# Cicero: A Declarative Grammar for Responsive Visualization

Hyeok Kim
Northwestern University
Evanston, IL, U.S.A.
hyeok@northwestern.edu

Ryan A. Rossi
Adobe Research
San Jose, CA, U.S.A.
rrossi@adobe.com

Fan Du
Adobe Research
San Jose, CA, U.S.A.
fdu@adobe.com

Eunyee Koh
Adobe Research
San Jose, CA, U.S.A.
eunyee@adobe.com

Shunan Guo
Adobe Research
San Jose, CA, U.S.A.
sguo@adobe.com

Jessica Hullman
Northwestern University
Evanston, IL, U.S.A.
jhullman@northwestern.edu

Jane Hoffswell
Adobe Research
Seattle, WA, U.S.A.
jhoffs@adobe.com

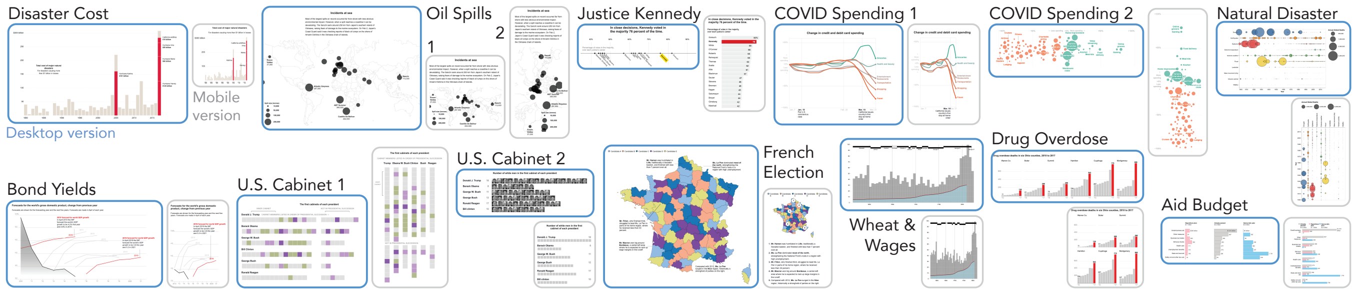


**Figure 1: Thirteen responsive visualization use cases reproduced using Cicero. The blue- and gray-bordered views are the desktop and mobile versions, respectively. The mobile versions of the Oil Spills case are from (1) the original article and (2) the version suggested by Hoffswell et al. [13]. Full size images are included in the Supplemental Material (https://osf.io/eg4xq).**

## ABSTRACT

Designing responsive visualizations can be cast as applying transformations to a source view to render it suitable for a different screen size. However, designing responsive visualizations is often tedious as authors must manually apply and reason about candidate transformations. We present Cicero, a declarative grammar for concisely specifying responsive visualization transformations which paves the way for more intelligent responsive visualization authoring tools. Cicero's flexible *specifier* syntax allows authors to select visualization elements to transform, independent of the source view's structure. Cicero encodes a concise set of *actions* to encode a diverse set of transformations in both desktop-first and mobile-first design processes. Authors can ultimately reuse design-agnostic transformations across different visualizations. To demonstrate the utility of Cicero, we develop a compiler to an extended version of Vega-Lite, and provide principles for our compiler. We further discuss the incorporation of Cicero into responsive visualization authoring tools, such as a design recommender.

## CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; **Visualization**.

## KEYWORDS

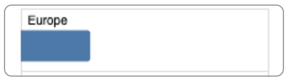
Visualization, responsive visualization, grammar

**ACM Reference Format:**
Hyeok Kim, Ryan A. Rossi, Fan Du, Eunyee Koh, Shunan Guo, Jessica Hullman, and Jane Hoffswell. 2022. Cicero: A Declarative Grammar for Responsive Visualization. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3491102.3517455

## 1 INTRODUCTION

Responsive visualizations adapt visualization content for different screen types, making them essential for most Web-based contexts due to an increasing proportion of mobile viewers. Responsive visualization authoring environments, however, tend to require considerable manual effort on the part of visualization designers. Prior findings on responsive visualization design practices [13, 16] indicate that authors often start from a source view and then apply responsive transformations to produce a set of target views optimized for different screen types. However, this approach can be tedious as authors must manually explore, apply, and evaluate different responsive strategies one by one. For example, authors might create responsive views by crafting an artboard and/or specification per responsive view, which is particularly problematic when one of the responsive views is revised. They may have difficulty in expressing changes that occur across a design specification

**Label-mark serialization**



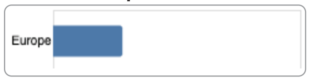
(a1) Using Vega-Lite

```
5  mark: {
6    type: “bar”,
7    yOffset: 5,
     ... },
10 encoding: {
11   y: {
       ...
14     axis: {
         ...
19       labelAlign: “left”,
20       labelBaseline: “middle”,
21       labelPadding: -5,
22       labelOffset: -15, ...
```

(a2) Using Cicero

```
7  { specifier: {
8      role: “axis.label” }},
9    action: “transpose”,
10   option: {
11     serial: true }}, ...
```

**Label-mark parallelization**



(b1) Using Vega-Lite

```
5  mark: { type: “bar”,
     ... },
6  encoding: {
7    y: {
       ...
14     axis: { ... }, ...
```

(b2) Using Cicero

```
7  { specifier: {
8      role: “axis.label” }},
9    action: “transpose”,
10   option: {
11     serial: false }}, ...
```

**Figure 2: Design specifications for label-mark serialization using (a1) Vega-Lite and (a2) Cicero and parallelization using (b1) Vega-Lite and (b2) Cicero.**

(e.g., example cases in Figure 2 and Figure 3). Authoring painpoints like these suggest a need for more intelligent authoring tools, such as semi- or fully automated recommenders that support exploring and reasoning about responsive design strategies [16, 17].

A key step toward such intelligent responsive visualization authoring tools is a concise, declarative grammar that can express a diverse set of transformation strategies. While declarative visualization grammars like Vega [34] and Vega-Lite [33] are well suited to developing more sophisticated visualization authoring tools, they are not necessarily well suited to representing visualization transformations; Hoffswell et al. [13] observe that different edit properties for text and marks in Vega-Lite [33] make it complicated to create the specifications for multiple versions of a visualization despite its high expressiveness. Indeed, many responsive visualization strategies that researchers have identified [16] can be written in Vega-Lite with high complexity. For instance, serializing labels and marks using Vega-Lite (i.e., placing them in a vertical order [16]) requires layout adjustment keywords (Figure 2a1, line 7, 19–22), while parallelizing them (i.e., arraying them horizontally) does not require layout modifications in Vega-Lite (Figure 2b1, line 14). Whereas Vega-Lite requires authors to create separate specifications for each responsive view that interleave complex layout changes throughout the specifications, a declarative grammar for responsive transformations can express the same strategies in a simpler way as shown in Figure 2 (a2) and (b2). Such an approach can help visualization authors easily and quickly compose responsive design specifications and can help developers to more effectively develop authoring tools for responsive visualization.

