Extending a Reactive Expression Language with Data Update Actions for End-User Application Authoring

Lea Verou  
MIT CSAIL  
leaverou@mit.edu

Tarfah Alrashed  
MIT CSAIL  
tarfah@mit.edu

David Karger  
MIT CSAIL  
karger@mit.edu

ABSTRACT
Mavo is a small extension to the HTML language that empowers non-programmers to create simple web applications. Authors can mark up any normal HTML document with attributes that specify data elements that Mavo makes editable and persists. But while applications authored with Mavo allow users to edit individual data items, they do not offer any programmatic data actions that can act in customizable ways on large collections of data simultaneously or that modify data according to a computation. We explore an extension to the Mavo language that enables non-programmers to author these richer data update actions. We show that it lets authors create a more powerful set of applications than they could previously, while adding little additional complexity to the authoring process. Through user evaluations, we assess how closely our data update syntax matches how novice authors would instinctively express such actions, and how well they are able to use the syntax we provided.

Author Keywords
Web design; End-user programming; Information architecture; Semantic publishing; Dynamic Media; Web; Query languages; Data updates; Reactive Programming.

INTRODUCTION
Many systems and languages exist for assisting novice programmers to manage information, and/or create CRUD applications for this purpose. They range from the well known commercial spreadsheet systems to more complex application builders [8, 5] or simplified declarative languages [16, 2].

These usually generate an editing interface for elementary data manipulations (editing a data unit, inserting items, deleting items) and a mechanism for lightweight reactive data computation. As an example, in spreadsheets the editing interface is the grid itself, and the computation is the spreadsheet formula.

These tools typically offer only direct editing of specific data items by the end user. Affordances may also be provided for aggregating certain kinds of commonly needed mass modifications. A few examples of these would be selecting multiple items for deletion or move, adding multiple rows or columns, or the spreadsheet fill handle. However, the set of potential data mutations is infinite, and it is not practical to predefine controls for every possible case. For more complex automation of data edits, users are typically directed to scripting or SQL queries. Learning a scripting language is almost as hard as learning a programming language, and SQL queries quickly become complicated when nested schemas are involved, which are represented by multiple tables and foreign keys [6].

Mavo [16] is an HTML language extension for defining CRUD Web applications by annotating a static HTML mockup to separate UI from data. A property attribute indicates that an element is data, and an mv-multiple attribute makes it repeatable i.e. turns it into a collection. Based on this markup, Mavo generates a suitable editing interface. Mavo stores its data locally or on a cloud service.

Figure 1: The complete HTML for a fully-functional To-Do app made with Mavo, with a data update action for deleting completed items. No Javascript is needed.
Many applications benefit from presenting values computed from their data, so Mavo implements a reactive expression language called MavoScript, similar to what can be found in spreadsheets, for lightweight computation. Expressions can be placed anywhere in the HTML and are denoted by square brackets ([`) or certain attributes (mv-if, mv-value etc). An expression can reference properties, with its evaluation depending on the location of the expression relative to the referenced properties in the data tree. Referencing a multi-valued property on or inside a collection item resolves to its local value on that item, whereas referencing that property name outside the collection resolves to all values. All operators and most functions can be used with both single values and lists. This makes many common expressions concise.

Our previous work [16] provided evidence that Mavo empowered users with no programming experience to author fully functional CRUD applications. But Mavo offered only direct editing of individual data items, while many applications call for richer, programmatic modification of large collections of data simultaneously. For example, while a simple To-Do list could be easily implemented with a few lines of HTML+Mavo, clearing all completed items was not possible. Mavo did offer controls for deleting individual items, but no way to specify such actions that programmatically delete certain items.

Our Contribution
In this work, we extended Mavo with a new HTML attribute, mv-action, for specifying programmatic data updates. Its value describes the data mutation as an expression, and it is placed on the element that will trigger the action by clicking. The expression leverages Mavo’s existing expression syntax as much as possible, but adds functions that modify the data. We also implemented functions and operators to filter and define data in Mavo expressions, to increase the expressiveness of such expressions. A short example implementing bulk deletion can be seen in Figure 1.

