

## Building a Microworld in Snap!

Zak Kolar, Hannah Moser, Paul Goldenberg

With the hopes of encouraging others to design and build their own microworlds (MWs), we publicly share the [technical tools](#) we developed for our own, and document them here. This documentation will be updated from time to time as Snap! and/or our MW ideas evolve.

Our project was funded by the National Science Foundation. The underlying idea was to build MWs that would let children ages 7–11, *in their regular school math classes* and *with their regular teachers*, use programming as a language for expressing and exploring the mathematics they are learning. The “deliverable,” the product that NSF sought, was *knowledge* of how to meet those constraints—how to design so that children are engaged and learning and so that teachers see the value and use it. Part of that knowledge is what we learned about pedagogical design (see Spencer, et al. 2023 and Goldenberg et al. 2023, summarized briefly below to clarify the rationale for our tools) and some consequences for the look and feel of MWs that meet the needs. The other part of that knowledge is what we learned about technical design that would allow others to build their own MWs with similar goals. Here, we focus solely on the latter, the structure and programming details common to all of our MWs, with the aim of sharing the tools.

### 1. Design principles

Children play and generalize from what they learn. Mathematicians, too. When facing a problem they’re trying to solve, mathematicians often tinker for a while, looking for pattern and structure and behavior from which they can generalize. Only when the problem is well understood do they reduce the results to a neat path that hides the messy detours and dead ends. Our work in mathematics curriculum design is grounded on that experience-before-formality approach. The NSF challenge was to design tools for use *in school* in regular classrooms and not by specialists, whence the need for easy entry for both teachers and students. It is not to be tutorial. It is not to evaluate children’s results as right or wrong: locus of control and assessment of results must remain with the child. It is not to coach or guide. Its mathematical focus and goals must be easily recognizable as “on target” by a teacher. It must be playful and engaging, with opportunities for accidental discovery. It uses *programming* as a language for the children to do their

mathematics,<sup>1</sup> with the “product” not being a written answer but a working program. The programming must make the mathematics learning *easier*, not be an extra burden or distraction for students or teachers. And the entire experience must be accessible in multiple languages. Currently, all our MWs are accessible in English, Spanish, and Brazilian Portuguese; many are accessible in German and Bulgarian, and a few in Ukrainian. One design principle is to build the technical tools in such a way that new translations (e.g., into Italian, Greek, or Russian—all pending) require no more effort than adding the new translations to an existing text file.

## 2. Impact on evolution of the technical tools

When we first proposed this idea to the NSF, we expected to develop the MWs in Scratch. ScratchMaths provided an excellent model for the curricular ideas and materials (Benton, et al., 2016, 2017; Noss & Hoyles, 2018), and the increasing spread of Scratch in schools satisfied our concern that the new work feel familiar and accessible to *teachers* as well as to a growing number of children. But by the time we began the project, we had realized that Snap! would suit our work better. The fact that it *looked* so much like Scratch satisfied the criterion of familiarity and acceptability, but it offered us—and, ultimately, the children as well—features and power that were needed and that Scratch did not have. Though cosmetically similar, the underlying capabilities (and some aspects of the UI) of the two languages were optimized for different purposes. Scratch is optimized for stage-plays—whence the name “stage” for the region where the action is seen—creative and interactive animation of sprites with varied costumes. Its visual and sound effects are well developed, and the stage is “bounded” so that a sprite can’t easily get lost off the stage. It allows the user to detect collisions of sprites with each other, with the edge, and with colors that the user specifies, and to compute distance (and other) measurements on the stage, giving the user many exciting options including building games and drawing complex figures accurately. But a language optimized for mathematical learning must let users build new mathematical functions, which Scratch did not. Even the protection that kept sprites from getting lost off the stage caused geometric algorithms to behave differently near the edge of the stage, not functioning truly as algorithms. So we moved to Snap!

---

<sup>1</sup> For the rationale for using programming as a mathematical language for young children—and why we felt young learners *needed* an extra language—see Goldenberg, et al., 2019, 2020, 2021a, 2021b.

One way to think about MW design is to see MWs as labs for research scientists. Compared to all that can be studied, research scientists look at a tiny slice. And they need specialized tools. Their work/lab is both *micro*—a tiny domain, not the entire universe—and a *world* broad enough to explore, complete with tools suited for exploring *that* domain. Research scientists must retain the childlike creativity and openness to ideas outside their immediate domain, and to serendipity. Yet adult scientists must develop the ability not to be distracted by too many of the interesting and curious things that can pop up mid-research. They need the ability to stay focused, a disposition that develops over time and is more adult than child. Children are explorers and experimenters, natural research scientists but without the focus and systematicity of adults.<sup>2</sup> So a programming MW for children must hide UI features that would be distracting, confusing, or defeating for children, or that would create “dangers” for a teacher, situations that children could create from which the teacher cannot rescue them. Regardless of a teacher’s Snap! knowledge or expertise, the teacher is outnumbered; teachers can’t be in twenty places at once to save children from unexpected or unwanted traps.

Snap! is ideal for our mathematical microworlds. It offers children the capabilities of an exceptionally powerful language, including sophisticated mathematical tools, and lets us, as designers, simplify the UI to avoid distractions and traps and to provide only what is needed.

Of course, at the outset, we had only a blurry glimmer of what we wanted to build or what features it needed to have or avoid. Despite our best planning, our seven-year-olds routinely demolished our early trial MWs, teaching us how much more we should have protected against—super-quick clicking, accidental escapes from the MW, accidental switches to a different sprite, typing something our program didn’t understand, making numbers so big that Snap! responded in mysterious scientific notation and *dozens* more. The children also taught us

---

<sup>2</sup> For children, this is *good* (meaning evolutionarily adaptive). Our species’ particular evolutionary advantage is not speed or claws. What makes our species adaptive is its ability to be born anywhere—hot, cold, dry, wet—and not be already preprogrammed to have been born somewhere else. That requires that, as children, our attention is all over the place, *not* overly focused, as we do not yet know what features of our world will be the important ones to attend to, the ones we will need to know deeply to keep ourselves well and safe. By contrast, adults of our species need to know how to ignore distractions—knowing both what they *can* ignore and being skilled at blotting it out—in order to blow-dart that rabbit while not being eaten by that tiger. Wild fantasy, creativity and (non-extreme) attention deficit are advantageous in youth and diminish as humans “grow up.” We often blame schools for squashing children’s curiosity and creativity, but the truth is a bit more moderate. That process of “loss” is evolutionarily built in. When school *accelerates* the loss (disadvantageously), it’s because it’s so natural for adults to teach kids the way *adults* learn most efficiently, not in the way kids naturally learn.

quickly how much more we should have *allowed*, what creative ideas they would bring that we hadn't counted on but could (with careful programming) capitalize on without sacrificing the simplicity or focus that MWs need.

Because we were still tinkering to improve our design and then re-try it with children, our design process had to be flexible and allow quick turnaround. Bernat Romagosa designed a set of MW-building blocks with which we could easily modify UI features in Snap! 5 until we had a good first approximation. Our team kept in regular touch with Jens Mönig and Bernat, and a few of our needs suggested features that they deemed useful beyond the task of designing MWs, features that they then incorporated directly into new releases of Snap! When we had a clearer idea of what we were doing, Bernat designed a specialized version of Snap!, a separate fork, further simplifying our work in designing MWs. At about this time, Zak Kolar joined the team and started adding to or tweaking the modifier blocks and the underlying Snap! design.

Our current UI (Fig. 1 shows an example from our fraction number line MW) has continued to evolve over 6 years of considerable experimentation with ~700 children in 7 schools (3 districts). No doubt it would evolve even further simplicity if our project had more time.