To this end, we present Cicero: a flexible, expressive, and reusable declarative grammar for specifying responsive visualization transformations. The flexible *specifier* syntax of Cicero enables querying visualization elements using their role (e.g., mark, axis labels, title), underlying data, and attributes of visualization elements, independent of the structure of a source view specification. Cicero provides a compact set of *action* predicates (`add`, `duplicate`, `remove`, `replace`, `swap`, `modify`, `reposition`, and `transpose`) that can encode a diverse range of transformation techniques (Figure 5c). Moreover, Cicero supports extracting and reusing generalizable transformations strategies across multiple responsive specifications. For example, the expressions (a2) and (b2) in Figure 2 can be reused on other visualizations with bar-like marks and axis labels.

To demonstrate the utility of Cicero, we develop a Cicero compiler for an extended version of Vega-Lite that we adapted to support annotations and other narrative devices and reproduce 13 real-world examples in Cicero (Section 6). We provide a set of principles for developing our Cicero compiler in terms of desirable properties of the association of visualization elements, preferable default behavior, and how to manage conflicts between transformations (Section 5). As Cicero is agnostic to the underlying structure of a source visualization, it can be leveraged in different visualization authoring tools. To demonstrate the feasibility of Cicero in such authoring tools, we describe how Cicero applies to a prototype recommender we developed for responsive transformations as a proof of concept and envision an approach to mixed-initiative authoring tools (Section 7). Future work can implement a Cicero compiler for other declarative grammars like the original Vega-Lite [33] or ggplot2 [37] and other recommender approaches (e.g., [40]).

## 2 RELATED WORK

This work is motivated by prior research on responsive visualization and declarative visualization grammars.

### 2.1 Responsive Visualization

Prior research has examined how visualization authors customize a visualization for smaller screens in terms of visual elements and structure [2, 8, 43], and interaction methods [15]. For instance, VISizer [43] provides a point-of-interest-based framework to resize a visualization while preserving regions with important insights. Recent works [13, 16] provide a more comprehensive snapshot of current responsive visualization design practices. Motivated by a qualitative analysis of 231 responsive visualizations and a formative interview study, Hoffswell et al. [13] implement an authoring tool that supports editing across different responsive views via simultaneous previews and global edits, as well as view-specific customization. Using a similar approach, Kim et al. [16] present a set of responsive visualization design patterns and identify a trade-off between achieving appropriate graphical density for each view and preserving intended takeaways across transformations. To address the trade-off between density and takeaways, Kim et al. [17] provide a set of task-oriented insight preservation measures for a responsive visualization recommender limited to a small set of design transformations (e.g., aggregation, axes-transposing). A recent machine learning-based approach [40, 42] provides automated methods to configure visualization layouts based on the chart size using a set of simple heuristics, yet it does not offer a grammar that can express a large set of responsive visualization techniques.

Responsive design has been well-studied for the Web more generally [7, 27], but such techniques are not directly applicable to responsive visualization design because they are intended for Web layouts and based on limited knowledge of visualization design. For example, CSS media queries [24] express breakpoints for each responsive version of the contents. CSS specifications under a media query of `@media screen and max-width 600px` are shown only on a screen-side application (e.g., Web browser) with width ≤ 600px. Similarly, CSS specifications under a media query of `@media speech` are used by speech synthesizers like a screen reader. However, using CSS alone cannot enable specification of many responsive transformations specific to visualization, such as transposing axis (requiring changes to scale functions), un-fixing tooltip positions, changing mark types (requiring dynamic positioning), and transforming data (requiring custom JavaScript functions).

In practice, designers create responsive visualizations with multiple tools in an iterative manner. D3.js [5] is a highly expressive JavaScript (JS) library for SVG- or Canvas-based visualizations. According to prior work on visualization authoring practices [4, 31], designers often use D3.js (or equivalent tools) with ai2html [35], which renders Adobe Illustrator vector images (`.ai` files) to HTML. Designers first draw a visualization using D3.js [5], then load and edit the SVG graphic of the visualization as responsive 'artboards' in Adobe Illustrator [31]. Authors can also define responsive condition parameters for interactive visualizations using D3.js (e.g., scale functions for $x$ and $y$ positions to be swapped for mobile screen). R3S.js [19] offers programming interfaces for such parameterization by extending D3.js [5]. However, it is not fully declarative, so authors need to imperatively define each transformation, which requires programming expertise. For example, to reposition a tooltip, which is a common responsive transformation strategy [16], R3S.js requires the use of custom CSS rules and/or JS functions.

For simple charts and quick edits, authors can utilize responsive properties of existing tools like Vega, Google Chart, and Microsoft Power BI. While Vega [34] and Vega-Lite [33] support some 'sensible' defaults, such as fitting the number of axis labels to the chart size, users need to have fully defined specifications for each of the responsive views. Google Chart [11] offers several default settings for mobile views such as truncating labels with an ellipsis (...). Power BI [26] provides defaults for responsiveness (e.g., making a visualization scrollable, rearranging legends, removing axis, etc.) [9]. While these tools can simplify the design process, their limited expressiveness may prevent authors from specifying intended responsive transformations, limiting their ability to convey insights.

Lastly, commercial tools like ZingChart and DataWrapper allow for responsive settings. ZingChart [44] provides 'media rules' through which a designer can declare a screen size condition for a visualization element (e.g., label: 'October 4' for screen size > 500 and 'Oct. 4' for screen size < 500). However, those media rules are dependent on the chart type—for example, transposing a scatterplot and a bar chart requires changes to data structure and the chart type, respectively—which limits the expressiveness and flexibility for responsive transformations. DataWrapper [1], an authoring tool for communicative visualizations, allows authors to choose whether and how to show a visualization element for mobile screens (e.g., showing a table as a stack of cards [30], or numbering annotations [29]). However, it is not available in the form of a declarative grammar which limits how easily it can be extended or applied to future authoring tools, such as a mixed-initiative authoring tool.

## 2.2 Declarative Visualization Grammars

Declarative grammars help visualization authors to avoid complex programming through a compiler that implements user-declared specifications (e.g., [12, 18, 28, 33, 34, 37]). For example, a Vega-Lite [33] specification uses JavaScript object notation (JSON) to encode chart size, data source and transformation, visual encodings, multiple views, and user interactions using predefined primitives. Some declarative grammars target specific use-cases by leveraging more general-purpose grammars. For example, Gemini's animated transition grammar formalizes chart animation entities [18] based on starting and ending visualizations specified using Vega [34]. Moreover, declarative grammars facilitate computational operations on visualization specification, which enables the development of useful visualization applications on top of the underlying grammar. For example, many end-user tools like visualization recommender(s) [28, 38, 39] and editor(s) [32] use Vega-Lite [33] to represent the visualization design specification. In responsive visualization settings, Hoffswell et al. [13] provide a design editor using Vega-Lite [33], and Kim et al. [17] propose automated recommendation of responsive visualization designs using Draco [28].

