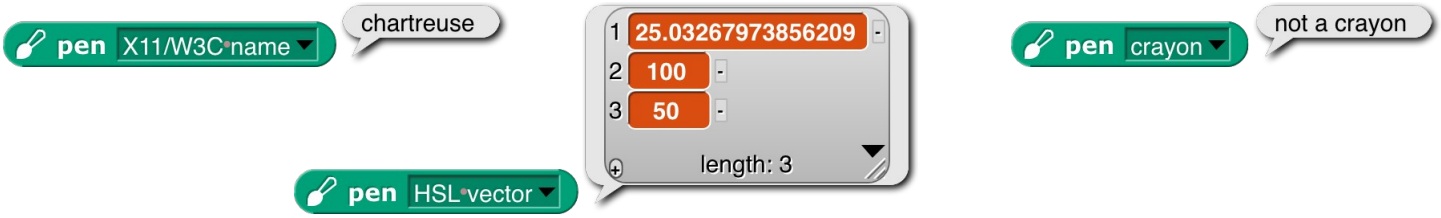
Both of these scales include the range of shades of gray, from black to white. Since black is the initial pen color, and black is Snap! users would traditionally try to use **set color** to escape from black, and it wouldn't work. By including black in the sam eliminate the Black Hole problem if people use only the recommended color scales.

We are making a point of saying "color number" for what was sometimes called just "color" in of the library, because we now reserve the name "color" for an actual color, an instance of the

How to Use the Library

There are three library blocks specifically about controlling the pen. They have the same name primitive Pen blocks:

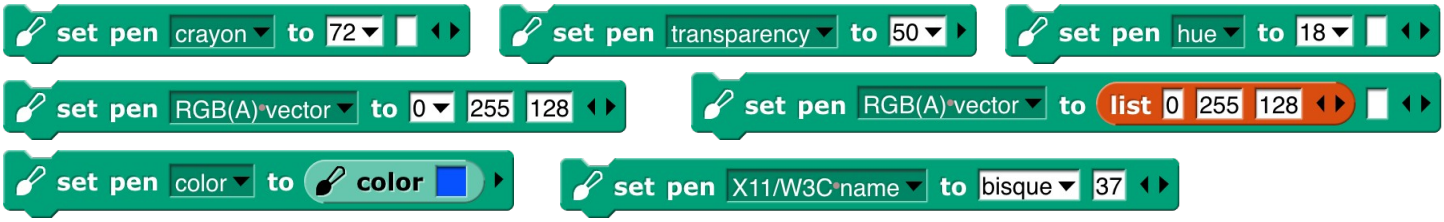
The first (Pen block-colored) input slot is used to select which color scale you want to use. (Thes allow reading or setting two block properties that are not colors: the pen size and its transparen reporter requires no other inputs; it reports the state of the pen in whatever dimension you cho



As the last example shows, you can't ask for the pen color in a scale incompatible with how you block can deduce what you want from what it knows about the current pen color.

The **change pen** block applies only to numeric scales (including vectors of three or four numb numeric or list input to the current pen value(s), doing vector (item-by-item) addition for vector

The **set pen** block changes the pen color to the value(s) you specify. The meaning of the whit depends on which attribute of the pen you're setting:



In the last example, the number 37 sets the *transparency,* on the scale 0=opaque, 100=invisibl attributes are on a 0–100 scale except for RGB components, which are 0–255.) A transparency combined with any of these attribute scales.

The library also includes two constructors and a selector for colors as a data type:

The latter two are inverses of each other, translating between colors and their attributes. The **c** attribute menu has fewer choices than the similar **set pen** block because you can, for example of the existing pen color leaving the rest unchanged, but when creating a color out of nothing y provide its entire specification, e.g., all of Red, Green, and Blue, or the equivalent in other scale on the next page, we provide two *linear* (one-dimensional) color scales that allow you to specify single number, at the cost of including only a small subset of the millions of colors your comput If you have a color and want another color that's the same except for one number, as in the Re can use this blo

Finally, the library includes the **mix** block and a helper:

We'll have more to say about these after a detour through color theory.

That's all you have to know about colors! *Crayons* for specific interesting ones, *color numbers* transformation from one color to the next. But there's a bit more to say, if you're interested. If (But look at the samples of the different scales on page 145.)

More about Colors: Fair Hues and Shades

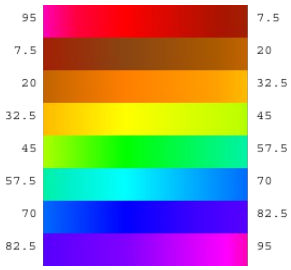
Several of the three-dimensional arrangements of colors use the concept of "hue," which more
where a color would appear in a rainbow (magenta, near the right, is a long story):

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

And this is already a handwave, because the range of colors that can be generated by RGB monitors doesn't inclu
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 th
ranges of wavelength of light waves are lumped together as named colors. The eye is just "tun
wide range of colors as green. (See Rods and Cones.) And different human cultures give names to different co
ranges. Nevertheless, in old Scratch projects, you'd say **change pen color by 1** and it'd take
color that wasn't green.

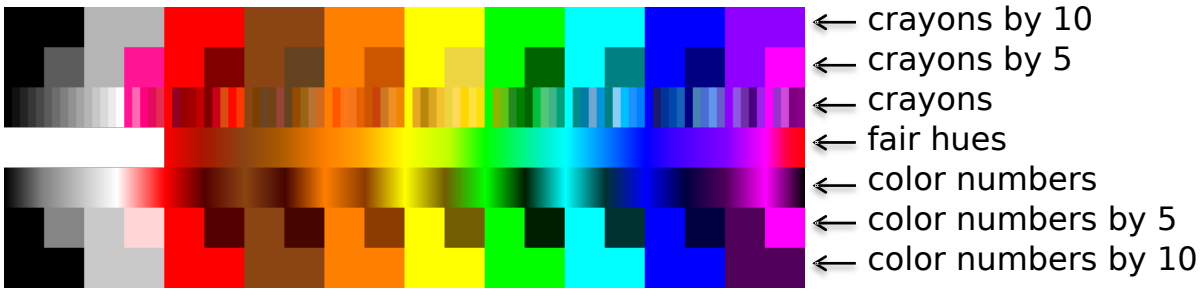
For color professionals, there are good reasons to want to work with the physical rainbow hue la
amateurs using a simplified, one-dimensional color model, there's no reason not to use a more
friendly hue scale:



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

Our *color number* scale is based on fair hues, adding a range of grays from black (color number
number14) and also adding *shades* of the spectral colors. (In color terminology, a *shade* is a da
color; a lighter version is called a *tint*.) Why do we add shades but not tints? Partly because I fi
exciting. A shade of red can be dark candy apple red or maroon, but a tint is just some k
pink. This admitted prejudice is supported by an objective fact: Most projects are made on
background, so dark colors stand out better than light ones.