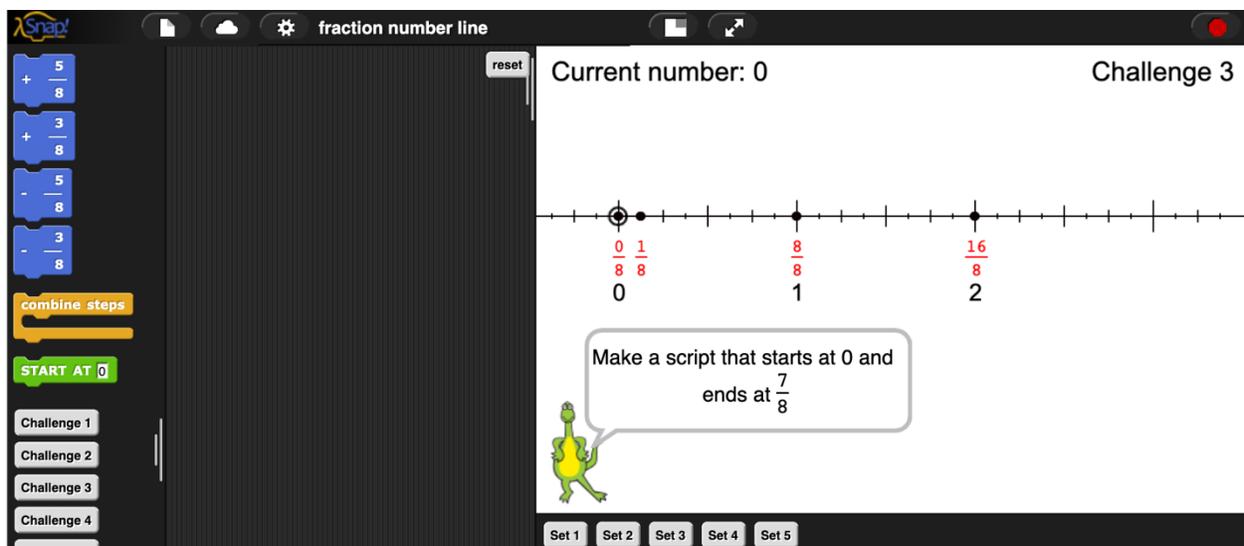


Figure 1: Header with Snap! logo, three menus (file, login, limited settings), the MW title, and the stop button; palette (left) with blocks and with buttons for selecting puzzles; scripting area (middle) with reset button at top right; and stage tailored to that microworld (right) with buttons below for selecting sets of puzzles (e.g., from explorations through challenges).

Nothing else.

When the Snap! team invented JS extensions, Zak undertook to rebuild our MW base from scratch as an extension. This freed us from the need for a separate, unevolving Snap! fork. The

extension code must still be maintained<sup>3</sup> as Snap! evolves, but would be imported automatically with any MW, increasing the stability of MWs. For security reasons, Snap! does not allow JavaScript extensions to run when a user loads a Snap! project from a link unless the source is trusted (on the Snap! “allowed extensions” list). Our domain, microworld.edc.org, is on that list, so loading MWs from our domain starts it automatically, requiring no extra step. MWs that others create will be stored in their Snap! accounts and so (unless they also get their source whitelisted), any user of such a shared file must first approve running it by checking JavaScript extensions in the settings menu  and must then click the green flag.

Control-clicking the Snap! icon  allows (regular-click) Escape microworld (revealing our full scripts in raw Snap!) and re-entry to the MW. Clicking  restarts the MW from scratch.

The rest of this paper documents the Snap! scripts that are the skeleton and vital organs of every MW, the special blocks required for the scripts, the places where scripts allow (or require) tailoring to suit a specific MW, and the details of language translation.

### 3. The two elements of a Microworld

A MW consists of two elements: a specialized programming language that allows rich exploration in a specific domain; and a UI that makes the necessary tools easy to find and minimizes distractions. Our project chose mathematics as its domain; other microworlds have been conceived for music, art, elementary (and more advanced) linguistics....

In a MW for exploring various representations of rational numbers, perhaps on a number line, the *programming language* component might present blocks like the green ones you see in Fig. 2 and any behind-the-scenes machinery that might be needed for checking validity of inputs. The choice of user blocks, the way results are displayed, and the UI depend on the MW’s purpose. A different fractions MW (see Fig. 1) might contain different blocks, behaviors, and a behind-the-scenes engine for drawing number lines and displaying results on the stage.

---

<sup>3</sup> This is a challenge we have *not* solved. NSF projects end. Support for maintaining the extension must be found.

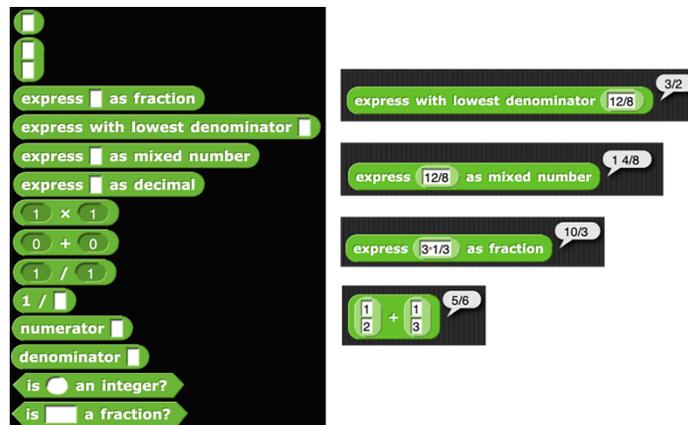


Figure 2: A possible set of programming tools for exploring fraction equivalence, operations, and expressions.

Because a MW is a *complete* programming language for the limited micro-domain, child or adult explorers can program flexibly within the limits. The UI that surrounds the micro-language can also present suggestions for exploration. We think of these not as exercises or “problems”—life already has too many problems—but as puzzles, things that require some thought (suited to the expected audience) and that are satisfying to solve. Though we design both the nature and sequence of the puzzles for learning-value and coherence, kids in all grades differ in how quickly they stray from our puzzles or sequence to pursue their own explorations. Nearly all students do at least some exploring on their own as well as most of our puzzles, a success in our view!

The language you design must also account for the knowledge and skill of the expected user. For skilled adults, a programming language generally does not account for mistyped inputs; the programmer is assumed responsible for finding the error. But if we are creating a MW for learners, we probably want to be as accepting of interpretable inputs as possible, even if their form does not follow convention, like extra spaces  $1\ 1/2$ , and totally acceptable forms like  $3\ 1/2$  that we may not have anticipated, but that children create, rather than letting these be treated as errors. The language itself requires careful design, observation, and redesign.

#### 4. Managing the UI: The structure of the Microworld Base

The images and examples in this documentation are drawn from our [Coordinates](#) MW. Our design typical uses at least four sprites (Fig. 3): User, Puzzler, MW base, and one MW-specific sprite—in our example, named Coord MW to identify the MW for which it is designed—that the designer must tailor to manage the UI features (blocks, buttons, puzzles, menus, etc.). The MW-

specific sprite must have MW base assigned as its parent. Some specialized tasks common to several MWs are worth assigning to a separate sprite that can be imported as needed without rewriting the code (e.g., displaying and [managing a chart](#) in the [Multiplication](#) and [Who Are We](#) MWs, or a [list](#) in Multiplication, or simply bearing a common costume as the Compass sprite does for [Map](#) and Coordinates). Sometimes an extra sprite is useful just as a convenience to keep the operating of a MW clear (e.g., the CoordRep sprite in Coordinates, which has no blocks or scripts of its own, but moves on the grid to report information to the user at appropriate spots).



Figure 3: Sprites in Coordinates microworld

The four universal sprites and their scripts are described in the next section. Following that, we separately describe the language support tools that allow the designer to manage translations of all text—puzzle text, block titles, block variable’s defaults and dropdown menus, messages that a block might give to a user (e.g., about errors)—from one set of translations, avoiding the need to handle updates to each feature every time a new language translation is added.

## 4.1 The scripts of the four universal sprites and the Stage

This section describes the scripts of the four universal sprites and the Stage, the most elaborate of which is the MW-specific sprite that manages the UI and that the MW-designer must tailor to the needs of the specific MW under construction. Our goal in this documentation is intended to allow you to copy the structure (and modify as needed) to design your own microworlds.