Our hypotheses are that (a) the set of primitives we have chosen is expressive enough to meaningfully broaden the class of data management applications that can be created without undue complexity, and (b) novice web authors can easily learn to specify programmatic data mutations. To examine the first hypothesis, we list a number of case studies of common interactions and how they would be expressed with our update syntax. To examine the second hypothesis, we conducted a user study with 20 novice web developers writing a variety of data mutations, using first their own imagined syntax and then ours. We found that the majority of users were easily able to learn and apply these data mutation expressions with 90% of our participants getting two thirds of the questions right on first try, with no iteration. We also found that in many cases, our syntax was very close to their imagined syntax.

Although the presentation of information is an important part of Mavo, our work here focuses on and extends the computational power of MavoScript, an expression language similar to that of spreadsheet functions. While we consider Mavo, we believe our work suggests more broadly a way to increase the power of such functional reactive programming environments, including spreadsheets, for novice users.

RELATED WORK
Our work extends the Mavo language [16]. A full discussion of related work can be found there. In summary, many platforms and systems have been developed over the past few decades to help web authors build web applications. Some of these tools were targeted for web developers with limited programming and database knowledge [4, 17], allowing them to make programmatic changes to the data using SQL queries. Others were developed for novice web designers who are interested in rapid development of web applications [1, 13, 7], with spreadsheet as a back-end, but with limited or no mechanism to make data updates programmatically.

MAVO DATA UPDATE LANGUAGE
The mv-action HTML attribute
Data updates are specified via an mv-action HTML attribute, which can be placed on any element. Its value is an expression that describes the action that will be performed. The action is triggered by clicking (or focusing and pressing spacebar, for keyboard accessibility), as that appears to be the most common way to invoke an action on most types of GUlS.

Data mutation functions
We extended Mavo’s expression language with four data modification functions (brackets indicate optional arguments):

- `set(reference, value)`
- `delete(ref1 [, ref2 [, ref3 ..]])`
- `add(collection [, data] [, position])`
- `move(from, to)`

The first three are analogous to the SQL primitives UPDATE, INSERT, DELETE, whereas the latter is a composite mutation (delete, then add). These functions are only available in expressions specified with the mv-action attribute. Regular Mavo expressions remain side-effect free.

We kept these functions minimal, to delegate selection and filtering logic to Mavo expressions. This maximizes the amount of computation specified in a reactive fashion.

Defining literal data in expressions
Originally in Mavo, data was created either in the HTML at application authoring time, or by the user’s direct manipulations. However, with programmatic data mutations, it is sometimes necessary to specify new or changed data programmatically. To support expressions describing literal complex structures, we defined a `:` (colon) operator, and two functions: `group()` for objects and `list()` for arrays. By combining these with existing literals, any JSON structure can be specified.

The `:` (colon) operator is used for defining key:value pairs, which the `group()` function can then combine in a single object. The `list()` function produces arrays, like those that Mavo list-valued properties return.

Other additions
Update actions often consist of multiple elementary updates, executed sequentially. The Dice Roller in Figure 3 demonstrates an example (add(), then set()). To facilitate this, we extended MavoScript to support multiple function calls in
the same expression. These can be separated by commas, semicolons, whitespace, or even nothing at all.

To enable filtering of list-valued properties, we implemented a where operator. In imperative languages, such filtering is not as essential, because programmers are expected to collect the target data by looping. However, in declarative data update languages (like ours or SQL), filtering is more important, because data is operated on at all once. The operator has low precedence, so parentheses are rarely needed, e.g. person where age > 30 and name = ‘Lea’.

EXAMPLE USE CASES
Part of our argument is that Mavo data actions have sufficient power to easily specify a broad range of programmatic data manipulations in applications. To support that argument, we outline a few common interactions below. None of these can be implemented with the original Mavo alone. In each case, we show the source code, primarily HTML and original Mavo syntax; we highlight the code leveraging data actions.