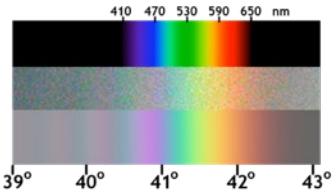
So, in our color number scale, color numbers 0 to 14 are kinds of gray; the remaining color num
the fair hues, but alternating full-strength colors with shades.



This chart shows how the color scales discussed so far are related. Note that all scales range fr
fair hues scale has been compressed in the chart so that similar colors line up vertically. (Its di

different because it doesn't include the grays at the left. Since there are eight color families, th
fair hues are at multiples of 100/8=12.5, starting with red=0.)

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



Among purples, the official spectral violet (crayon 90) is the end of the spe
Magenta, brighter than violet, isn't a 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 p
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.)

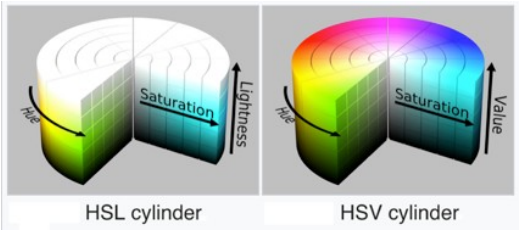
The light gray at color number 10 is slightly different from crayon 10 just because of roundoff i
crayon values. Color number 90 is different from crayon 90 because the official RGB violet (equa
blue) is actually lighter than spectral violet. The purple family is also unusual because magenta
color number 95, is lighter than the violet at 90. In other families, the color numbers, crayons, a
hues all agree at multiples of ten. These multiple-of-ten positions are the standard RGB primary
colors, e.g., the yellow at color number 50 is (255, 255, 0) in RGB. (Gray, brown, and orange do
simple RGB settings.)

The color numbers at odd multiples of five are generally darker shades than the corresponding
latter are often official named shades, e.g., teal, crayon 65, is a half-intensity shade of cyan. Th
numbers, though, are often darker, since they are chosen to be the darkest color in a given fam
different from black. The pink at color number 15, though, is quite different from crayon 15, be
former is a pure tint of red, whereas the crayon, to get a more interesting pink, has a little mage
Color numbers at multiples of five are looked up in a table; other color values are determined by
interpolation in RGB space. (*Crayons* are of course all found by table lookup.)

The **from color** block behaves specially when you ask for the *color number* of a color. Most colors don't exactly
and for other attributes of a color (crayon number, X11 name) you don't get an answer unless the color exactly m
names or numbers in that attribute. But for color number, the block tries to find the *nearest color number* to the
result will be only approximate; you can't use the number you get to recreate the input color. But you can start c
numbers as you animate the sprite.

Perceptual Spaces: HSV and HSL

RGB is the right way to think about colors if you're building or programming a display monitor; C
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 e you m
37% red light, 52% green, and 11% blue. The *hue* scale is one dimension of mos ots at a
perceptual scale. The square at the right has pale blues along the top edge, dar long th
right edge, various shades of gray toward the left, black at the bottom, and pure spectral blue i
top right corner. Although no other point in the square is pure blue, you can tell at a glance tha
spectral color is mixed with the blue.



attribution: Wikipedia user SharkD, CC BY-SA 3.0

Aside from hue, the other two dimensions of a color space hav
represent how much white and/or black is mixed with the spec
color. (Bear in mind that "mixing black" is a metaphor when i
comes to monitors. There really is black paint, but there's no
thing as black light.) One such space, HSV, has one dimensio
the amount of color (vs. white), called *saturation,* and one for

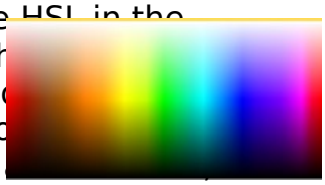
amount of black, imaginatively called *value.* HSV stands for Hue-Saturation-Value. (Value is als
brightness.) The *value* is actually measured backward from the above description; that is, if val
pure black; if value is 100, then a saturation of 0 means all white, no spectral color; a saturation
white at all. In the square in the previous paragraph, the *x* axis is the saturation and the *y* axis
entire bottom edge is black, but only the top left corner is white. HSV is the traditional color spa
Scratch and Snap!. **Set pen color** set the hue; **set pen shade** set the value. There was origin
set the saturation, but there's a **set brightness effect** Looks block to control the saturation of
costume. (I speculate that the Scratch designers, like me, thought tints were less vivid than sha
white background, so they made it harder to control tinting.)

But if you're looking at colors on a computer display, HSV isn't really a good match for human p
Intuitively, black and white should be treated symmetrically. This is the HSL (hue-saturation-lig
space. *Saturation,* in HSL, is a measure of the *grayness* or *dullness* of a color (h it com
being on a black-and-white scale) and *lightness* measures *spectralness* with pur t one e
pure black at the other end, and spectral color in the middle. The *saturation* num tually
opposite of grayness: 0 means pure gray, and 100 means pure spectral color, pr at the
lightness is 50, midway between black and white. Colors with lightness other than 50 have som
mixed in, but saturation 100 means that the color is as fully saturated as it can be, given the an
black needed to achieve that lightness. Saturation less than 100 means that *both white and bla*
the spectral color. (Such mixtures are called *tones* of the spectral color. Perceptually, colors wi
100% don't look gr but colors with saturation 75

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

More fine print: It's misleading to talk about the spectrum of light wavelengths as if it were the same as perceived
computer display is showing you a yellow area, for example, it's doing it by turning on its red and green LEDs ove
what hits your retina *is still two wavelengths of light, red and green, superimposed.* You could make what's perce
using a single intermediate wavelength. Your eye and brain don't distinguish between those two kinds of yellow.
automatically adjusts perceived hue to correct for differences in illumination. When you place a monochrome obj
sunlight and half in the shade, you see it as one even though what's reaching your eyes from the two regions diffe
it's HSL whose use of "saturation" disagrees with the official international color vocabulary standardization comm
this from this tutorial, which you might find more coherent than jumping around Wikipedia if you're interested.