### 4.1.1 MW base

Most specialized blocks for running a generic MW are local to this sprite. It has no scripts of its own and serves only as parent to the MW-specific sprite (in our example, Coord MW).

**User Sprite:** The User Sprite’s essential purpose is to be the user’s vantage point. The sprite may be hidden or the designer can choose to assign it additional functions, like being the “Smiley” that moves around in the Coordinates microworld or the “Painter” that paints rows in the [Array MW](#). Whatever other functions this sprite might serve, it must have no scripts of its own so that the scripting area is completely empty awaiting the user’s work.

What the user sees from this vantage point—for example, the palette of available blocks—is controlled entirely by the MW-specific sprite. The palette must offer all (and only) the blocks that the user will need for creating scripts in that MW. Those blocks are best defined as global. The user never interacts directly with any other sprite or sees any of their scripts.

**MW Stage:** On entering any MW the stage shows the “loading screen” with two buttons below.



Figure 4: What a user sees when the microworld first starts.

The stage, itself has a single script (Fig. 5) that broadcasts stage clicked which tells the MW-specific sprite to close that intro banner (see Fig. 13) and get ready for puzzles.



Figure 5: Stage script for dismissing title screen.

The stage costume is generally empty but can be modified as needed.

**Puzzler sprite:** The Puzzler is the sprite that displays the puzzle text in any available language. In Figure 1, it wears the dinosaur costume. It may have other costumes to illustrate scripts to build and use or designs to draw or modify. If needed, costumes can be 2- to 4-second animations to help show a part of a script being built or illustrate some other action that helps supplement verbal description.<sup>4</sup> Costumes that are scripts can be automatically translated to whatever language has been selected (see section 4.1.2). The Puzzler sprite’s one script (Fig. 6) supports accessibility by letting users click on the sprite or its text to have that text read out loud.

---

<sup>4</sup> We designed this feature but have not yet implemented it in any MW. It will be documented in future update.



Figure 6: The script in Puzzler’s scripting area. The broadcast is processed by the MW sprite (in this example, Coord MW).

**MW-Specific Sprite:** This sprite owns ten or so scripts—all the mechanism to manage the UI, all the special scripts to run its *specific* MW, and all the generic scripts needed for running *any* MW. Each essential script is described below, using images/examples from Coord MW.

→ green flag **script** (Fig. 7) and change language **script** (Fig. 8): A microworld can run only with JavaScript extensions enabled. Running the green flag script (Fig. 7) starts the MW. This script is run when  or the translate message (Fig. 8) is received, and runs automatically when the MW is loaded via a shared link (e.g., [Who-are-we?](http://Who-are-we?)) to [microworld.edc.org](http://microworld.edc.org) (or another trusted source, whitelisted for JS extensions). If opened from one’s own file menu, or by dragging in a project file, or is loaded from a shared link from a non-whitelisted source, one must check JavaScript extensions in the settings menu  and then  to run it.

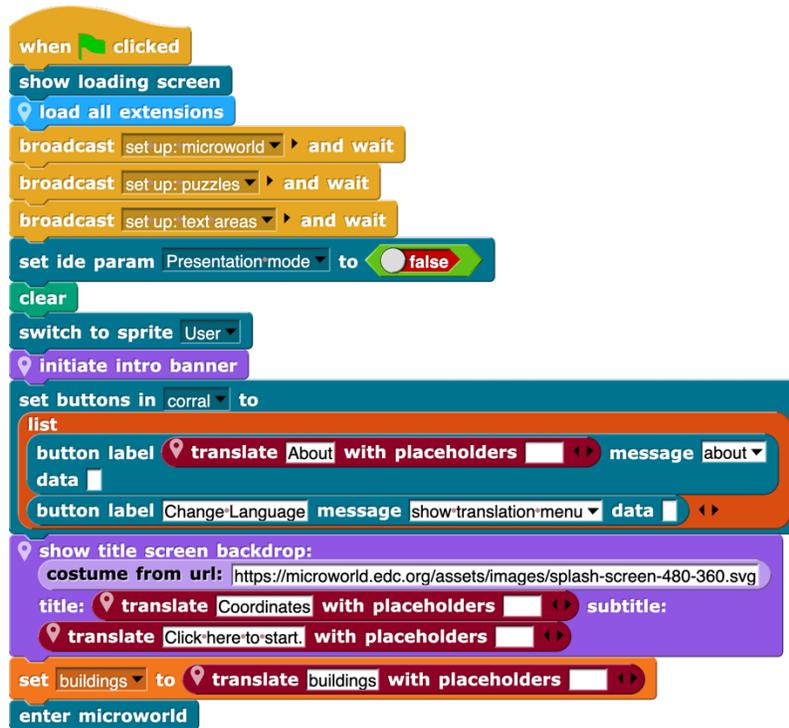


Figure 7: The “go” script.

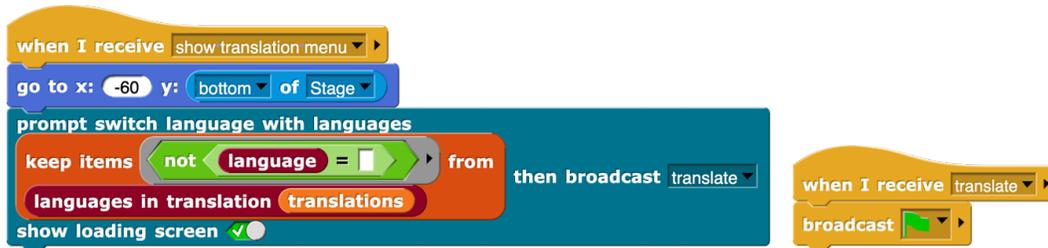


Figure 8: Pressing the Change Language button below the stage shows a menu of available translations (8a). Clicking that menu or switching languages using  broadcasts translate, restarting the MW (8b).

The first half of the “go” script needs no tailoring. It loads the extensions, broadcasts calls to three set-up scripts (described later), and sets Presentation mode to false giving the user access to the palette and scripting area and preparing for entering the MW or switching sprites. Initiating the intro banner hides blocks, sprites, and all UI features, clears the stage, and removes buttons.

The remainder is tailored to specific MWs. All of ours set two special buttons, About and Change Language, in the sprite corral and have a special screen backdrop with EDC and NSF logos and the title (in this case, “Coordinates Microworld”) and opening instruction (“Click here to start.”). The two maroon translate blocks automatically translate the title and opening instruction into whatever languages have been made available in the MW (see when I receive [set up: puzzles] script, Fig. 17, and section 4.1.2).

→ when I receive [set up: microworld] **script:** This script (Fig. 9) lets the designer specify UI features to which the user has access, including menu items and what blocks can be edited.

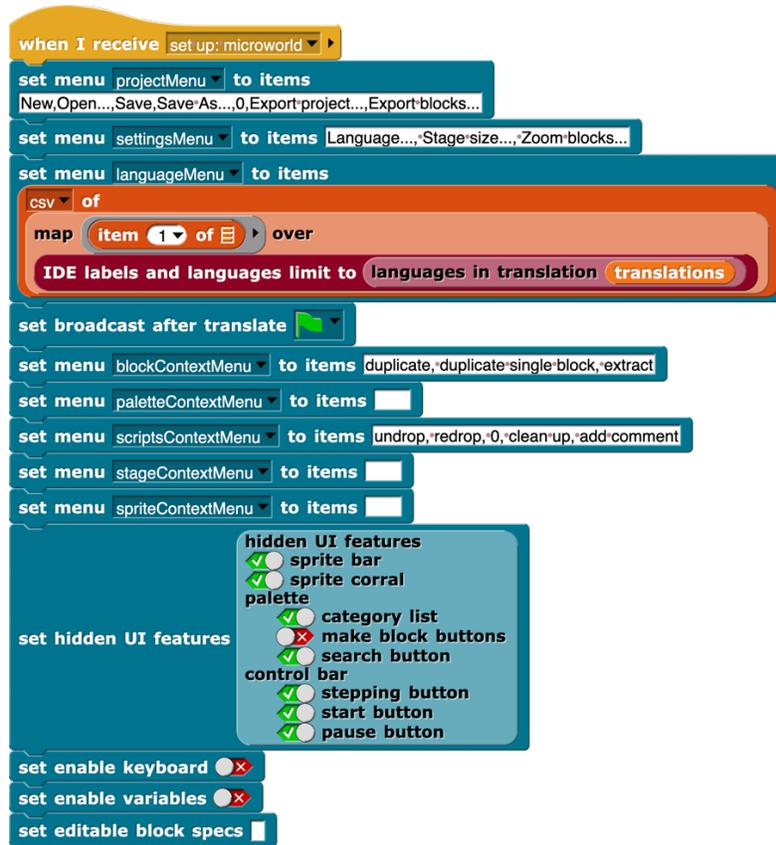


Figure 9: Setting up the microworld's UI.