However, existing declarative visualization grammars are often limited when it comes to supporting expressive responsive visualization designs. For example, common responsive visualization strategies like fixing a tooltip position, aggregation, internalizing labels, and externalizing annotations (c.f. [16]) are not supported or are complicated to specify in Vega-Lite [33]. In addition, many commonly used visualization grammars (e.g., ggplot2 [37], Vega-Lite [33]) require authors to define multiple full visualization specifications for each responsive view, which makes it difficult to propagate changes from one design to another. ZingChart [44] provides 'media rules' to specify conditions for responsive properties, yet it is often difficult (or impossible) to express a large set of design transformations like transposing layout or changing mark types.

Our approach proposes a novel declarative grammar that can express various responsive transformations, accompanied by a compiler built on an extended version of Vega-Lite. To demonstrate the utility of Cicero for visualization tooling, we develop a proof-of-concept prototype recommender for responsive design transformations that encodes a larger set of design strategies than the scope of Kim et al. [17], using Cicero as the representation method.

## 3 THREE DESIGN GUIDELINES FOR A RESPONSIVE VISUALIZATION GRAMMAR

We derive three central design guidelines for a responsive visualization grammar based on prior work [2, 4, 13, 16, 19, 31, 44].

**(D1) Be expressive.** A responsive visualization grammar should be able to express a diverse set of responsive design strategies spanning different visualization elements. One approach is to characterize a responsive transformation strategy as a tuple of the visualization element(s) to change and a transformation action [13, 16]. Selecting visualization element(s) should support varying levels of customization for responsive transformations because transformations can include both systematic changes (e.g., externalizing all text annotations or shortening axis labels) and individual changes (e.g., externalizing a subset of annotations or highlighting

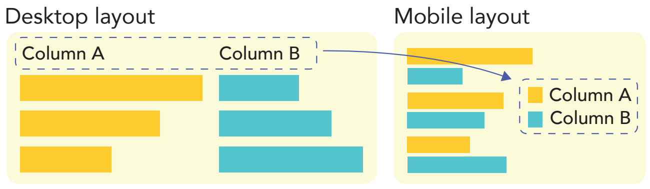


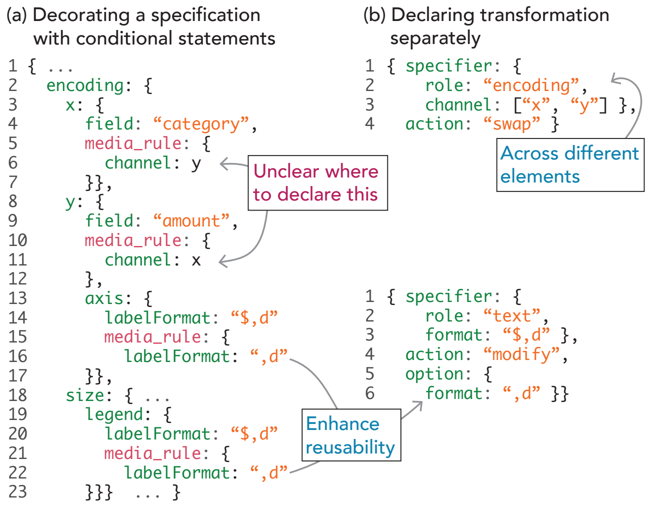
**Figure 3: Responsive transformation from axis labels to a legend accompanied by a layout change for smaller display.**

a particular mark) [16]. A grammar needs to express responsive transformations as a concise set of 'actions' describing how visualization elements are changed [13, 16]. **To be expressive, our grammar provides (1) a query syntax for selecting visualization elements both systematically and individually and (2) consistent, high-level action predicates that can encode a diverse set of responsive design strategies.**

**(D2) Be flexible.** A responsive visualization grammar should offer flexibility in how an author can specify the behavior of an entity under a responsive transformation, independent of how the entity is expressed in the specification (or structure) of the source visualization. For example, suppose a visualization that has a nominal color encoding that maps dog, cat, and fox to red, blue, and green. Then, to select red marks, some authors can specify simply "red marks" (using attribute) while others can make the same selection by specifying "marks for dog" (using data). Furthermore, responsive transformations can occur across different visualization elements. For instance, as illustrated in Figure 3, one can change the layout by moving a column element to the row (partial view transpose) to accommodate a portrait aspect ratio. Following the previous transformation, the column labels can be replaced with a legend if there is a redundant mark property encoding. **To be flexible, our responsive visualization grammar supports multiple expressions for specifying visualization elements that can be independent of the structure of a visualization.**

**(D3) Be reusable.** A responsive visualization grammar should enable authors to easily (i.e., without making big changes) reapply generalizable responsive transformations across different visualizations. While reuse is straightforward for visualizations sharing the same properties, many responsive designs utilize generic transformations that are independent of the specific chart design, data, or base visualization (e.g., transposing the layout, numbering annotations, using a fixed tooltip position). Moreover, authors might want to repeat techniques only for certain features of a visualization (e.g., removing a data field regardless of chart type). **To be reusable, our responsive visualization grammar represents each responsive transformation in a form that helps users to easily extract and apply transformations to other views.**

With these guidelines in mind, there are several possible approaches for specifying responsive transformations, such as: (1) decorating a complete visualization specification and (2) separately defining responsive transformations. The first approach uses conditional keywords (e.g., `media_rule` in ZingChart [44]) to express transformations. For example, in Figure 4a, the `media_rule` keywords for the *x* (line 5–7) and *y* (line 10–12) encodings describe the


(a) Decorating a specification with conditional statements
(b) Declaring transformation separately
Unclear where to declare this
Across different elements
Enhance reusability

```
1 { ...
2   encoding: {
3     x: {
4       field: "category",
5       media_rule: {
6         channel: y
7       }},
8     y: {
9       field: "amount",
10      media_rule: {
11        channel: x
12      },
13      axis: {
14        labelFormat: "$,d"
15        media_rule: {
16          labelFormat: ",d"
17      }},
18    size: { ...
19      legend: {
20        labelFormat: "$,d"
21        media_rule: {
22          labelFormat: ",d"
23      }}}  ... }
```

```
1 { specifier: {
2     role: "encoding",
3     channel: ["x", "y"] },
4   action: "swap" }
```

```
1 { specifier: {
2     role: "text",
3     format: "$,d" },
4   action: "modify",
5   option: {
6     format: ",d" }}
```

**Figure 4: Two possible approaches to specifying responsive transformations. (a) Decorating a specification with conditional statements. (b) Separately defining responsive transformations.**

changes for each encoding when viewed in a media format (e.g., a 'swap' action). The `media_rule` keywords for the *y* `axis` (line 15–17) and the *size* `legend` (line 21–22) describe the same change to the label format for both types of elements. For the same set of transformations, the second approach in Figure 4b directly declares that the two position channels should be swapped and concisely describes changes to the label format for all text elements. While we choose to use the JSON format, other formats could be used to extend our approach; for example, Altair [36] is a Python wrapper for Vega-Lite [33] that leverages object-method chains rather than Vega-Lite's JSON format.