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 th
is *lightness,* the scale with black at one end and white at the other en
no sense to have only the bottom half of this selector (HSV Value) or
(HSV Saturation). And, given that you can only fit two dimensions on
makes sense to pick HSL saturation (vividness) as the one to keep at 100%.



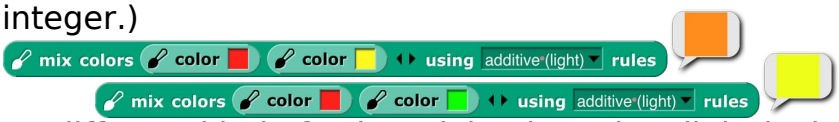
(In the fair-hue picker, some colors appea
twice: "spectral" (50% lightness) browns as shades (≈33% lightness) of red or orange, and shades of those brown

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

Mixing Colors

Given first class colors, the next question is, what operations apply to them, the way arithmetic to numbers and higher order functions apply to lists? The equivalent to adding numbers is mixi unfortunately there isn't a simple answer to what that means.

The easiest kind of color mixing to understand is *additive* mixing, which is what happens when y colored lights onto a (white) wall. It's also what happens in your computer screen, where each image is created by a tiny red light, a tiny green light, and a tiny blue light that can be combine strengths to make different colors. Essentially, additive mixing of two colors is computed by ad components, the two green components, and the two blue components. It's not *quite* that simp each component of the result must be in the range 0 to 255. So, red (255, 0, 0) mixed with gre gives (255, 255, 0), which is yellow. But red (255, 0, 0) plus yellow (255, 255, 0) can't give (510 limiting the red in the result to 255 would mean that red plus yellow is yellow, which doesn't ma Instead, if the red value has to be reduced by half (from 510 to 255), then *all three* values must half, so the result is (255, 128, 0), which is orange. (Half of 255 is 127.5, but each RGB value m integer.)

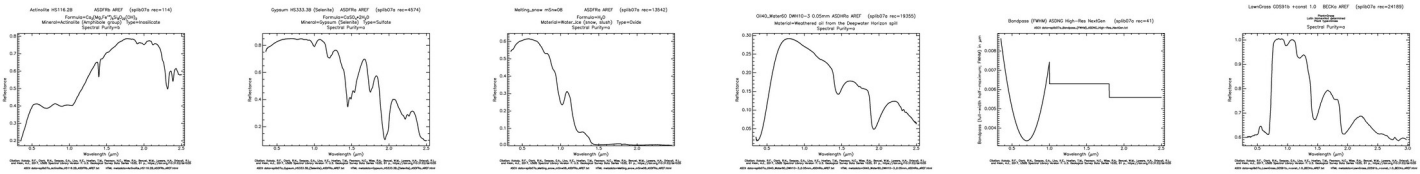


A different kind of color mixing based on light is done when different colored transparent plast held in front of a white light, as is done in theatrical lighting. In that situation, the light that get filters is what remains after some light is filtered out by the first one and some of what's left is f second one. In red-green-blue terms, a red filter filters out green and blue; a yellow filter allows through, filtering out blue. But there isn't any green light for the yellow filter to pass; it was filte red filter. Each filter can only remove light, not add light, so this is called *tractive* mixing:

Perhaps confusingly, the numerical computation of subtractive mixing is done by *multiplying* th taken as fractions of the maximum 255, so red (1, 0, 0) times yellow (1, 1, 0) is red again.

Those are both straightforward to compute. Much, much more complicated is trying to simula mixing *paints.* It's not just that we'd have to compute a more complicated function of the red, g values; it's that RGB values (or any other three-dimensional color space) are inadequate to desc of paints. Two paints can look identical, and have the same RGB values, but may still behave v when mixed with other colors. The differences are mostly due to the chemistry of the paints, bu affected by exactly how the colors are mixed. The mixing is mostly subtractive; red paint *absor* colors other than red, so what's reflected off the surface is whatever isn't absorbed by the color But there can be an additive component also.

The proper mathematical abstraction to describe a paint is a *reflectance* graph, like this:



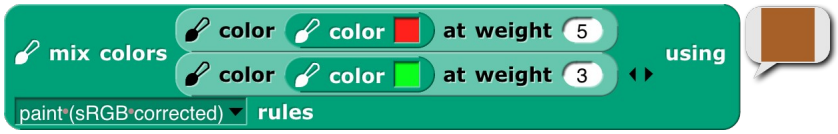
(These aren't paints, but minerals, and one software-generated spectrum, from the US Geologic Spectral Library. The details don't matter, just the fact that a graph like these gives much more than three RGB numbers.) To mix two paints properly, you multiply the *y* values (as fractions) a matching *x* coordinate of the two graphs.

Having said all that, the **mix** block takes the colors it is given as inputs and converts them int
are *typical* paint reflectance spectra that would look like those colors, and then mixes those spe
back to RGB.

But unlike the other two kinds of mixing, in this case we can't say that these colors are "the righ
would happen with real paints depends on their chemical composition and how they're mixed. There are
more mixing options, but these three are the ones that correspond to real-world color mixing.