In this example, the script limits the settingsMenu  to choosing a language, stage size, and zoom blocks. The languageMenu automatically limits to only those languages that the MW offers. If we wanted to allow the user to inspect or modify custom blocks that we provide, we would list their names in set editable block specs. The empty slot here indicates that *no* blocks (other than those created by the user) are editable by the user. Users can always edit blocks that they create.

→ when I receive [set up: puzzles] script: This script has three parts (Figs. 10–12). The load translations block (Fig. 10) converts a block of text from JSON format to a Snap! list and saves the list as **translations**. It includes the text of all puzzles, button names, the MW title, other special messages, and all their translations into whatever languages the MW currently has. Texts are grouped by language; adding a language requires only a section with the new translation. Each entry starts with a key (e.g., "Puzzle 1.text"), always in the default language (here, en). Actual puzzle (and other) texts follow in translation. Notations like "Exploration \$1": "Exploração \$1"



Figure 11: Updating block titles and input defaults, and any input options (drop-down menus) programmatically.

Finally, the  block (Fig. 12) loads all parameters for each puzzle in each set.

```

"repeat _": {
  "label": "repita _ vezes _"}
})}
update custom block translations from translations (translations)
translate input options with translations (translations) fallback en
load puzzle sets
[{"key": "Set 1", "puzzles": [
  {"key": "Exploration 1", "parameters": {"puzzler scale": 30, "puzzler costume": "dog2 b", "puzzler coords": [-212, -237], "referenced locations": [[3, 5]]},
  {"key": "Exploration 2", "parameters": {"puzzler scale": 30, "puzzler costume": "dog2 b", "puzzler coords": [-212, -237], "referenced locations": [[3, 5], [7, 1]]}},
  ...
  {"key": "Exploration 7", "parameters": {"puzzler scale": 30, "puzzler costume": "walk script 2", "puzzler coords": [-180, -227], "referenced locations": [[8, 4]]},
  {"key": "Set 2", "puzzles": [
  {"key": "Puzzle 1", "parameters": {"puzzler scale": 30, "puzzler costume": "dog2 b", "puzzler coords": [-212, -237], "referenced locations": [[2, 2], [11, 5], [5, 3],
  ...
  {"key": "Puzzle 6", "parameters": {"puzzler scale": 30, "puzzler costume": "dog2 b", "puzzler coords": [-212, -237], "referenced locations": [[3, 5], [5, 3], [2, 2]]},
  {"key": "Set 3", "puzzles": [
  {"key": "Challenge 1", "parameters": {"puzzler scale": 35, "puzzler costume": "distance from", "puzzler coords": [-168, -258], "referenced locations": [[0, 0]]},
  ...
  {"key": "Challenge 8", "parameters": {"puzzler scale": 60, "puzzler costume": "middle of block", "puzzler coords": [-212, -237], "referenced locations": [[8, 4], [
INITIAL DEFAULT STATE

```

Figure 12: The load puzzle sets block contains all specifications for the structure of puzzles. "key" marks the unique name of each puzzle set (e.g., Set 1) and each puzzle within a set (e.g., Exploration 2). Parameter names/values are as needed.

For each MW the designer specifies parameters that puzzles may need, things like the puzzler's costume, coordinates and size; special locations on the stage; specifications for the extent or kinds of numbers on a number line, or anything else specific to the puzzle. No parameters are inherent to the MW structure. The designer makes up a name for any needed parameter (e.g., "puzzler coords") and sets its value. (Fig. 12 shows an excerpt from Coordinates.) These specs and the puzzle text are used by the when I receive [puzzle] script (see Fig. 17). Setting initial default state is optional, mostly a container for designer to keep notes on default settings.

→ when I receive [set up: text areas] **and** when I receive [redraw stage labels] **scripts** (Figs. 13 and 14):

```

when I receive set up: text areas
  set eraser color r: 255 g: 255 b: 255
  text area called current puzzle size: 18 font: Arial align: right x:
    right of Stage - 10 y: top of Stage - 10 color: rgb(0,0,0)
  text area called current coordinates size: 18 font: Arial align: left x:
    left of Stage + 10 y: top of Stage - 10 color: rgb(0,0,0)

```

Figure 13: Each required text area gets its own text area called block.

This script (Fig. 13) reserves locations on the stage for the puzzle label (e.g., Challenge 5) or other information needed for a particular MW (e.g., Current number on number lines). The eraser color is the color of the stage background. These areas are created once when the MW is set up.

The redraw stage labels script (Fig. 14) refreshes these text areas whenever it is called, usually by the when I receive [puzzle] script but potentially in other ways like a reset button that the designer may invent. It needs individual write text in text area blocks for each text area that must be refreshed.

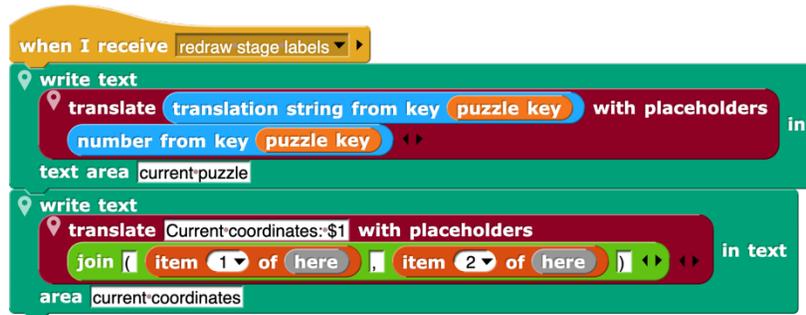


Figure 14: Each text area to refresh needs its own write text [] in text area [] block, with language-translation information.

→ when I receive [stage clicked] **script:** Launching a MW shows its title and other startup information. Clicking the stage broadcasts a message which this script (see Fig. 15) acts on only if the title-screen/intro-banner is still displayed. Its blocks include elements needed in all MWs, like **close intro banner**, **set up puzzle set buttons** which creates buttons below the stage for choosing puzzle sets and **broadcast puzzle set**, which prepares for a new puzzle set (Fig. 16). Otherwise, the entire script is designer-customized. It’s a good place to establish any settings that don’t vary across puzzles or puzzle-sets, and to create unchanging extra buttons, like reset or undo. In the Coordinates MW specifications (Fig. 12), puzzler size and costume are given because they do vary from puzzle to puzzle. In such cases, there’s no need to specify here. If palette blocks don’t vary in the MW, they can be established here; otherwise in the when I receive [puzzle set] script or even more locally. The Coordinates MW includes a “frivolous” element—drawing a heart shape—that is executed only on February 14. The non-frivolous idea behind the frivolous element is to hint to students that they can *play* with coordinates to specify designs, not just required paths. Largely for the same purpose, we include the ability to rename buildings using **put name [ ] on building at (0,0)** and **build a building for JM's Pizza at (0,0)**. The *consequence* isn’t frivolous. Though explicitly used in only one or two puzzles to show kids what these blocks let children do, these blocks get lots of spontaneous extra use precisely because they *are* fun, giving children much more practice with coordinates than they could get with our puzzles alone.