Generalizing existing Mavo data updates
All Mavo data update controls except drag and drop can be expressed concisely as data actions, which facilitates UI customization. The following examples assume the updates are modifying a collection with property="item". Note that index (or $index) is a built-in Mavo variable that resolves to the index of the closest collection item, starting from 0.

<table>
<thead>
<tr>
<th>Action</th>
<th>Data update Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add new item button (outside collection)</td>
<td>add(item)</td>
</tr>
<tr>
<td>Add new item after current</td>
<td>add(item, item)</td>
</tr>
<tr>
<td>Duplicate current item</td>
<td>add(item)</td>
</tr>
<tr>
<td>Delete current item</td>
<td>delete(item)</td>
</tr>
<tr>
<td>Move up</td>
<td>move(item, index - 1)</td>
</tr>
<tr>
<td>Move down</td>
<td>move(item, index + 1)</td>
</tr>
</tbody>
</table>

Common Interactions and Widgets
One natural class of use cases for data actions is in creating rich new UI widgets. These widgets generally present underlying data from the traditional model in some novel fashion. But the widgets also tend to come with their own internal view model describing the state of their presentation. Data actions can be used to control the state of the view model, and thus to manage the widget’s data presentation.

Select All
It is common to offer an affordance to simultaneously check or uncheck all items in a list.

```html
<button mv-action="set(selected, true)">Select All</button>
<button mv-action="set(selected, false)">Unselect All</button>
<div input type="checkbox" property="selected" />
</div>
```

Accordion / Tabs
An accordion permits a user to show one of several distinct sections of content, while the rest are hidden. The same markup, with different CSS, could also be used to implement a tabbed view.

```html
<details property="prop" mv-multiple open="[open]"
  mv-action="set(open.$all, false) set(open, true)"
>
  <summary property="title"></summary>
  <meta property="open" />
  <!-- content -->
</details>
```

Pagination / Slideshow / Carousel
The following markup implements a working pagination widget and the content it paginates. It uses the Mavo mv-value attribute to generate a dynamic collection of page markers, then mv-action to make them clickable. An expression on each item controls whether it should be displayed based on the current page. The common slideshow and carousel paradigms can be implemented much the same way, essentially paginating one item per page.

```html
<meta property="cur" content="1" />
<meta property="per" content="10" />
<meta property="pages" content="[ceil(count(item) / per)]" />
<a mv-action="set(cur, cur - 1)" mv-if="cur > 1">«</a>
<a mv-if="cur < pages">»</a>
<div property="post" mv-multiple hidden="[page != cur]">
  <meta property="page" content="[ceil((index + 1) / per)]" />
</div>
```

Sorting table by clicking on column header
Mavo provides an mv-sort attribute whose value is the property name(s) to sort by. Data actions can dynamically change that via a helper property that holds the property name.

```html
<table>
  <th>
    <th mv-action="set(sortBy, 'name')">Name</th>
    <th mv-action="set(sortBy, 'age')">Age</th>
  </th>
  <tr property="person" mv-multiple mv-sort="[sortBy]">
    <td property="name"></td>
    <td property="age"></td>
  </tr>
</table>
```

An alternative solution would be to overwrite the collection with a sorted version of itself each time the header is clicked:

```html
<th mv-action="set(person, sort(person, name))">Name</th>
<th mv-action="set(person, sort(person, age))">Age</th>
```

Adding events to a map
This example positions each collection item over a 720 × 360 equirectangular map by storing the mouse position at the time of clicking. Similar logic can be used for adding events to a calendar view.
We recruited 20 participants (age $\mu = 36.2$, $\sigma = 9.25$; 60% female, 40% male) by publishing a call to participation on social media and local web design meetup groups. Their (self-reported) skill levels in HTML and CSS ranged from beginner to expert, but intermediate or below in JavaScript. 11/20 described themselves as beginners or worse in any programming language, while 9/20 were intermediate. Regarding data concepts, 5/20 stated they could write JSON, 4/20 could write SQL, and none could write HTML metadata (RDFa, Microdata, Microformats). We asked our participants to read through the Mavo Primer\(^1\) and optionally to create a shopping list application with Mavo before coming in for the study.