While the first approach simplifies the learning process by extending an existing grammar, it can sometimes be tedious and unclear how to specify responsive transformations that apply to multiple elements. In particular, this approach often requires a single responsive change (e.g., transposing an axis) to be interleaved across multiple parts of the specification (Figure 4a, Line 5–7 and 10–12). In contrast, the second approach can enhance the reusability (**D3**: reusable) of a transformation specification by separating the desired responsive changes from the original visualization design. Furthermore, this approach can support more generalizable transformations that are independent of the original visualization structure (**D2**: flexible; e.g., changing all text formats directly). Therefore, in this work we opt for the second approach.

## 4 RESPONSIVE VISUALIZATION GRAMMAR

We introduce *Cicero*, a declarative grammar designed to concisely express responsive transformations. Paired with a declarative specification for a source visualization, Cicero provides a concise syntax for describing responsive changes independent of the structure of the original visualization specification. A single Cicero specification defines how to *transform* an initial visualization design to a new design, thereby encoding the responsive transformations required to convert a visualization into a responsive version for a particular format. A Cicero specification consists of a metadata object (`metadata`, line 2–4 of Figure 5a) and a list of transformation rules (`transformations`, line 5–78 of Figure 5a). The `metadata` object contains meta-information about the context for the target view (i.e., the intended environment, including information like the media type and screen size). The responsive strategies are encoded as separate rules in the list of `transformations`. We use a 'list' structure to enhance the reusability of the grammar by ensuring that each `rule` modularly describes a single responsive change to the source view (**D3**: reusable). The formal specification of the Cicero grammar is shown in Figure 6 and the Supplemental Material.

The core components of a `rule` object include the `specifier` (which elements to change), an `action` (how to change the elements), and the `option` (what properties to change). The `specifier` queries the source visualization to identify the set of existing visualization elements to be transformed, and supports flexibly referencing visualization elements with varying levels of scope (**D2**: flexible). Then, the `action` and `option` provide high-level direction and detailed information about the change to be made to the selected elements, respectively, together encoding a wide range of transformations to elements selected by the `specifier` (**D1**: expressive). For example, the rule object in line 6–9 of Figure 5a states that the compiler should 'modify' (`action`) the 'mark' (`specifier`)'s 'color' to be 'red' (`option`).

In Section 6, we provide a complete walk-through of the "Bond Yields" example; twelve additional examples are available in the Supplemental Material. We chose properties and values for the `specifier`, `action`, and `option` in a principled fashion based on these example use cases (Section 6) and prior work [13, 16]. As a Cicero specification is independent of the structure of the source visualization, Cicero's properties and values can be extended in the future as needed.

## 4.1 Specifier: Selecting elements to transform

A `specifier` indicates which elements to transform on the target visualization. A `specifier` should only express existing element(s) from the target view, which the compiler then uses to identify the corresponding element(s) and extract relevant properties. Authors tend to apply responsive transformations to groups of element(s) sharing the same role, such as axis labels, mark tooltips, or legend marks, as characterized in prior work [13, 16]. In addition, authors may want to include transformations specific to some data features (e.g., mark labels for specific data points, the axis corresponding to a particular data field) and/or the visual attributes of the visualization element(s) (e.g., red-colored bars). To express visualization elements using different characteristics, one can declare a specifier by *structure*, *data*, and *attribute* queries.

**Structure query:** Many declarative visualization grammars like ggplot2 [37], Vega [34], and ZingChart [44] define roles for visualization elements (e.g., marks, axes). Structure queries identify elements based on this role, and provide additional flexibility for selecting and grouping elements in different ways, regardless of how the original visualization specification define them (**D2**: flexible). Keywords for structure queries include `role`, `mark`, `index`, and `id`.

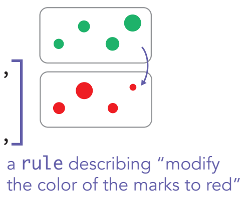
The `role` keyword specifies the role of a visualization element (see Figure 5b). The `role` can be cascaded to specify subordinate elements like `"mark.label"` for labels associated with the visualization marks or `"legend.mark"` for legend marks. For brevity, cascaded role keywords can be shortened when its parent role is unambiguous (e.g., `"layer.mark"` as `"mark"`; `"view.row"` as `"row"`, possible short forms are indicated as gray-colored and parenthesized in Figure 5b). The `mark` keyword specifies the type of mark, which is useful when there are multiple mark types in a visualization. One can include the `index` keyword to indicate the specific element to select from a group of related elements (e.g., `{role: "title", index: 1}` selects the second title element). To indicate the first and last element, one can use `"first"` or `"last"` for the index value. Using `"even"` and `"odd"` can express every other (even and odd) element, respectively. The `id` keyword selects informational marks (`emphasis`) by their defined names or identifiers (e.g., line 43 in Figure 5).

**Data query:** A data query can reference a subset of data (`data`), a data field (`field`), the type of a variable (`datatype`), and values for elements (`values`) to support varying level of customization in selecting visualization elements (**D1**: expressive). For example, the specifier `{role: "mark", data: {price: 30}}` selects all marks that encode a price value of 30. Likewise, the specifier `{role: "axis", field: "price"}` expresses axes for the `price` field; `{role: "legend", datatype: "nominal"}` selects legends for nominal data variables. The `values` keyword expresses a subset of values for a reference element that is tied to a certain data field like axis and legend . For instance, the specifier `{role: "axis.label", values: [30, 50]}` indicates the labels of an axis that encode value of 30 or 50. Similar to the `index` keyword for a structural query, one can use `"even"` and `"odd"` to specify every other (even and odd) value element. In order to support more complex data queries, we also provide a set of logical (NOT, AND, OR), arithmetic (=, ≠, ≤, ≥, ≤, ≥), and string operations (`regex` pattern, `startsWith`, `includes`, `endsWith`) that can be composed to further select and filter elements based on properties of the data (**D2**: flexible).