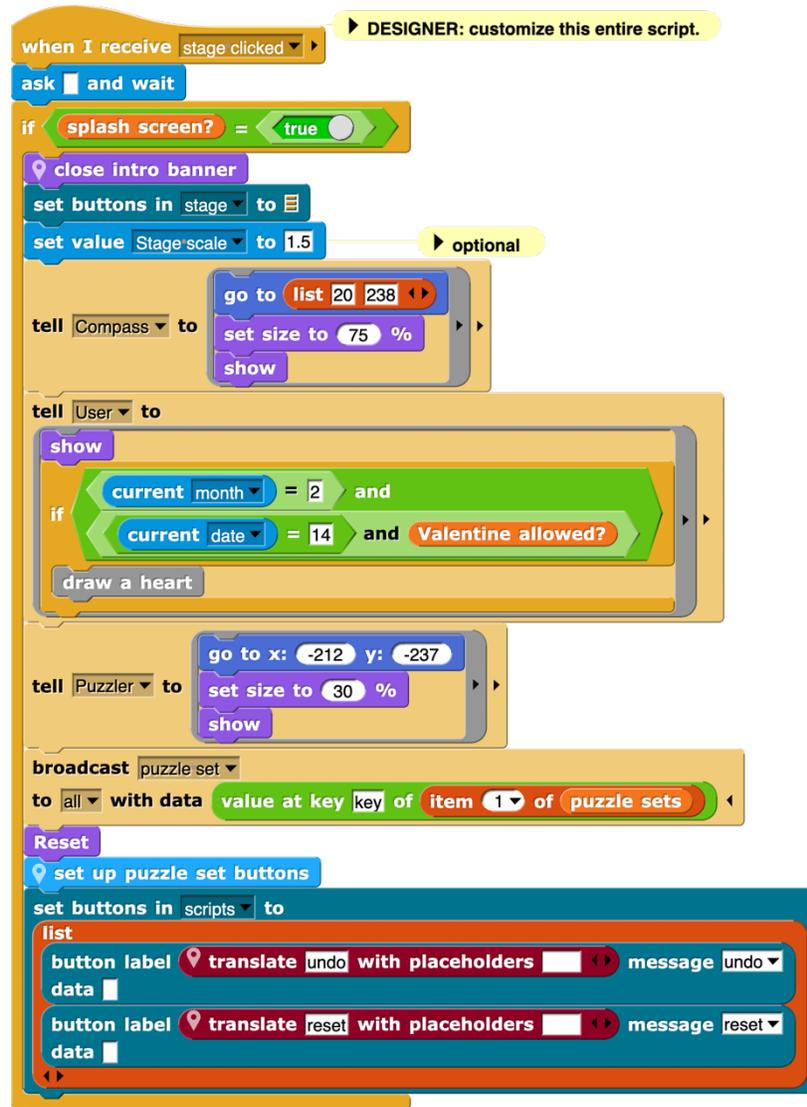


Figure 15: Essential elements of stage-click script are close intro banner, set up puzzle set buttons, and broadcast [puzzle set].

Designer optionally customizes the script with other elements.

→ when I receive [puzzle set] **script:** The first three blocks and the last block must be exactly as shown in Fig. 16. It uses puzzle specs (Fig. 12) to set buttons in the palette for each puzzle in the current set. It then launches the processing of the first puzzle. In this case, the nested ifs also conditionally choose what blocks to offer in the user’s palette.

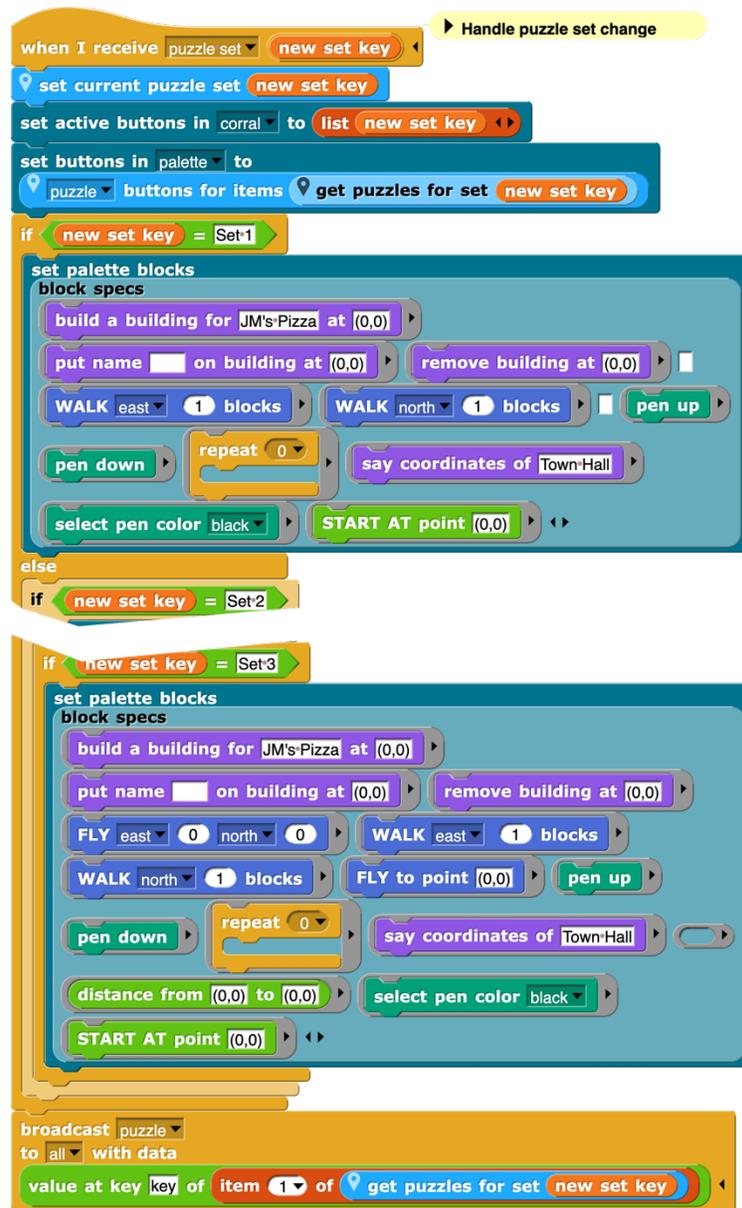


Figure 16: The required elements are the first three blocks of this script, which create the puzzle buttons in the palette, and the last block, which launches the first puzzle in the current set. In this particular MW, the user's choice of blocks varies depending on the set, handled by the nested if blocks.

→ when I receive [puzzle] **script:** The set current puzzle block usually has only the first input (new puzzle key) but in this case needs to be able to replace some text (the building names, which users can change) and so gets a list of placeholders. In this MW, set hidden UI features unhides the make a block button. The rest of this script is entirely designer-specified to use the puzzle specs to set up any needed framework. If scripts are to be displayed for the user to explore, as in this MW,

they must be translatable to the available languages. To do this, those scripts are put into a costume list consisting of two-element lists containing a script name and the ringified script.