**Study Design**

Sessions were conducted one-on-one, in person and were limited to 90 minutes. Participants were shown a Mavo application with two collections (men and women) each containing a name, an age and a collection of hobbies (Figure 2). We decided on this schema because it is nested, and the properties have an obvious natural meaning.

First, participants were asked to write expressions that compute counts for five questions of increasing difficulty, starting from the simplest (“Count all men”) down to filtered counts (e.g. “count women older than 30”, “count women who have ‘Coding’ as a hobby”, etc), which participants found problematic in the first Mavo study [16]. Participants were discouraged from iterating on their expressions, and were told we wanted to capture their initial thinking.

The purpose of this part of the study was three-fold: (a) to assess their understanding of existing Mavo capabilities, (b) to verify whether filtered counts were indeed harder, and (c) to prime them into thinking in terms of declarative functional expressions for the study that was yet to follow.

The second part was a natural programming elicitation study [11]. We briefly explained the problem that Mavo data updates are solving, as well as our idea for addressing it on a high level. More specifically, we mentioned the `mv-action` attribute, as well as the `set()`, `delete()`, `add()`, and `move()` functions, but presented this as ideas whose syntax we are not sure about and had not developed yet. We then asked participants to answer 17 data update questions of increasing complexity (Table 2) by writing the syntax that felt more natural to them. They were also encouraged to even use different function names, if that felt more natural to them.

After this stage, we revealed our language prototype so that they could experiment with it during the study. After a brief tutorial (5-10 minutes), participants had to answer the same questions, in the same order, using our syntax. After this

\(^1\) mavo.io/docs/primer

<table>
<thead>
<tr>
<th>HTML</th>
<th>CSS</th>
<th>JavaScript</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Beginner</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Intermediate</td>
<td>5</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Advanced</td>
<td>12</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Expert</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2: All 17 data manipulation questions. The third column indicates whether filtering was needed to answer the question.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Delete all men</td>
<td>delete</td>
</tr>
<tr>
<td>2</td>
<td>Add new man (with no info filled in)</td>
<td>add</td>
</tr>
<tr>
<td>3</td>
<td>Delete all people</td>
<td>delete</td>
</tr>
<tr>
<td>4</td>
<td>Add a new man and a new woman</td>
<td>add</td>
</tr>
<tr>
<td>5</td>
<td>Delete current man</td>
<td>delete</td>
</tr>
<tr>
<td>6</td>
<td>Make current man 1 year older</td>
<td>set</td>
</tr>
<tr>
<td>7</td>
<td>Make everyone 1 year older</td>
<td>set</td>
</tr>
<tr>
<td>8</td>
<td>Set everyone’s name to their age</td>
<td>set</td>
</tr>
<tr>
<td>9</td>
<td>Delete women older than 30 years old</td>
<td>delete ✓</td>
</tr>
<tr>
<td>10</td>
<td>Move the current woman to the collection of men</td>
<td>move ✓</td>
</tr>
<tr>
<td>11</td>
<td>Add a woman with the name “Mary” and age of 30</td>
<td>add</td>
</tr>
<tr>
<td>12</td>
<td>Add a woman with the name “Mary” and age of 30 to the beginning of the women collection</td>
<td>add</td>
</tr>
<tr>
<td>13</td>
<td>Delete “Dining” as a hobby from everyone</td>
<td>delete ✓</td>
</tr>
<tr>
<td>14</td>
<td>Rename every man with age &gt; 40 to “Chris”</td>
<td>set ✓</td>
</tr>
<tr>
<td>15</td>
<td>Move the current woman to the beginning</td>
<td>move ✓</td>
</tr>
<tr>
<td>16</td>
<td>Change the age of the woman named “Mary” to 50</td>
<td>set ✓</td>
</tr>
<tr>
<td>17</td>
<td>Move all men to the collection of women</td>
<td>move</td>
</tr>
</tbody>
</table>

In this part of the study, we wanted to explore what syntax participants found natural, with the only suggestion being that they had to use the four functions (set, add, move, delete). This suggestion was introduced to put participants in the mindset of writing expressions instead of natural language. They were even encouraged to use different function names if they wished to, and 6 did so at least once (half of them inconsistently).