The **mix** block will accept any number of colors, and will mix them in equal proportion. If (for a
mixing) you want more of one color than another, use the **color at weight** block to make a "w

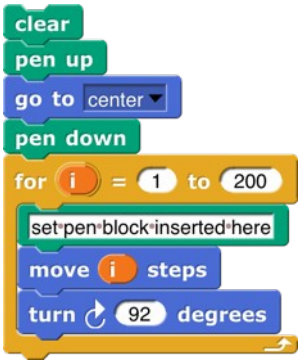
This mixes four parts red paint to one part green paint. All colors in a mixture can be weighted:



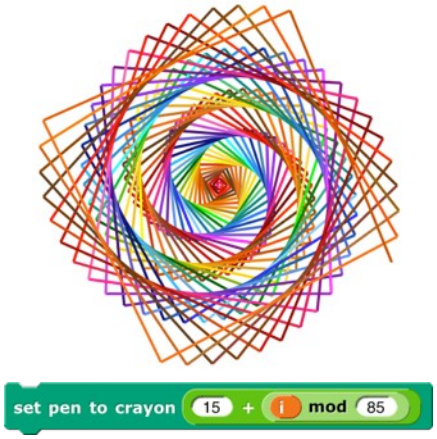
(Thanks to Scott Burns for his help in understanding paint mixing, along with David Briggs's tutorial. Remaining m

tl;dr

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



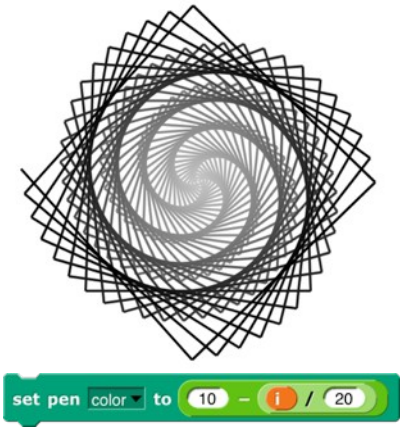
Fair hues.



Crayons, no grays.

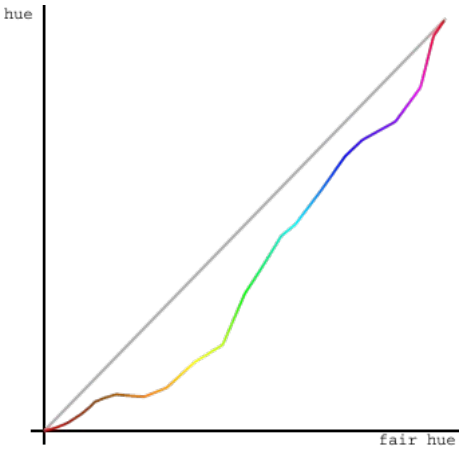
All color numbers.

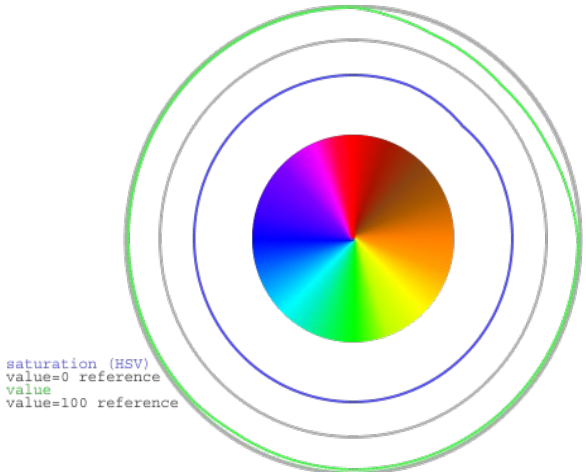
Color numbers, no grays.



Just grays.

Subappendix: Geeky details on fair hue



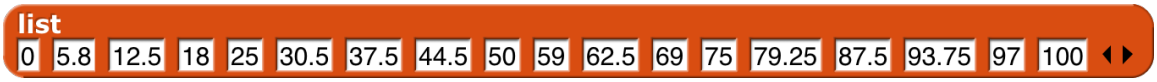


The left graph shows that, unsurprisingly, all of the brown fair hues make essentially no progress in real hue, with section actually a little retrograde, since browns are really shades of orange and so the real hues overlap betwee oranges. Green makes up some of the distance, because there are too many green real hues and part of the goa to squeeze that part of the hue spectrum. But much of the catching up happens very quickly, between pure mag and the start of the purple-red section at fair hue 97. This abrupt change is unfortunate, but the alternatives invo space from red or stealing space from purple (which already has to include both spectral violet and RGB magenta discontinuous derivative at the table-lookup points, of which there are two in each color family, one at the pure-n multiples of 12.5, and the other *roughly* halfway to the next color family, except for the purple family, which has l (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.)

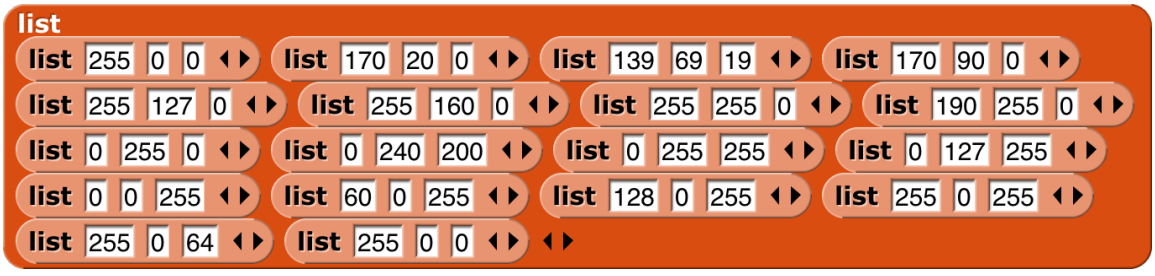
The right graph shows the HSV saturation and value for all the fair hues. Saturation is at 100%, as it should be in for a very slight drop in part of the browns. (Browns are shades of orange, not tints, so one would expect full satu some of the browns are actually mixtures with related hues.) But value, also as expected, falls substantially in th about 56% (halfway to black) for the "pure" brown at 45° (fair hue 12.5). But the curve is smooth, without inflecti 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 b saturation and value are the product (in percent) of the innate shading of the specific brown fair hue and the user saturation/value setting. When the user's previous color setting was in a real scale and the new setting is in a fai assumes that the previous saturation and value were entirely user-determined; when the previous color setting w and the new setting is also in a fair scale, the program remembers the user's intention from the previous setting. are based on HSV, even though we recommend HSL to users, because HSV comes to us directly from the JavaScri 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]):

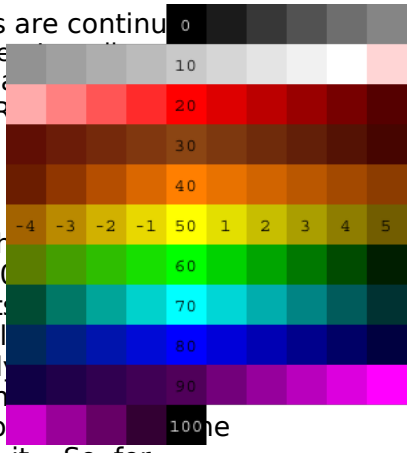


and here are the RGB settings at those points:



Subappendix: Geeky details on color numbers

Here is a picture of integer color numbers, but remember that color numbers are continu
(As usual, “continuous” values are ultimately converted to integer RGB values, so the
granularity.) Color numbers 0-14 are continuously varying grayscale, from 0=bl
14=white. Color numbers 14+ε to 20 are linearly varying shades of pink, with R
color number 20.