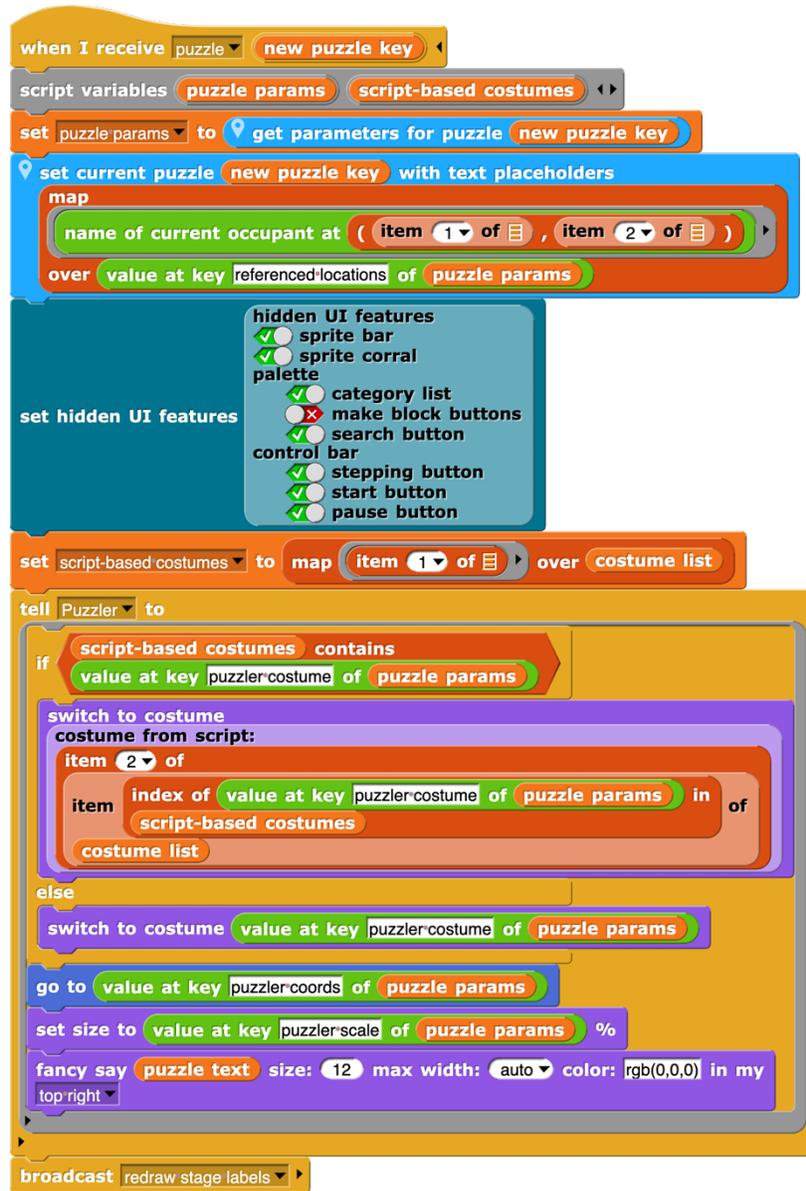


Figure 17: Running a puzzle.

If the puzzle’s parameters call for one of these costumes, the purple costume from script block translates the script before it is assigned to the Puzzler sprite. Then any remaining instructions needed for presenting the puzzle are given. The last block, required, redraws stage labels. The green value at key blocks indicate which puzzle parameter (Fig. 12) to use and what to do with it. The **puzzle text** for each puzzle is already set to the desired language translation (see translation priorities where sound is discussed after Fig. 20) so fancy say can say it just as is.

→ when I receive [reset] **script:** Optional UI buttons like reset can be created (see final block in the script in Fig. 15) to broadcast a message. A script that receives a reset message might simply restart the puzzle (Fig. 18a) or might call a custom block (b, c) that performs some special action.



Figure 18: Optional UI buttons like reset or undo can be created. Scripts like these respond when they are clicked.

### 4.1.2 Translation: language support and speech

Accessibility in multiple languages is a priority for us. Clicking the Change Language button (Fig. 7) displays a menu of the available languages for the current MW and lets the user choose. If a language is chosen, it restarts the MW in that language.

#### → Translating Blocks

Block translations follow other text in each language in the translation JSON under the key `_blocks`. This key exists at the same level as regular translation strings. Example:

```

{
  "en": {
    "Exploration 1.text": "Make a script that does...",
    "_blocks": {
      "WALK__blocks": {
        "inputs": {"direction": {"default": "east (x)", "options": ["east (x)", "west (x)", "north (y)", "south (y)"]}}
      }
    }
  },
  "es": {
    "Exploration 1.text": "Haga un script que...",
    "_blocks": {
      "WALK__bloques": {
        "label": "CAMINA __ bloques",
        "inputs": {"direction": {"default": "este (x)", "options": ["este (x)", "oeste (x)", "norte (y)", "sur (y)"]}}
      }
    }
  }
}

```

Figure 19: Syntax of translation information.

Within the `_blocks` key, each block gets its own entry using the label as the key.

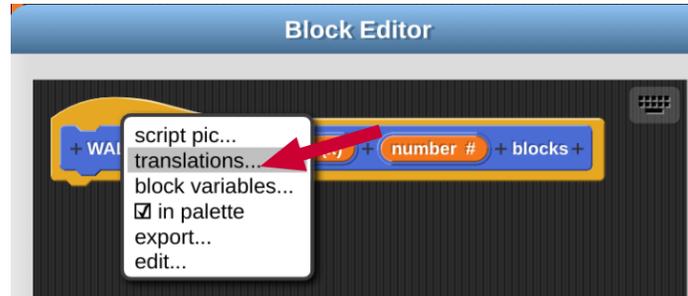
The label is the exact text of the block title with all inputs replaced by underscores (`_`). Icons, line breaks, and other formatting characters must be included. If you don't know the label for a

block, you can retrieve it using `label of block` `WALK east (x) blocks` `WALK_-$-1_ blocks`.

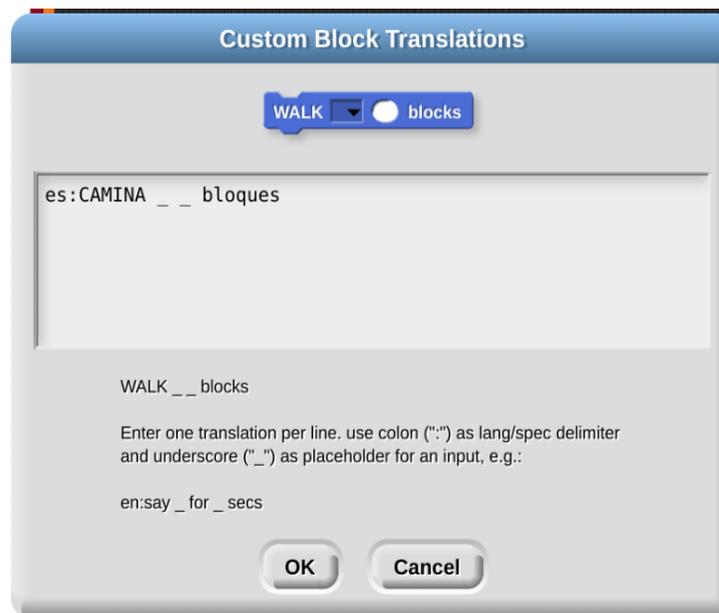
In this case, an extra space has been added to the block title to distinguish it from another block with a visually similar title (the WALK north/south block).

## → Translating labels (block title text)

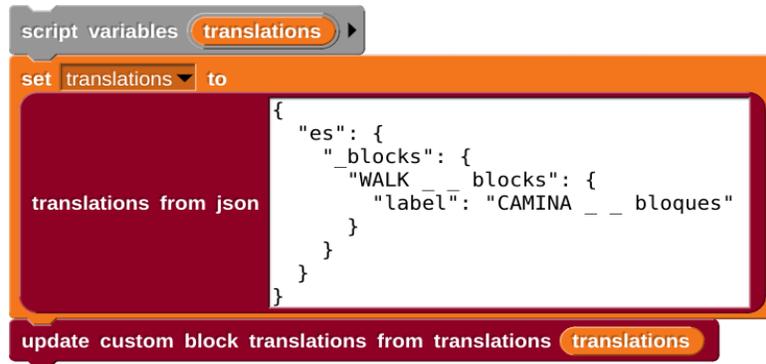
Snap! Includes a built-in capability to translate custom blocks. This UI can be accessed via control + clicking (or right clicking) on the hat block in the block editor:



Once the translation tool is opened, translations you can add translations in the specified format:



The translation extension allows a developer to programmatically update Snap!'s built-in translations by adding them to the `_blocks` section of the translation JSON. Within each language, you can specify the translated label (see Fig. 18). To apply translated labels to blocks, use the **update custom block translations from translations**  block:



Because translations are stored as part of the blocks' definitions in Snap!, translated labels persist after the update block is run even if the project is saved and re-loaded. This means that, if desired, the load translations and update custom block translations blocks need to be run only once after labels are changed or new translations are added.