Despite emphasizing that constraint, 6/20 participants did not use any functions for answering at least one of the 17 questions but wrote statements instead (such as `age = age+1`) and 4 more used a hybrid approach, with some parameters outside the function call, such as `add(woman) name=Mary age=30`.

RESULTS & DISCUSSION

Counting questions

All participants correctly answered all simple counting questions, even when they had to count a scoped property (counting all hobbies from outside both collections of men and women). Also, they seemed to have no trouble with filtered aggregates like `count(age > 3)` with 17/20 getting them right, and the remaining three making only minor syntax errors.

Participants had trouble disambiguating between scoped properties with the same name across two collections (e.g. getting only women’s ages or men’s hobbies). Mavo (like SQL) uses dot notation for this (`woman.age` only returns women’s ages), which only 8/20 participants used. However, as there was no example of this in the Mavo Primer, we did not consider these failures a sign of poor understanding of Mavo functionality.

Free-form Syntax

In this part of the study, participants found natural, with the only suggestion being that they had to use the four functions (set, add, move, delete). This suggestion was introduced to put participants in the mindset of writing expressions instead of natural language. They were even encouraged to use different function names if they wished to, and 6 did so at least once (half of them inconsistently).

Despite emphasizing that constraint, 6/20 participants did not use any functions for answering at least one of the 17 questions but wrote statements instead (such as `age = age+1`) and 4 more used a hybrid approach, with some parameters outside the function call, such as `add(woman) name=Mary age=30`.

Scope

We previously mentioned that in Mavo expressions, property names can be used anywhere and resolve to the local value or all values depending on the expression placement, enabling very concise references for common cases. Thus, `delete(man)` used “inside” a particular item in a collection of man objects would delete only that item, while `delete(man) “outside”` the collection would delete all those man objects.

However, in their own syntax, many subjects wanted to make this distinction explicit. 8/20 used a keyword or function to refer to all items (e.g. `man.all`) at least once, and 8/20 used an explicit keyword or function for the current item, such as `woman.current` or `this`. Only 3 participants did both. Interestingly, none followed their own referencing schemes consistently, using these explicit references only in some of the questions or some of the arguments, and plain property names elsewhere. This may indicate one reason why this referencing scheme is useful: it eliminates error conditions.

More work is needed to understand why our subjects attempted this more verbose language when the more concise one would work. Based on participant answers to probing questions, the survey format may have played a role: they were writing their answers in text fields, separately from the HTML, so the context of their expression was removed. In that setting, it may have been jarring to write the same expression as an answer to completely different questions (e.g. “Delete all men” and “Delete current man” are both `delete(man)` with our syntax). Perhaps if they’d been writing Mavo expressions inside actual HTML, the disambiguation through context would have eliminated the desire to disambiguate through syntax.

Another possibility is that novice programmers prefer verbosity. Pane et al. [12] showed that 32% of non-programmers constructed collections by using the keywords every or all. The use of such verbose syntax could be seen as a form of commenting, adding clarity over more concise code. It is easy to provide syntactic sugar to allow such explicit references. In fact, Mavo already defines special all and this variables that work in a similar way although we did not mention this.
We also observed the reverse, of users trying to be more concise. 7/20 participants indicated that the target of their action is the current item by omitting a parameter, such as writing `delete()` for deleting the current item or `move(man)` for moving the current woman to the collection of men.

### Underspecified expressions
Stylistic choices such as punctuation should be distinguished from expressions which must be regarded as incorrect because they are missing necessary information for the operation. However, even in those cases, it is hard to be certain that there is a logic error at play. Are the missing parameters actually missing, or did the participant have a clever heuristic in mind for inferring them? And if not, is it a logic error, or merely a slip? In this section, we describe some of the most common patterns of (ostensibly) underspecified expressions that we observed.