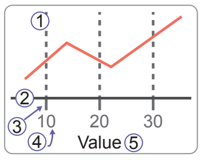
**Attribute query:** An attribute query references visualization elements based on their properties or attributes. The primary attribute query keywords for identifying properties of visualization elements are: `channel`, `operation`, and `interaction`. The `channel` keyword indicates whether the element has a certain encoding channel. For instance, the specifiers `{role: "layer", channel: "color"}` and `{role: "legend", channel: "color"}` indicate layers and legends with a color encoding channel, respectively. The `operation` keyword captures the type of data transformation operations applied to the elements (e.g., filter, aggregate), and the `interaction` keyword expresses the type of interaction features (e.g., zoom, interactive filter). Cicero also supports the use of style and position attribute keywords such as color, font size, orient, relative positions etc. (see `$OtherAttributes` in Figure 6). For marks, those style attributes can be used to indicate mark properties (e.g., static color value or color encoding channel). For example, `{role: "mark", color: "red"}` indicates red-colored marks.

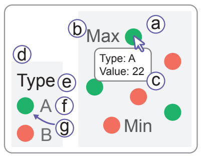
(a) Cicero specification overview

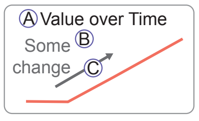
```
{ name: “exampleSpec”,
  metadata: {
    condition: “small”,
    aspectRatio: “portrait” },
  transformations: [
    { specifier: { role: “mark” },
      action: “modify”,
      option: {
        color: { value: “red” }}},
    ... // more rules
  ]}
```



(b) Cicero role expressions



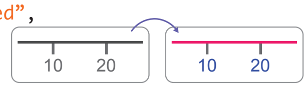




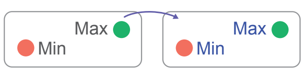
| | |
|---|---|
| data | data sets |
| (data.)transform | transformations on raw data (e.g., filtering) |
| view | view/layout |
| (view.)row | the row elements of a view |
| (view.)column | the column elements of a view |
| (view.)facet | the facets of a multiple-view chart |
| (view.)axis | the axes of a view |
| (view.)hAxis | the horizontal axes of a view |
| (view.)vAxis | the vertical axes of a view |
| [axis].grid ① | the grid lines of axes |
| [axis].domain ② | the domain lines of axes |
| [axis].tick ③ | the tick lines of axes |
| [axis].label ④ | the labels of axes |
| [axis].title ⑤ | the titles of axes |
| [axis] = (view.)axis/hAxis/vAxis | |
| (view.)layer | the layers of a view |
| (view.)layer.transform | transformations on data for layers |
| (view.layer.)mark ⓐ | the marks of layers |
| (view.layer.)mark.label ⓑ | the text labels attached to marks |
| (view.layer.mark.)tooltip ⓒ | the tooltips attached to marks |
| (view.layer.)legend ⓓ | the legends of layers |
| (view.layer.)legend.title ⓔ | the titles of legends |
| (view.layer.)legend.label ⓕ | the labels of legends |
| (view.layer.)legend.mark ⓖ | the marks of legends |
| (view.)title Ⓐ | the title of a view |
| (view.)annotation Ⓑ | the non-data text annotations |
| (view.)emphasis Ⓒ | the non-data informational marks |

(c) Example transformations

```
{ comment: “modify axis labels’ color to blue
  and axis domains’ color to red”,
  specifier: { role: “axis” },
  action: “modify”,
  option: {
    label: { color: { value: “blue” },
    domain: { color: { value: “red” }}}}
```

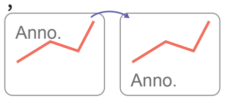


```
{ comment: “modify mark labels’ color to blue”,
  specifier: { role: “mark” },
  action: “modify”,
  option: {
    role: “label”,
    color: { value: “blue” }}}
```

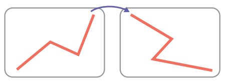


(c) Example transformations (continued)

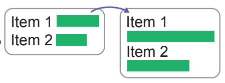
```
{ comment: “externalize annotations”,
  specifier: {
    role: “annotation” },
  action: “reposition”,
  option: { external: true }}
```



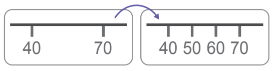
```
{ comment: “transpose axes”,
  specifier: { role: “view” },
  action: “transpose” }}
```



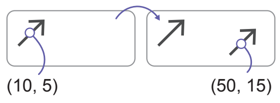
```
{ comment: “transpose axes (equivalent)”,
  specifier: { role: “layer” },
  action: “swap”,
  option: {
    channel: [“x”, “y”]}}

{ comment: “serialize label-marks”,
  specifier: { role: “mark.label” },
  action: “transpose”,
  option: { serial: true }}
```



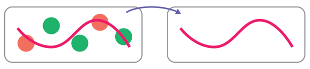
```
{ comment: “add values of 50 and 60 to axis”,
  specifier: { role: “axis” },
  action: “add”,
  option: { values: [50, 60] }}
```



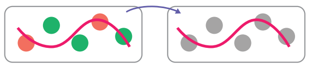
```
{ comment: “duplicate an arrow mark (non-data)”,
  specifier: {
    role: “emphasis”,
    id: “arrow” },
  action: “duplicate”,
  option: { x: 50, y: 15 }}
```



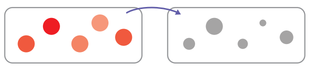
```
{ comment: “remove marks with a color channel”,
  specifier: {
    role: “mark”,
    channel: “color” },
  action: “remove” }
```



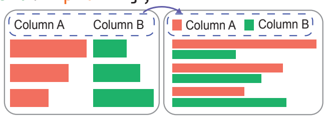
```
{ comment: “remove the color channel of marks”,
  specifier: {
    role: “mark” },
  action: “remove”,
  option: {
    channel: “color” }}
```



```
{ comment: “convert color channel to size channel”,
  specifier: {
    role: “mark” },
  action: “replace”,
  option: {
    channel: { from: “color” , to: “size”}}}
```



```
{ comment: “replace axis label with color legend”,
  specifier: {
    role: “axis.label”, field: “plan” },
  action: “replace”,
  option: {
    to: {
      role: “legend”,
      channel: “color” }}}
```



```
{ comment: “exchange color and size channels”,
  specifier: { role: “mark” },
  action: “swap”,
  option: {
    channel: [“color”, “size”]}}
```