A label translation is **not** necessary in the default language. For example, if the default language for a project is English, the redundant label entry below can be skipped (although including it will not cause any issues):

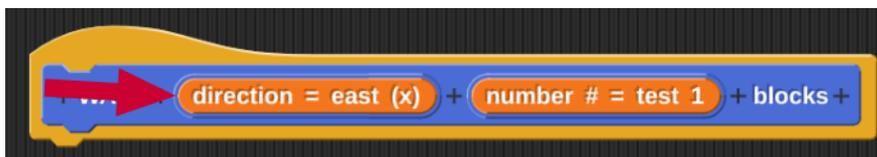
```
{
  ...
  "en": {
    ...
    "_blocks": {
      "WALK__blocks": {
        "label": "WALK__blocks",
        ...
      }
    }
  }
}
```

### → Translating dropdown menus

Translated dropdown menu options are not supported in raw Snap!, so this relies completely on the translation extension. Within each block's entry in the `_blocks` key, all dropdown-related translations are specified under the `inputs` key:

```
{
  ...
  "es": {
    ...
    "_blocks": {
      "WALK__blocks": {
        ...
        "inputs": {
          "direction": {"default": "este (x)", "options": ["este (x)", "oeste (x)", "norte (y)", "sur (y)"]} }
        }
      }
    }
  }
}
```

Each input's key should match the name of the input in the block editor.

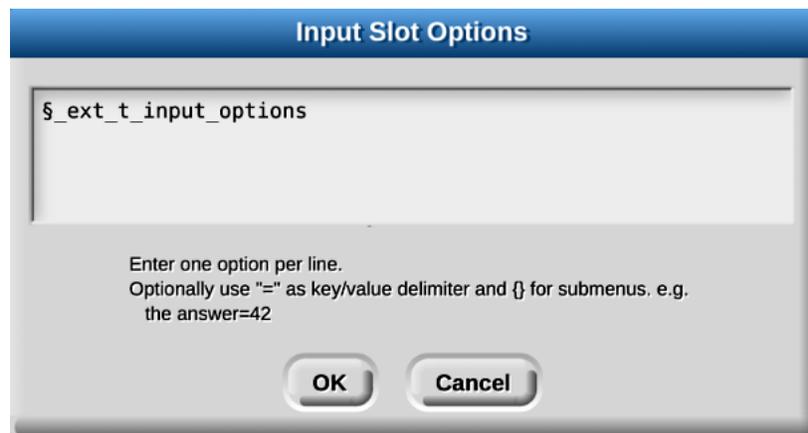


Each dropdown input contains two keys:

<b>options</b>	An array of options to display in the dropdown menu. These must be listed in the same relative order in each language.
<b>default</b>	The default value for the input. Just as in the Snap! block editor, this can be empty or a value not present in the dropdown.

Dropdown options and defaults *must* be specified in *all* languages, including the default language.

To add these translations to the blocks, first set the “options” for the input to `$_ext_t_input_options` using the block input editor:



All blocks and inputs should receive this exact same string, which tells Snap! to use the extension’s function to determine the options when populating the dropdown.

Then run the `translate input options with translations`  `fallback`  block. The fallback input is the language code that should be used if there is no translation for the current language set in Snap!’s IDE. In the example below, if Snap! was set to German, the dropdown labels would display in English because no German translations are specified here and the fallback is English.

```

script variables translations
set translations to
translations from json
{
  "en": {
    "_blocks": {
      "WALK __ blocks": {
        "inputs": {
          "direction": {
            "options": ["east (x)", "west (x)", "north (y)", "south (y)"]
          },
          "default": "east (x)"
        }
      }
    }
  },
  "es": {
    "_blocks": {
      "WALK __ blocks": {
        "inputs": {
          "direction": {
            "options": ["este (x)", "oeste (x)", "norte (y)", "sur (y)"]
          },
          "default": "este (x)"
        }
      }
    }
  }
}
translate input options with translations translations fallback en

```

The translate input options block must be run *every time the project is loaded*, including after the language setting in Snap! changes.

### → Checking user input

Because the dropdown option could be selected in multiple languages, testing user input using the  block will not work, unless the designer manually tests for the translation in each language. To avoid changing the block definition each time a language is added, use the

 block to test for equality:

The screenshot shows a Snap! script editor. At the top, there are tabs for 'script variables' and 'translations'. Below the tabs, a 'set translations to' block is visible. The main script area contains a 'say' block with a dropdown menu set to 'este(x) translates to east(x) from translations translations' and a 'for 2 secs' block below it. The script area is filled with a JSON object defining translations for English and Spanish.

```

translations from json
{
  "en": {
    "_blocks": {
      "WALK __ blocks": {
        "inputs": {
          "direction": {
            "options": ["east (x)", "west (x)", "north (y)", "south (y)"]
          },
          "default": "east (x)"
        }
      }
    }
  },
  "es": {
    "_blocks": {
      "WALK __ blocks": {
        "inputs": {
          "direction": {
            "options": ["este (x)", "oeste (x)", "norte (y)", "sur (y)"]
          },
          "default": "este (x)"
        }
      }
    }
  }
}

```

The order of the choice inputs does not matter. In the above example,

dropdown choice east(x) translates to este(x) from translations translations would give the same result.

→ when I receive [play puzzle sound] **and** when I am [stopped] **scripts:** Children for whom reading the puzzle text is a distraction or barrier can click on Puzzler (Dino in Fig. 1) or its text to hear that text read aloud in the appropriate language. That sprite broadcasts (Fig. 6) play puzzle sound which the MW-specific sprite enacts (Fig. 17a). The second script (Fig. 17b) assures that speech synthesis stops when the red stop-sign (top right in the header bar) stops all other action. Other commands (like reset) can, if the designer wants, force a stop to all action including speech.

Any sound, including the read-aloud text, can be recorded and stored in the project and played from the recording. But the designer can alternatively use the browser's text-to-speech system to read the puzzle text in the selected language.



Figure 20: Playing and stopping speech synthesis.

**Play current puzzle** marches down the following list of priorities to decide what sound to play.

1. Look in the sounds tab in the child MW sprite (e.g., in Coord MW) for a recorded sound named [IDE language code].[puzzle key] (e.g. en.Puzzle 3).
2. Play a remote sound from the (optional) Remote Sounds list that matches [IDE language code].[puzzle key] (e.g. en.Puzzle 3)

If these fail, it switches to text-to-speech (TTS) using the following choices of text as input:

3. [puzzle key].audio (e.g., "Puzzle 3.audio") in the current IDE language
4. [puzzle key].text in the current IDE language
5. [puzzle key].audio in the fallback language (in ours, that is English, but the designer can override this by setting the child MW sprite's fallback language to any language code like de, bg, pt, pt\_BR, es)
6. [puzzle key].text in the fallback language (English)

Text translations also prioritize any IDE translation in the puzzle text (Fig. 9). If no translation has been supplied, the fallback language is used.

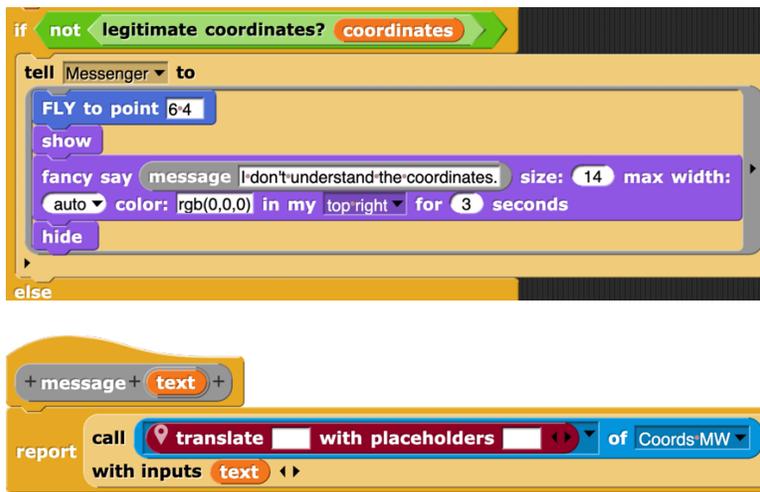
The .audio suffix in the translation files can be used if the text displayed in the speech bubble should be different from the read-aloud text (e.g., for text like fractions or decimals or money that is formatted one way in print but pronounced differently). Omit this specification if the speech bubble text suffices for the TTS.