By far the most common one was `delete(<PREDICATE>)` with no reference to the item(s) to be deleted. For example, `delete(woman.age>30)` for deleting women older than 30, `delete(hobby='Dining')` to remove the hobby “Dining” from all people. 18/20 participants did this at least once, and 10 did so in both of the conditional delete questions (Q9, Q13). One possible interpretation would be that the set targeted for removal is specified by the first token in the expression—e.g. "woman" and "hobby" in the above examples. But this seems likely to be fragile in general use—for example, does the second expression above try to delete all "Dining" hobbies, or does it try to delete all women who have dining as a hobby?

Another common pattern was using `set(age+1)` to increment all ages. 15/20 participants used a variation of this syntax. This is consistent with the proposed interpretation above, that the update target is the first named token. Interestingly, there was no such underspecification in Q8, which involved two properties. None of the participants realized the inconsistency when they answered the latter and did not think to back and change their answer to the former. Asking a subset of participants about this at the end revealed that some of them thought of `age+1` as an increment operator (like C’s `age++`) whereas for others, omitting the property to be modified was a slip.

Both patterns may indicate a distaste for parameter repetition, which is on par with natural language: “parameters” are only explicitly specified when different and are otherwise implied.

5/20 participants wrote their expressions as if the “name” property had a special meaning, i.e. was an implied primary key. For example they would use `Mary` as a way to refer to the person that has name="Mary"”, without specifying “name” anywhere in their expression. This did not seem to correlate with a lack of (self-reported) programming skill, as only one of them had not been exposed to programming at all. It is unclear whether this has to do with the word “name” itself, or with the fact that names were indeed unique in the data we gave them.

Using objects as numbers was common, e.g. omitting “age” from `delete(woman>30)` or `set(man+1)`. Many participants attempted it at first, and 4/20 submitted their answers with it. In many cases this turned out to be a slip, but two participants articulated a consistent mental model: it automatically operates on all numeric properties! In Pane et al. [12], 61% of non-programmers modified the object itself instead of its properties, which is higher than the percentage we observed.

### Argument separation
While commas are likely the most widely used argument separator, they did not appear to be very natural to our subjects. 7/20 did not use any commas, but instead separated arguments by other symbols, or even whitespace. 5/20 only used commas for repetition of the same argument type (e.g. `delete(man, woman)`). From the remaining 8 subjects, only 2 used commas
16/20 subjects used “=” to separate arguments at least once. 9/20 used “to”, primarily in set() and move(). Other separators (whitespace, colons, and parentheses) were used by 3 people or fewer.

**Sequencing function calls**

Only 8/20 participants used multiple function calls in an expression (such as add(man) add(woman)). The rest tried to express compound actions via arguments of one function call (such as add(man, woman), even when this was inconsistent with their later responses.

In spreadsheets, expressions have no side effects and only produce one output, therefore there is never a need for multiple adjacent function calls. Therefore, using more than one function call may feel foreign and unnatural to these users.

**Filtering**

Four of our questions required filtering on a collection (cases where the corresponding SQL query would need a WHERE or HAVING condition). Half of our participants defined a language-level filtering syntax, such as if or where keywords, or parentheses (e.g. woman(age>30)). 6/20 expected that the data update functions would allow a filtering argument.

5/20 expected that predicates would act as a filter of the closest collection item and consistently used them in that fashion. For example, they expected that man.age > 40 would return a list of men whose age was larger than 40, and wrote expressions like set((man.age > 40).name to "Chris") for Q16. However, in Mavo currently the inner expression returns a list of booleans corresponding to the comparison for each man.

**Relationship to prototype syntax**

Participant free-form syntax was consistent with our current prototype syntax (would have produced the correct result) with no changes in 4.35/17 answers on average ($\sigma = 2.16$) and with minor changes (different symbols or removing redundant tokens) in 8.6/17 answers on average ($\sigma = 2.06$).