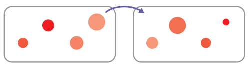


**Figure 5: Examples and roles in the Cicero grammar. (a) An overview of a Cicero specification with a rule describing “modify the color of the marks to red”. (b) role expressions used in Cicero. (c) Example transformations referred to in Section 4.**

```
CiceroSpec := Name?, Metadata?, Transformations

Name := <String>

Metadata := Condition?, MediaType?, AspectRatio?, …?
Condition := xsmall | small | medium | large | xlarge | …
MediaType := screen | paper | …
AspectRatio := portrait | landscape | <Number> | …

Transformations := <Rule>[]
Rule := Specifier, Action, Option?
                                            ------- Structure query
Specifier := Role, Mark?, Index?, Id?,      ------- Data tquery
    Data?, Field?, Values?, Datatype?,     -
    Channel?, Operation?, Interaction?,    ------- Attribute query
    $OtherAttributes?*

Action := modify | reposition | transpose | add
    | duplicate | remove | replace | swap

Option := Specifier° | To?, From?
To := Specifier°
From := Specifier°
```

```
Role‡ := view | layout | layer | mark | …
Mark := point | circle | rect | bar | line | …
Index := <Number> | first | last | even | odd
Id := <String>

Data := Datum | <Datum>[]
Datum := { <Field>: (<Any> | <Any>[] | <Op>[]) }
Op:= { <Operator>: <Any> }
Operator := not | and | or | == | > | >= | startsWith | …
Field := <String>       Values := <Any>[]
Datatype := nominal | ordinal | quantitative | temporal | …

Channel := x | y | color | size | arc | …
Operation := OperationType | <OperationType>[]
OperationType := filter | aggregate | bin | …
Interaction := InteractionType | <InteractionType>[]
InteractionType := zoom | context | …

$OtherAttributes ~ position, x, y, color, label, title, bin,
    aggregate, scale, fontSize, strokeWidth, …
$OtherAttributes := <Any> | By | Prod
By := <Number> Addition to an existing value
Prod := <Number> Product with an existing value
```

Notation
"a := b, c": a is defined as a tuple of b and c, "a?": a is an optional argument, "…": extensible arguments, "<Abc>": data type,
"a ~ b, c": possible names for a are b and c, "a|b|c": either one of a, b, or c, "<A, B>[]": a list of a tuple oft A and B,
"{}": key-value map (e.g., JavaScript Object, Python Dict), "<Number>": either a number or a string of a number with its unit (e.g., 350, "350px").

Note
*$OtherAttributes include encoding channels, role values, and other appearance-related properties (e.g., font styles, stroke styles, etc.).
°An option and its to and from properties share the same structure as a specifier but with different semantics (see Section 4.2).
‡Possible role names are listed in Figure 5b.

**Figure 6: The formal specification of Cicero. The Supplemental Material provides more detailed description.**

## 4.2 Action & Option: Applying transformations

The action indicates how to change the elements queried by a specifier. We designed Cicero to provide a concise set of action predicates that can encode a large range of transformations (**D1**: expressive). The actions currently supported by Cicero are: modify, reposition, transpose, add, duplicate, remove, replace, and swap, chosen based on prior work [13, 16]. Our aim was to support a minimal set of action predicates from the prior work [13, 16]. For example, reposition actions in Kim et al. [16] can be efficiently expressed with using a single 'reposition' action and various option properties (e.g., externalize → reposition + external: true and fix → reposition + fix: true). The 'modify' action can also express these changes to positions, yet having a single 'reposition' keyword is likely simpler for authors to remember. This smaller set of action predicates does not sacrifice much expressiveness, as shown in our diverse set of examples in Figure 5, Section 6, and the Supplemental Material.

The option object in a rule further details the change indicated by the action. While the core structure of an option object is the same as a specifier, the structure and keywords vary based on the type of action. Keywords used in an option object refer to the properties or subordinate elements of the elements that were identified by the specifier (e.g., axis labels are subordinate elements of an axis), so a compiler should interpret an option object with regard to the specifier.

For example, one can use the role keyword to specify subordinate elements in an option object. An option {role: "label"} means legend labels if the specifier is {role: "legend"} or mark labels if the specifier is {role: "mark"}. When an option does not include the role keyword, then the properties in the option indicate those of the element identified by the specifier. For example, in line 8–9 of Figure 5a, "color" refers to the color of the "mark" (the specifier in line 6), while the color keyword in line 13 of Figure 5c expresses the color property of the marks' (specifier) labels (option). Finally, when role values are used as a keyword in the option, they indicate the subordinate elements of the element specified by the specifier. For instance, in Figure 5c, line 5–6 mean the color of the axes' (specifier) labels and domain lines (option), respectively. The entire transformation rule (line 1–6) states that the compiler should specify all the axes in the chart, and modify the labels' color to be blue and the domains' to be red.

**A `modify` action** changes the properties of an element to specific values, with an associated option object for expressing attributes of the elements selected by the specifier. For instance, one can modify the color of mark labels using the rule in line 8–13 of Figure 5. To make relative changes, including adding or multiplying an attribute value by some value, one can use by and prod operators, respectively. For instance, a user can express modifying the size of the specified marks by subtracting 30 using the by operator: {specifier: {role: "mark"}, action: "modify", option: {size: {by: -30}}}.

**A `reposition` action** is a special type of the modify action designed to more intuitively support common transformations related to position properties like absolute positions (x, y), relative positions (dx, dy), externalization (external, internal), etc. For example, externalizing text annotations can be expressed as line 14–18 in Figure 5c. If a user wants to change the style and position properties together, then a modify action is recommended.

**A transpose action** expresses the relative position of a pair of elements, the relationship of which is defined a priori, like two positional axes (*x* and *y*), labels associated with an axis or marks. A transpose action helps simplify expressions for relational properties. For example, the rule in line 20–22 (Figure 5) transposes the entire channel. The equivalent is to swap the *x* and *y* position channels in layers, as in line 24–28. To serialize labels to their marks, one can use the rule in line 30–33 with a serial keyword in the option. This behavior is the same as adjusting the label positions (relative *x* and *y* values) and mark offsets.

**An add action** adds new elements in a visualization. Since the specifier only expresses existing elements (Section 4.1), the newly added elements are provided in an option object. For example, to express "add values of 50 and 60 to axis", one can use the rule in line 35–38 in Figure 5c. When the existing axis selected by the specifier (line 36) has ticks and labels for each axis value, then the rule should result in adding ticks and labels for those values specified in the option (line 38).

**A duplicate** action copies the element identified by the specifier. If provided, an option indicates the properties for the duplicated element to change after duplication (e.g., repositioning the duplicated element in line 40–45 of Figure 5c). In this case, the option acts as a shortcut for a second modify transformation to update the newly added element.

**A remove action** removes elements identified by the specifier when no option is provided; when included, the option specifies the properties or subordinate elements that should be removed from the elements identified by the specifier. For instance, line 47–51 of Figure 5c removes all marks that include a color channel (no option is provided); to instead remove the color channel of these marks requires an option to be expressed (line 53–58).