Beyond that point, in each color family, the multiple of ten color number in th
RGB standard color of that family, in which each component is either 255 or 0
are brown, which is of course darker than any of those colors; orange, with its
component half-strength: [255, 127, 0]; and violet, discussed below.) The foll
five is the number of the darkest color in that family, although not necessarily
the multiple of ten color number. Color numbers between the multiple of ten
following multiple of five are shades of colors entirely within the family. Colo
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,
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.

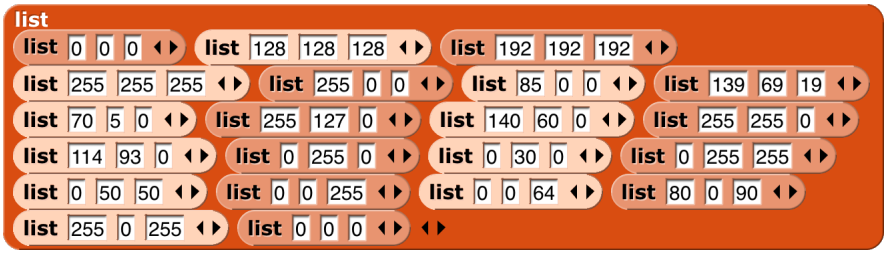
In the color number chart, all the dark color numbers look a lot like black, but they’re quite different. Here are the
each color number family.

Darkest yellow doesn’t look entirely yellow. You might see it as greenish or brownish. As it t
darkest color that really looks yellow is hardly dark at all. This color was hand-tweaked to
green nor brown to me, but ymmv.

In some families, the center+5 *crayon* is an important named darker version of the cen
family, [128, 0, 0] is “maroon.” In the cyan family, [0, 128, 128] is “teal.” An early ver
number scale used these named shades as the center+5 color number also. But on this
word “darkest” advisedly: You can’t find a darker shade of this family anywhere in the co
but you *can* find lighter shades. Teal is color number 73.1, 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
usually given as RGB [128, 0, 255], but that’s much brighter than the violet in an actual spectrum (see page 142).
a value hand-tweaked to look as much as possible like the violet in rainbow photos, as color number 90. (*Crayon*
Magenta, [255, 0, 255], is color number 95. This means that the colors get *brighter,* not darker, between 90 and
is actually color number 87.5, so it’s bluer than standard violet, but still plainly a purple and not a blue. It’s [39,0,
tweaked; it’s a linear interpolation between darkest blue, [0, 0, 64], and the violet at color number 90. I determin
that color number 87.5 is the darkest one that’s still unambiguously purple. (According to Wikipedia, “violet” nam
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 colo



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 196
formal language that would look like what mathematicians write on chalkboards. The then-unna
would later take its name from the first letters of the words in the book's title. It was little-know
when a formal description of the just-announced IBM System/360 in the *IBM Systems Journal* us
notation. (Around the same time, Iverson's associate Adin Falkoff gave a talk on APL to a New York Association for
Machinery chapter, with an excited 14-year-old Brian Harvey in the audience.) But it wasn't until 1966 that the first publi
implementation of the language for the System/360 was published by IBM. (It was called "APL\3
the normal slash character / represents the "reduce" operator in APL, while backslash is "expan

The crucial idea behind APL is that mathematicians think about collections of numbers, one-dir
vectors and two-dimensional *matrices,* as valid objects in themselves, what computer scientists
"first class data." A mathematician who wants to add two vectors writes $v_1 + v_2$, while a programmer writes "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 *sc*
one whose natural domain is individual numbers or text characters. A *mixed function* is one wh
includes arrays (vectors, matrices, or higher-dimensional collections). In Snap!, scalar functions
found in the green Operators palette, while mixed functions are in the red Lists palette. The thi
confusingly for Snap! users, is called *operators* in APL, but corresponds to what we call higher o
functions whose domain includes functions.

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

As explained in Section IV.F, this termwise extension of scalar functions is the main APL-like fe
Snap! itself. We also include an extension of the **item** block to address multiple dimensions, ar
length block with five list functions from APL, and a new primitive **reshape** block. The APL libra
implementation of APL features to include a few missing scalar functions and several missing m
and operators.

Programming in APL really is *very* different in style from programming in other languages, eve
appendix can't hope to be a complete reference for APL, let alone a tutorial. If you're interested
those in a library or a (probably used) bookstore, read it, and *do the exercises.* Sorry to sound l
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
language named J. The latter uses plain ASCII characters, unlike the ones with APL in their nam
the mathematician's character set, with Greek letters, typestyles (boldface and/or italics in bool

upper case, or lower case in APL) as loose type declarations, and symbols not part of anyone's a
⌊ for floor and ⌈ for ceiling. To use the original APL, you needed expensive special computer termi
was before you could download fonts in software. Today the more unusual APL characters are i
U+2336 to U+2395.) The character set was probably the main reason APL didn't take over the
has a lot to recommend it for Snap! users, mainly because it moves from the original APL idea t
must be uniform in dimension, and the elements of arrays must be numbers or single text chara
that a list can be an element of another list, and that such elements don't all have to have the s
Nevertheless, its mechanism for allowing both old-style APL arrays and more general "nested ar
complicated and hard for an APL beginner (probably all but two or three Snap! users) to underst
starting with plain APL. If it turns out to be wildly popular, we may decide later to include APL2

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.
worth the effort. For example, we didn't hyperize the **=** block because Snap! users expect it to
or-no answer about the equality of two complete structures, whatever their types and shapes.
scalar function; it compares two numbers or two characters. How could APL users live without t
if two *structures* are equal? Because in APL you can get that answer. Reading from right to le
a=b reports an array of Booleans (represented in APL as 0 for False, 1 for True); the comma op
shape of the array into a simple vector; and ∧/ means "reduce with **and**"; "reduce" is our **combine** funct
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
equal, you'd just use **+/** instead of ∧/. Note also that our APLish blocks are a little verbose, because the
up to three notations for the function: the usual Snap! name (e.g., **flatten**), the name APL prog
talking about it (**ravel**), and, in yellow type, the symbol used in actual APL code (**,**). We're not c
it; 

seems self-documenting. And **LCM (and)** is different even though it has two
out that if you represent Boolean values as 0 and 1, then the algorithm to compute the least co
two integers computes the **and** function if the two inputs happen to be Boolean. Including the A
serves two purposes: the two or three Snap! users who've actually programmed in APL will be s
they're using, but more importantly, the ones who are reading an APL tutorial while building pro
will find the block that matches the APL they're reading.