**Prototype Syntax**

In this part, we revealed our syntax prototype to participants and asked them to answer the same questions, but this time using our syntax, to test the learnability and usability of our prototype. Participants were not allowed to test their expressions, and were discouraged from iterating because we wanted to capture their initial thinking. Therefore, correct answers in this section are equivalent to participants getting the answer right on first try and with no preceding training tasks.

Overall, 11 out of 17 questions had a correctness rate of 75% or above with 8 (Q1-3, Q5, Q8-10, Q17) having 90% or above, i.e. almost every participant got them right on first try.

The most prominent patterns from the previous step persisted, though to a lesser extent. 7/20 participants remained unable to use a sequence of two function calls for Q4 despite this being covered in the tutorial, and wrote add(man, woman) or a variation. Curiously, based on later answers, all seemed to understand that the second parameter of add() holds initial data, but none realized the inconsistency. Similarly, 4/20 participants still used set(age+1) to increment ages, 2/20 used objects as numbers, and 8/20 used delete(<predicate>).

Almost all failures in “add with initial data” questions (Q1-12) were related to grouping the key-value pairs, or incorrectly using equals signs (=) instead of colons (:) to separate them.

Two questions asked participants to delete items with a filter, but had vastly different success rates. 18/20 participants got Q9 correct, while only 9/20 got Q13 right, despite the superficial similarity of the two questions. Participants had a very hard time using hobby twice in Q13 (The correct answer is delete(hobby where hobby = ‘Dining’) and even those that got it right hesitated a lot before writing it.

By far the hardest questions were Q14 and Q16, where participants had to filter on one property and set another. Only 7/18 of participants answered them correctly. All knew which function to use, and almost all used where correctly for filtering, but were then stuck at where to place the property they were setting. In Q14, a common answer was set(man where age > 40, "Chris"). Users when then unsure where to put name. The correct syntax in this case (if using where) would have been set(man.name where age > 40, ‘Chris’). which is indeed confusing as one would expect the property being set to be grouped with its value, not with the filtering predicate.

**Hands-on Tasks**

16 participants completed the hands-on section of the study (see Figure 3). Half were randomly assigned to the DICE ROLLER application, and the rest to the WORDS GAME application. 13 also completed the Shopping List tasks.

**Dice Roller Application**

All 8 participants solved the first two tasks correctly and were able to display a random dice roll (task 1) within a median time of 55 seconds and to display it in the history (task 2) within a median time of 70 seconds. 5/8 and 3/8 did so on first try. 5/8 participants hesitated before using multiple function calls in mv-action, even if they had answered Q4 with two function calls in the survey, but they eventually got it right.

The third task was to prevent the current dice roll from showing up in the history. Despite the second task being carefully worded to avoid implying a particular order, all 8 participants used add() after the set() they had written in the first task. This places the current die in the history as well as the main display. The opposite order would have rendered the third task redundant, yet nobody realized this. Furthermore, only 1 participant was able to solve the third task. All they had to do was use add() before set(), i.e. swap the order of the two functions. This would add the dice to the history before they replace its value with a new random value. None of the other 7 participants was able to figure out why this was happening, nor how to fix it. Some participants thought that multiple function calls are executed in parallel, a common misconception of novice programmers [10]. This appears to be a general failure of computational thinking, not specific to Mavo.
Words Game
This proved to be substantially easier than the dice roller. 6/8 participants succeeded in all three tasks. Clicking on words to add them to the sentence took a median time of 220 seconds, deleting the last word (Undo) took 43 seconds, and deleting all words took 115 seconds. For the first task, a common mistake (3/8 participants) was to use move() instead of add() to copy the clicked word into the sentence. Even after realizing their mistake, they were ambivalent about using add().

Shopping List
13 subjects carried out the SHOPPING LIST tasks, copying to (task 1) and from (task 2) a “Common Items” collection. 6 participants brought their own application and 7 used ours. Almost all participants succeeded in both tasks, with only 1/13 failing the first task and 3/13 failing the second one. It took slightly longer for participants using their own app to get started on the first task with a median of 133 seconds vs 55 seconds. By the second task the difference had been eliminated (55 vs 50 seconds). Three participants were confused about whether to use move() or add() to copy the shopping list item to the common items, but quickly figured it out after trying.