**A replace action** expresses changes to the function of an entity while retaining its attributes. Sometimes, a visualization author may wish to change the role of an element such as changing from axis labels to legends (Figure 3) or changing an encoding channel of the marks to use increased screen space efficiently. There are two types of replace actions: replacing a property with another within an element and replacing the role of an element with another. For the first case, users can use the from and to keyword to indicate the original property and the replacing property. For instance, converting a channel from color to size can be expressed as the rule in line 60–65 (Figure 5c). Second, authors often change the role of elements across the visualization structure, which requires an option to not be subordinate to the specifier. In that case, users can use a to keyword to indicate that this rule is changing the structural property. For instance, one can replace an axis for the field plan with a legend for the color channel (which is meaningful only when the color channel encodes the same field) by having a rule shown in line 67–74.

```
{ specifier: { role: "view" },
  action: "replace",
  option: {
    from: {
      role: "column",
      index: 0 },
    to: {
      role: "row",
      index: 1 }}},
```

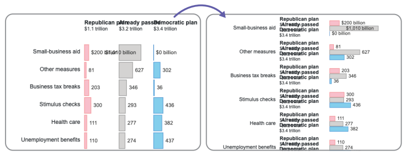


**Figure 7: An example Cicero rule describing partial transpose. The bars are grouped by columns in the left view (before) and by rows in the right view (after). The entire set of transformations for this case (Aid Budget) can be found in the Supplemental Material.**

**A swap action** exchanges two entities (roles and encoding channels) while retaining their properties, which shortens two replace actions and helps avoid potential conflicts. While a swap action has the same option structure with a replace action, it can also use an array to indicate properties to be swapped. For instance, to exchange the color and size channels, one can have a swap action and an array-based option as shown in line 77–81 (Figure 5c).

## 4.3 Reusability of Cicero Expressions

Responsive transformation strategies differ in how well they generalize across visualizations. Sets of public-facing Web visualizations often appear together in a data-driven article and may share data sets, chart types, and style schemes, thereby facilitating transformation reuse. For example, the data filtering rule in line 5–10 of Figure 13 can be reused for other charts sharing the same data set because it references the data fields (year, forecasted_year) directly. However, this rule cannot necessarily be reused on charts with different data sets. On the other hand, authors can reuse the partial axes transpose rule in Figure 7 for charts with a similar format regardless of the underlying data set as the transformation is declared independently. The flexible specifier syntax of Cicero is designed to allow authors to express more reusable transformations. For instance, the transformation for adding axis values in line 35–38 of Figure 5c can be reused on neighboring charts to provide better consistency. Alternatively, one can express the same rule as `{specifier: {role: "vAxis"}, action: "add", option: {index: "odd"}}` to make the rule more generalizable by not making direct reference to the underlying data scheme. Expression reusability is a core attribute of Cicero that naturally supports sophisticated visualization authoring tools, such as recommender systems, which we discuss further in Section 7.

## 5 PRINCIPLES FOR OUR CICERO COMPILER

To demonstrate the feasibility of Cicero and our proposed approach, we developed a compiler for our extended implementation of Vega-Lite. In the process, we identified ten principles we considered when implementing our Cicero compiler. As outlined in Figure 8,our prototype Cicero compiler takes as input a Cicero specification and a visualization design specification written in our extended Vega-Lite. Then, the Cicero compiler returns a transformed design specification in our extended Vega-Lite, which is eventually rendered by the compiler of our extended Vega-Lite. For each transformation rule,

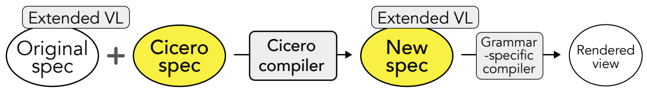


**Figure 8: The pipeline for our Cicero compiler, developed for our extended version of Vega-Lite.**

our compiler first selects an element(s) indicated by the `specifier`. If the element(s) exists, then the compiler applies the changes specified by the `action` and `option`. While developing the prototype compiler and deriving the principles below, we examined examples from prior work [13, 16] and considered how our compiler should deal with downstream effects to associated elements, the default behavior of a rendering grammar, and conflicting transformation rules. Future work can leverage our principles as useful semantics of the Cicero grammar when implementing custom Cicero compilers for other declarative visualization grammars. We describe our custom Cicero compiler API in the Supplemental Material (https://osf.io/eg4xq).

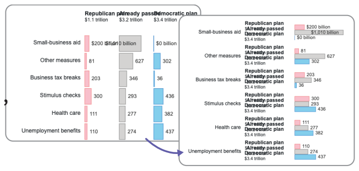
**Our extended version of Vega-Lite** provides a set of workarounds for public-facing visualization technique, such as text-wrapping and supplemental text (captions), that are currently not supported in Vega-Lite [33], but were needed for our examples (e.g., external annotations). We use this extension to demonstrate the capabilities of Cicero for real-world use cases. The key differences from the current Vega-Lite are that our extension (1) uses trellis plot-based layouts [3] (rows and columns) instead of $x$ and $y$ encodings, (2) has many shortcuts to design techniques (e.g., wrapping text, map visualizations, interactive filters) for which Vega-Lite currently requires further specifications, and (3) supports richer communicative functionalities such as defining supplemental text elements like multiple subtitles or captions, creating graphical emphases that are not bound to data, allowing different formats of labels in the same axis, and so forth. The formal specification and description of our extended Vega-Lite are in the Supplemental Material.

## 5.1 Associated elements

Visualization elements can have associations between them, which should inform how our Cicero compiler selects and handles the elements. For example, axis labels are dependent on the range of visualized data encoded by the $x$ and $y$ positions; hence, axis labels are associated with the ranges of visualized data values (line 15–21 of Figure 13). When a subset of data is omitted under a responsive transformation, then text annotations attached to the corresponding marks should be omitted as well (line 5–10 of Figure 13).

We describe two principles involving associated elements. First, our Cicero compiler **detects associated elements depending on how a user has defined the original design (P1)**. In the previous example (Figure 13), the two longer labels are declared as `text` elements of the line marks (i.e., tied to the marks in the same layer; `{type: "on-mark", field: "forecasted_year", items: [...], ...}`). Thus, filtering out a subset of data subsequently removes the corresponding marks and their associated labels. On the other hand, if the user has declared the text elements directly (without anchoring to certain data points), then the compiler should interpret them as independent elements that are not subordinate to any other element(s) or data.

```
1 { specifier: {
2     role: "view" },
3   action: "replace",
4   option: {
5     from: { role: "column",
6             index: 0 },
7     to: { role: "row",
8           index: 1 }}},
9 { specifier: { role: "row", field: "plan" },
10  action: "modify",
11  option: {
12    sort: {
13      sortBy: ["Already passed",
14               "Republican plan",
15               "Democratic plan"] }}},
```



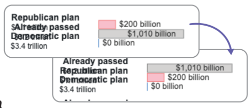


**Figure 9: An example case (Aid Budget) for a downstream effect to the layout of elements (moving a column axis to a row axis; line 1–8) and applying a rule (reordering a nominal $y$ axis) to the previously transformed view (line 9–15).**

Second, **a transformation affecting the layout of a series of elements**, such as adding, removing, or repositioning, **has a downstream effect on the layout of their associated elements (P2)**, but not the static style. We do not allow downstream changes to style because the layout of one element and the static style of another are not meaningfully related whereas the relative layout between different elements does have a meaningful relationship. In the previous example, filtering out data points should not impact any independent, non-data annotations but should remove any associated text element(s). Similarly, converting a field from the column to the row of the chart (partial transpose) should move the axis labels (defined as `{type: "on-axis", field: "plan", items: [...], ...}`; i.e., tied to the axis of the plan field) for the field accordingly (see line 1–8 in Figure 9), but should not have side effects to their other properties—like the font weight or font size.