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

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

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

Non-goal: Emulate the terse APL syntax. It's too bad, in a way; as noted above, the terser expressing a computation affects APL programmers' sense of what's difficult and what isn't. Bu "terse" and "block language" in the same sentence. Our whole *raison d'être* is to make it possi program without having to memorize the syntax or the names of functions, and to allow those n enough to be self-documenting. And APL's syntax has its own issues, of which the biggest is tha functions with more than two inputs; because most mathematical dyadic functions use infix not function symbol between the two inputs), the notion of "left argument" and "right argument" is APL documentation. The thing people most complain about, that there is no operator preceden multiplication-before-addition rule in normal arithmetic notation), really doesn't turn out to be a Function grouping is strictly right to left, so **2×3+4** means two times seven, not six plus four. T getting used to, but it really doesn't take long if you immerse yourself in APL. The reason is tha many infix operators for people to memorize a precedence table. But in any case, block notatio problem, especially with Snap!'s zebra coloring. You can see and control the grouping by which which other block's input slot. Another problem with APL's syntax is that it bends over backwar reserved words, as opposed to Fortran, its main competition back then. So that dyadic "circular" function uses the left argument to select a trig function (x), so **1○x** is cos(x), and so **¯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. API respectively. The APL style of programming depends heavily on doing arithmetic on Booleans, a conditionals insist on only 0 or 1 in a Boolean input slot, not other numbers. Snap! *arithmetic* fu as 0 and **true** as 1, so our APL library tries to report Snap! Boolean values from predicate functi

Scalar functions

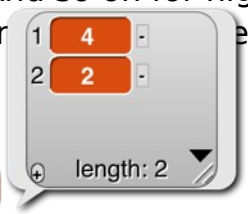
These are the scalar functions in the APL library. Most of them are straightforward to figure out block provides an APL-style version of **=** (and other exceptions) as a hyperblock that extends te **Join**, the only non-predicate non-hyper scalar primitive, has its own **scalar join** block. **7 deal 5** random vector of seven numbers from 1 to 52 with no repetitions, as in dealing a hand of cards number reports 1 if the number is positive, 0 if it's zero, or -1 if it's negative. **Roll 6** reports a r six-sided die. To roll 8 dice 

, which would look much more pleasan as **?8ρ6**. But perhaps our version is more instantly readable by someone who didn't grow up w library functions have help messages available.

Mixed functions

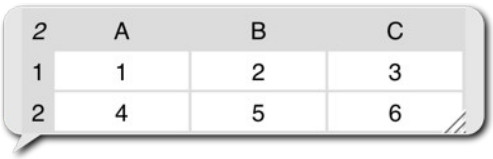
Mixed functions include lists in their natural domain or range. That is, one or both of its inputs r it always reports a list. Sometimes both inputs are naturally lists; sometimes one input of a dya function is naturally a scalar, and the function treats a list in that input slot as an implicit **map**, 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 structu dimension. For a vector, it returns a list of length 1 containing the **length of** the input. For a n two-item list of the number of rows and number of columns of the input. And so on for higher d the input isn't a list at all, then it has zero dimensions, and **shape of** repor ector. Equivalent to the **dimensions of** primitive, as of 6.6.



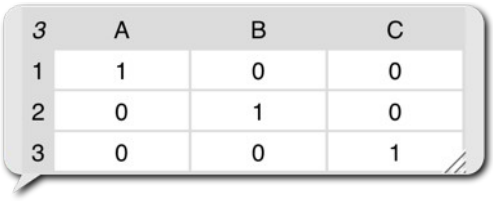
Rank of isn't an actual APL primitive, but the combination ρρ (shape of shape of a structure), which reports number of dimensions of the structure (the length of its shape vector), is too useful to omit. (It' the same character twice on the APL keyboard, but less easy to drag blocks together.) Equivale primitive, as of 6.6.

Reshape takes a shape vector (such as **shape** might report) on the left and any structure on th the shape of the right input, stringing the atomic elements into a vector in row-major order (tha row left to right, then all of the second row, etc.). (The primitive **reshape** takes the inputs in th then reports an array with the shape specified by the first input containing the items of the seco



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 ignored without reporting an error. If the right input has too *few* atomic elements, the process reported array starts again from the first element. This is most useful in the specific case of an input, which produces an array of any desired shape all of whose atomic elements are equal. B sometimes useful too:



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

ID ← {(ω,ω)⍴1,ω⍴0}

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

Catenate is like our primitive **append**, with two differences: First, if either input is a scalar, it i
one-item vector. Second, if the two inputs are of different rank, the **catenate** function is recurs
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 inte
reports a vector of the integers from 1 to the input. This is an example of a function classed as
because of its domain but because of its range. The difference between this block and the prim
from block is in its treatment of lists as inputs. **Numbers from** is a hyperblock, applying itself
input list:

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

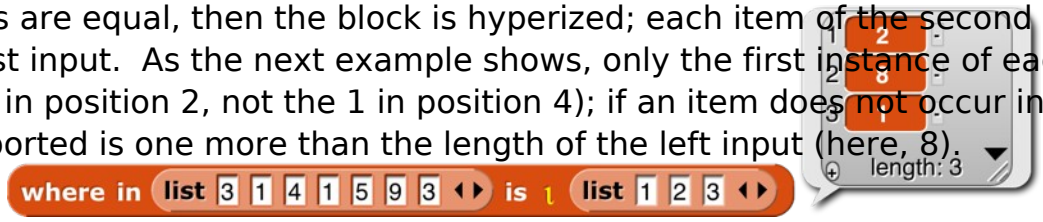
Iota has a special meaning for list inputs: The input must be a shape vector; the result is an ar
shape in which each item is a list of the indices of the cell along each dimension. A picture is w
but Snap! isn’t so good at displaying arrays with more than two dimensions, so here we reduce
list to a string:

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

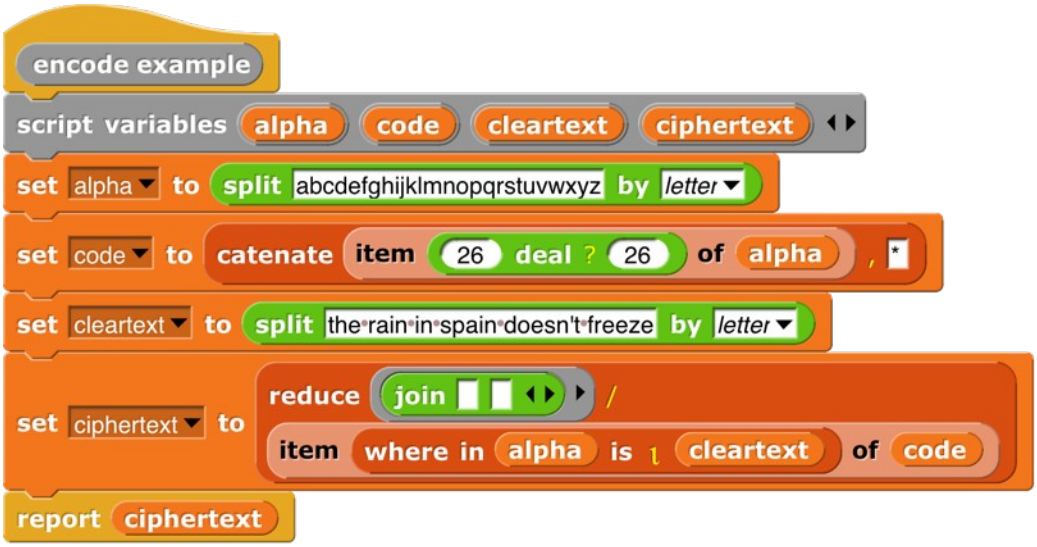
Dyadic **iota** is like the **index of** primitive except for its handling of multi-dimensional arrays.
It looks only for atomic elements, so a vector in the second input doesn’t mean to search for
that vector as a row of a matrix, which is what it means to **index of**, but rather to look
separately 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.

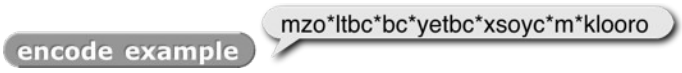
In this example, the 4 is in the second row, second column. (This is actually an extension of API more like a hyperized **index of**.) Generalizing, if the rank of the second input is less than the ra input by two or more, then iota looks for the entire second input in the first input. The reported vector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

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 m value such as 0 or **false**? Here's why:

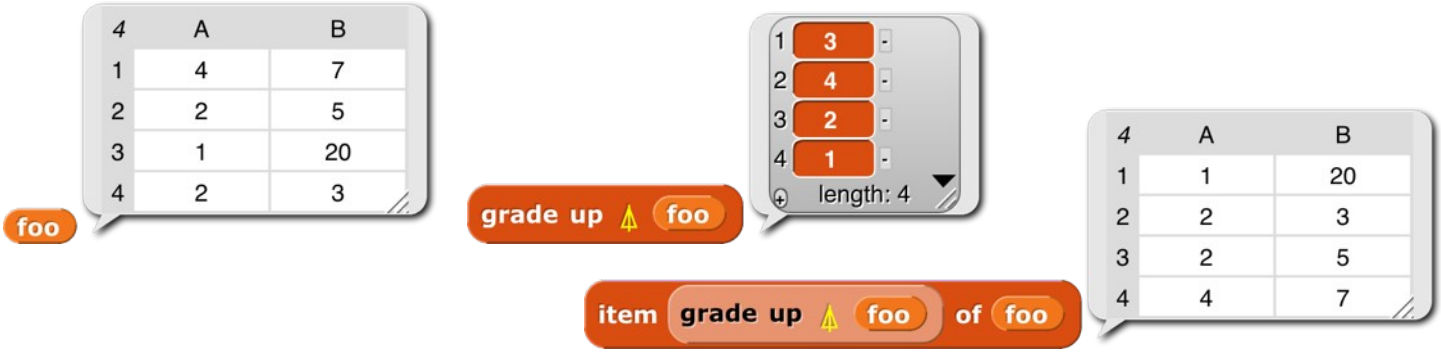




Note that **code** has 27 items, not 26. The asterisk at the end is the ciphertext is the translation alphabet characters (spaces and the apostrophe in "doesn't"). This is a silly example, because random cipher every time it's called, and it doesn't report the cipher, so the recipient can't deci message. And you wouldn't want to make the spaces in the message so obvious. But despite b 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 a Booleans the same shape as the left input. The shape of the right input doesn't matter; the blo atomic elements.

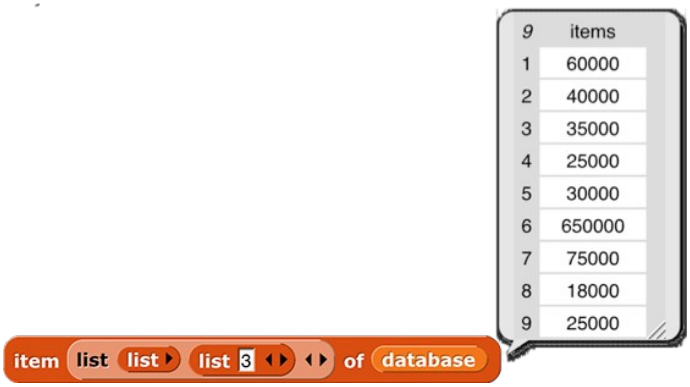
The blocks **grade up** and **grade down** are used for sorting data. Given an array as input, it re the indices in which the items (the rows, if a matrix) should be rearranged in order to be sorted. clearer with an example:



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

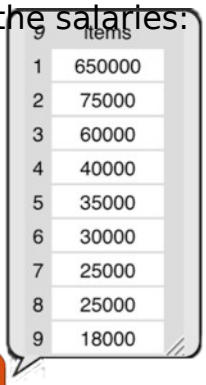
Why this two-step process? Why not just have a **sort** primitive in APL? One answer is that in a application you might want to sort one array based on the order of another array:

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



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 columns we set out to find. As usual the code is more elegant in APL: **database[⍋database[;3];1]**.