System Usability Scale (SUS)
At the end of each session, subjects rated their subjective experience on a 7-point SUS scale with 10 alternating positive and negative questions. The answers were then coded on a 5-point scale and the SUS score was calculated according to the algorithm in [3]. We removed one participant who had selected “Agree” on all 10 questions (positive and negative), indicating lack of attention, a common problem with SUS.

Our raw SUS score was 76.3 (σ = 2.43), which is higher than 77.5% of all 446 studies detailed by Sauro [15]. Our raw Learnability and Usability scores as defined by Lewis and Sauro [9] were 78 and 69.7 respectively.

General Observations
The overall reactions to the data mutation functions ranged from positive to enthusiastic. Several participants remarked on the perceived intuitiveness of the syntax. One participant answered several questions on the survey in one go, without looking at the documentation, then paused and said “It’s so intuitive, I don’t even need to look at the docs!” Many other participants remarked on expressiveness. “It is very easy to do complex things!”, as one of our participants phrased it.

Most participants described our data update actions as easy, even those who made several mistakes. One user said “This is very application-centered, a page that can actually do something!”. Another user commented on the data mutation functions saying “I think they are very useful, easy, and approachable”, and another user said “It is definitely more accessible than having to program, so that’s pretty cool”. Another one said “They are easier and quicker to make things without worrying about technicality. It is very easy to use”.

As with the first Mavo study, many participants liked being able to add these functionality by editing HTML as opposed to editing in a separate file and/or language. One user said “Interesting to be able to do these things from the HTML!” and another said “It is interesting!...being able to do this in HTML, I was able to use it pretty easily, once I knew what functions there were and the syntax it has was very easy.”

Users also liked the fact that they can build applications that typically require programming. One said “This is easier than JavaScript! If I wanted to do something complicated I would be frustrated to use JavaScript cause I’m not good at it, this is easier”. Another said “It’s easier and quicker to make things without worrying about technicality. It’s very easy to use”.

Several participants commented positively on the where operator. “the where syntax is like natural language, I did not expect it to be there and written as if I am saying it”.

FUTURE WORK
Improving the learnability of our syntax
We will apply the user study findings to iterate on our syntax and make it more natural. Many participants wanted to use a to keyword, which can be easily added. Several participants were confused about the group() function, what it does, and when it is needed, so we will examine whether it is possible to design the language in such a way that group() is not required, possibly by using a variable number of arguments in add() or by requiring plain parentheses instead of group().

We may decide to special case certain patterns to match user expectations: predicates will be allowed as the sole argument in delete() and will target the closest item.set(a = b) could be rewritten as set(a, b). Repetition in where can be avoided by expanding property where value to property where property = value. Underspecified assignments, such as set(age + 1) could target the first named token.

We also need to improve the syntax for tasks which filter one property and set another (Q14 and Q16), since our user study indicated clear problems with the current syntax.

More Triggers
Currently, our data actions are primarily triggered by clicking. In the future we are planning to add the ability to specify different triggers, such as double clicking, keyboard shortcuts, dragging, or mousing over the element. This could be done via an HTML attribute whose value would be an “event selector”, as defined by Satyanarayan et al. [14]. Event selectors facilitate composing and sequencing events together, allowing users to specify complex interactions very concisely.

CONCLUSION
This paper extends the Mavo language, adding programmatic data updates that are triggered by user interaction. Our user study showed that HTML authors can quickly learn to use this syntax to specify a variety of data mutations, significantly expanding the set of possible applications they can build, with only a little increase in language complexity.

ACKNOWLEDGEMENTS
We thank Eirik Bakke for his help with SUS. We would also like to thank our study participants and pilots for their time and hard work, and our reviewers for their deep, thoughtful feedback and their helpful guidance towards shortening the paper.
REFERENCES