The TTS follows the same priorities as the speech bubble translation, so the synthesized text should always match the speech bubble even in weird cases. For example, if the language is set to German but this particular puzzle is missing a translation, both the speech bubble and TTS will use the English puzzle text (or whatever the fallback language is).

The accent of the TTS engine will also attempt to match whatever language the input string is (whether IDE language or fallback language)

**Translation inside blocks:** Sometimes, the user's blocks also may send out messages that need to be translated. For example, in the Coordinates world, there's often the need to check

legitimacy of coordinates and tell the user if they aren't interpretable. The message needs translation, which is accomplished by this code.



## 4.2 Specialized sprites

Recall that the NSF funding was to generate knowledge about implementing an idea and set of principles, figuring out how to do it well, and then better, and what we can learn about learning from doing it as well as we can. In some real sense, it was well understood that we did not start out knowing what to do and have the luxury, then, of just writing the code to do it. So, after six years of evolving ideas, the code evolved, too. That included lots of good ends, but also lots of inconsistency and ad hoc solutions. When Hannah Moser joined the team, her focus was on bringing greater order to the Snap! coding both to create more modular code—code that could be inserted easily in any MW that needed it without reinvention or retrofitting—and to bring consistency to the evolution of code across similar MWs.

We had written code for one instance of a 0-to-99 number chart (for our Multiplication MW) before seeing other opportunities that suggested a more generic use of it. For the same MW, we had also built code to let children keep a list of the multiplication expressions they created and the resulting products, and to display, sort, remove duplicates, and clear that list. Again, we had a one-shot piece of code that we began to want to reuse in other MWs, and needed to make more generic and portable. Hannah started by creating separate sprites for these list and chart functions so that they could be imported whole and then used easily in any MW that needed them.

The **grid sprite** uses several local variables to keep track of the grid size and the padding on the stage. It offers the designer three new blocks: one to set up the grid in the first place; one to set a color calculation, allowing the designer to color numbers on that grid differently depending on some criteria that are relevant to the puzzle (e.g., numbers that are the results of students' expressions, numbers that are on some special list we want to highlight, etc.); and one to draw the grid and highlight certain cells specified in the parameters for the puzzles. The interface for designers is therefore quite easy. Working in the grid sprite, set size and padding once; set grid parameters once; and set color calculation function once (Fig 18). If there are numbers whose cells should be highlighted, they must be listed in the Puzzle specs (Fig. 22; see also Fig. 12).



Figure 21: Setting a color calculation. In this example, numbers that the user generates are printed black; others in gray.

```
{
  "key": "Exploration 10",
  "parameters": {
    "size": 50,
    "costume": "dinosaur1 d",
    "description": "11|n, n>10",
    "target numbers": [11, 22, 33, 44, 55, 66, 77, 88, 99]
  }
},
```

Figure 22: The target numbers will be assigned “required numbers” in the when I receive [puzzle] script (see Figs. 15 & 20).

The when I receive [puzzle] script always manages puzzle-specific actions, so this script assigns the required numbers and uses **draw grid and highlight required cells** to produce the grid (Fig. 23).



Figure 23: A section of the Puzzle script that uses the Grid sprite.

Hannah also created an importable sprite for list manipulations with the same easy interface to the designer's MW. To use these functions, the designer creates a mechanism—e.g., buttons that the MW-specific sprite creates that broadcasts the messages build my list of scripts and results, refresh list, sort my list, remove duplicates from my list, or clear my list—that let the user select the desired action. Designing new MWs can now be faster and more consistent.

We also built number lines for integers, fractions with various mixtures of denominators, and decimals. Although even the first of these—the integer line—was constructed with an eye toward a generic number line “engine” (an underlying structure that could be used in all three contexts), each incarnation of a number line suggested better ways of doing things, but we never had the time to update older number lines, and we sometimes forgot or overlooked good old ideas that had worked. Evolution is messy. Hannah built a consistent and more flexible number line engine for all these incarnations.

## 5. Invitation

Our project focused on designing for elementary school students ages 7 through 11 and evolved (largely through Zak) to work even with younger students. It was funded to investigate MWs focused on mathematics, but MWs could be designed for exploring languages one is learning in school (Goldenberg, 2019; Goldenberg & Feurzeig, 1987) or other spelling or other linguistic topics. Or music. Or designing games. Or just a first experience programming. At any age, people who are programming for the first time should have the chance to do interesting things on their first day rather than spending a week learning a UI and sifting through irrelevant and distracting features. We are eager to share this base for others to use and invent their own MWs.

## References

- Benton, L., Hoyles, C., Kalas, I., and Noss, R. (2016) Building mathematical knowledge with programming: Insights from the ScratchMaths project. In Sipitakiat, A., and Tutiyaphuengprasert, N. *Constructionism in action 2016. Conference proceedings*. Retrieved from <https://drive.google.com/file/d/0BwnqbVelN16LbXY3NHJZaldQY3c/view>
- Benton, L., Hoyles, C., Kalas, I., and Noss, R. (2017) Bridging primary programming and mathematics: Some findings of design research in England. *Digital Experiences in Mathematics Education*, 1–24. <https://doi.org/10.1007/s40751-017-0028-x>
- Goldenberg, E.P. (2019) Problem posing and creativity in elementary-school mathematics. *Constructivist Foundations* 14(3), 601–613. <https://constructivist.info/14/3/319.goldenberg.pdf>
- Goldenberg, E.P. and C.J. Carter. (2020) How programming can serve young children as a language for expressing and exploring mathematics in their classes. Pp. 347-356. In Tangney, B., Rowan Byrne, J. & Girvan, C., *Constructionism 2020*, The University of Dublin Trinity College Dublin, Ireland, May 26-29, TARA, 620pp. URI: <http://hdl.handle.net/2262/92768>
- Goldenberg, E.P., and C. J. Carter. (29 May 2021). Programming as a language for young children to express and explore mathematics in school. *Brit. J. Ed. Tech.* <https://bera-journals.onlinelibrary.wiley.com/doi/epdf/10.1111/bjet.13080>
- Goldenberg, E.P., Carter, C.J., Mark, J., Reed, K, Spencer, D., and K. Coleman. (2021) Programming as language and manipulative for second grade mathematics. *Digital Experiences in Math. Educ.* 7, 48-65 <https://doi.org/10.1007/s40751-020-00083-3>

Goldenberg, E.P., Carter, C.J., Mark, J., Reed, K., Spencer, D., Coleman, K., and K. Chiappinelli. (2023) *Constructionism 2023*. Programming Microworlds for Elementary School Mathematics: What we've learned.

Goldenberg, E.P. and W. Feurzeig. 1987. *Exploring Language with Logo*. Cambridge, MA: MIT Press.

Noss, R. and Hoyles, C. (2018) The ScratchMaths project is directed by Richard Noss and Celia Hoyles, with Ivan Kalaš, Laura Benton, Alison Clark Wilson, and Piers Saunders. See <http://www.ucl.ac.uk/ioe/research/projects/scratchmaths>. Accessed March 29, 2018.

Spencer, D., Mark, J., Reed, K., Goldenberg, E.P., Coleman, K., Chiappinelli, K., and Kolar, Z. (2023). Using Programming to Express Mathematical Ideas. *Mathematics Teacher: Learning and Teaching PK–12*, 116 (5), 322–329. <https://doi.org/10.5951/MTLT.2022.0354>