In case you've forgo 

or 

would select the third *row* of the database; we need the **list 3** in the second input slot of the ou columns rather than by rows.

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

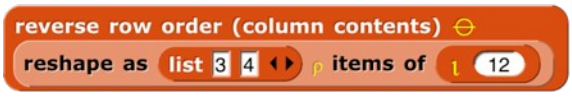
The **compress** block selects a subset of its right input based on the Boolean values in its left in be a vector of Booleans whose length equals the length of the array (the number of rows, for a right input. The block reports an array of the same rank as the right input, but containing only t whose corresponding Boolean value is **true**. The column 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 abou Take the standard **/** version, to avoid talking about both at once. One way to think about it is th some of the rows. The other way is that it shortens the columns. For Lispians, which includes y learned about **keep**, the natural way to think about **/** is that it keeps some of the rows. Since w matrix as a list of rows, that also fits with how this function is implemented. (Read the code; yo inside.) But APL people think about it the other way, so when you read APL documentation, **/** is operating on the last dimension (the columns), described as operating on rows. We were more th month into this project before I understood all this. You get long block names so it won't take y

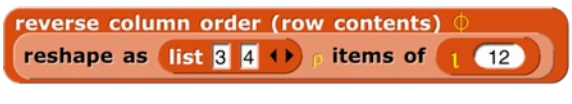
Don't confuse this block with the **reduce** block, whose APL symbol is also a slash. In that bloc the left of the slash is a dyadic combining function; it's the APL equivalent of **combine**. This bl nearly equivalent to **keep**. But **keep** takes a predicate function as input, and calls the function the second input. With **compress**, the predicate function, if any, has already been called on all right input in parallel, resulting in a vector of Boolean values. This is a typical APL move; since equivalent to an implicit **map**, it's easy to make the vector of Booleans, because any scalar fun predicates, can be applied to a list instead of to a scalar. The reason both blocks use the / char 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 gro

matrix. The APL symbols are all a circle with a line through it; the lines are the different axes of the **reverse row order** block reverses which row is where; the **reverse column order** block r 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

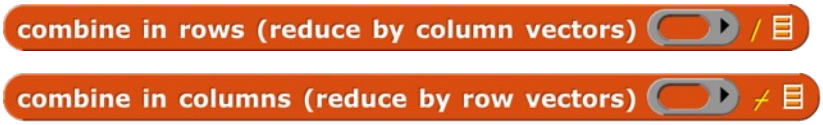
Except for **reverse row order**, these work only on full arrays, not ragged-right lists of lists, be the other two would be an array in which some rows had “holes”: items 1 and 3 exist, but not it have a representation for that. (In APL, all arrays are full, so it’s even more restrictive.)

Higher order functions

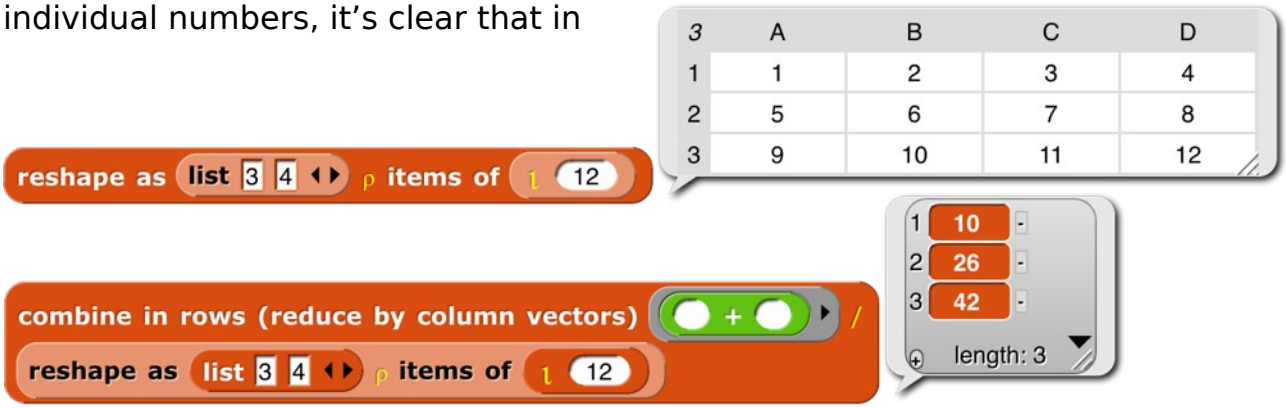
The final category of function is operators—APL higher order functions. APL has no explicit **ma** because the hyperblock capability serves much the same need. But APL2 did add an explicit **m** might get around to adding to the library next time around. Its symbol is ¨ (diaeresis or umlaut

The APL equivalent of **keep** is **compress**, but it’s not a higher order function. You create a ve (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 tra to a single value; the **!** version translates each column to a single value. That’s the only way to thin from the perspective of combining individual elements: you are adding up, or whatever the func numbers in a single row (**/**) or in a single column (**!**). But APLers think of a matrix as made up of vectors, row vectors or column vectors. And if you think of what these blocks do as adding vectors, rath 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). I the same result this way:

mapping over the *rows* of the matrix, applying **combine** to each row. Combining rows, reducin

The **outer product** block takes two arrays (vectors, typically) and a dyadic scalar function as in an array whose rank is the sum of the ranks of the inputs (so, typically a matrix), in which each of applying the function to an atomic element of each array. The third element of the second ro the value reported by the function with the second element of the left input and the third eleme input. (The APL symbol ∘. is pronounced “jot dot.”) The way to think about this block is “multiplicat 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 colu matrix must equal the number of rows in the right matrix. When the two operations are + and 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 a compact representation that still makes the structure clear:

Experts will recognize this as the Lisp representation of list structure,

Index

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