## 5.2 Default behaviors

Declarative grammars often have default behaviors to make it easier to create a visualization. For example, Vega-Lite automatically generates legends and axis labels as a user declares color/size and position encoding channels. In compiling a Cicero specification, we were able to relatively easily reason about default behaviors regarding removing, modifying, and externalizing actions (e.g., "modify only what is specified" as a general software quality guideline or "externalize annotations at the bottom of the chart unless specified otherwise" based on our examples). However, adding a new element and internalizing an element can complicate the compile process, particularly when a user has underspecified the behavior. For example, when a user adds a new text annotation in the chart without specifying its position, then it is unclear how our Cicero compiler should behave. To guide such complex situations, we used a set of high-level default behaviors for our Cicero compiler.

First, **when adding a new element to a series of elements, its appearance should mimic the existing elements in the series (P3)**. For example, line 7–9 in Figure 10 adds new values for a vertical axis, resulting in newly added grid lines and labels. Then, they should look similar to the existing grid lines and labels

```
1 { specifier: { role: "title" },
2   action: "replace",
3   option: {
4    to: { role: "annotation",
5          internal: true },
6    separate: false }},

7 { specifier: { role: "vAxis" },
8   action: "add",
9   option: { values: [50, 150, 250] }},
```

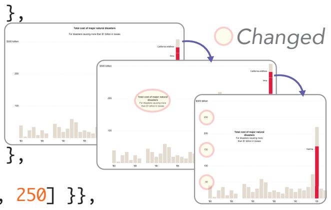


**Figure 10: An example use case (Disaster Cost) for our Cicero compiler's default behavior for replacing the title as an internal annotation (line 1–6) and for introducing newly added elements (axis labels and grid lines; line 7–9).**

without further specifying their appearance. Our Cicero compiler performs this addition by including those values in line 9 to the axis label and grid component in the specification (i.e., {..., values: [100, 200, 300], ...} → {..., values: [50, 100, 150, 200, 250, 300], ...}).

Second, our Cicero compiler **considers the appearance of elements in a similar role for new elements that are not part of an existing series of elements (P4)**. For example, when adding labels to a $y$ axis that has no existing labels, although they are not in the same series, it is more sensible to set their appearance similar to the labels on the $x$ axis rather than the default presets of the rendering grammar. The similarity of the role between two series can be determined by whether they can be specified as the same `role` keyword (e.g., {role: "axis.label"} can specify both {role: "hAxis.label"} and {role: "vAxis.label"} if they both exist). Then, our compiler reuses the appearance attributes of the similar series of elements.

Third, **when there are multiple series of existing elements, our Cicero compiler selects the one with the most similar structure (P5)**. As shown in Figure 11, for instance, when adding a new label to an axis that has two groups of existing labels in different styles, our Cicero compiler reasons about which of the two groups is most similar to the new label. We use the number of subelements (e.g., text segments) and the format of elements to find the most similar series of elements. For our approach, the compiler first identifies the number of newly added text segments (two). The one starting with "Jan. 19 ..." has two segments with different styles, and the "Feb. 29" one has a single segment. Then, by comparing the numbers of segments, the compiler matches the two-segment one ("Jan. 19 ...") with the new labels.

Lastly, we consider the case where the position and style of a newly added or repositioned element cannot be fully determined because there is no existing series with a similar role. In this case, the compiler should leverage the following default behavior if not specified otherwise: as an overarching principle, **use the default options of the rendering grammar's compiler (P6)** for newly added elements because users are expected to have some basic knowledge about how the rendering grammar behaves. For example, our extended Vega-Lite implementation does not automatically generate a legend for a new color scale, so our Cicero compiler for this extension similarly does not introduce a legend when adding a new color encoding. On the other hand, Vega-Lite's default is to include a legend, so a Cicero compiler for Vega-Lite *should* add a

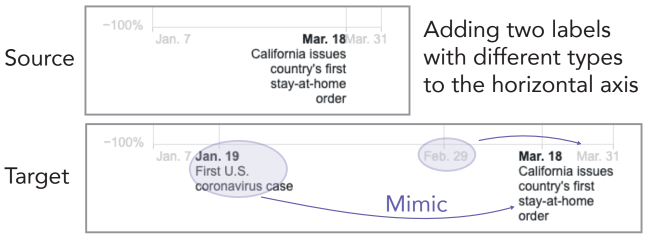

Source
-100%
Jan. 7
Mar. 18
Mar. 31
California issues country's first stay-at-home order
Adding two labels with different types to the horizontal axis
Target
-100%
Jan. 7
Jan. 19
First U.S. coronavirus case
Feb. 29
Mimic
Mar. 18
Mar. 31
California issues country's first stay-at-home order

**Figure 11: An example use case (Covid Spending 1) for our Cicero compiler's treatment of multiple series of existing elements. In this case, our Cicero compiler adds new axis labels by mimicking the most similar type of the existing axis labels according to the number of subelements (text lines).**

legend. We had the following default behaviours for cases where the rendering grammar has no relevant default options based on our observations of common responsive design principles:

- Place (new) externalized annotations below the chart (see a4 in Figure 14).
- Place (new) internalized data annotations (or mark labels) at the center or the bottom of the associated data mark (see c4 in Figure 14).
- Place (new) internalized non-data annotations at the center of the largest contiguous empty space in the chart (see line 1–6 in Figure 10).

## 5.3 Conflict management

Cicero's list-based specification explicitly indicates the order of declared transformation rules. However, there are some cases where the order of rules may impact how the Cicero compiler interprets a given specification. Our compiler solves conflicts using the following methods, some of which are inspired by relevant CSS principles [22] that similarly deal with managing conflicts between ordered rule items. First, it may be confusing to select visualization element(s) in a `specifier` when other rules in the specification also transform the same element, which differs from general CSS use cases. For example, suppose there is a rule to transpose the $x$ and $y$ positions. This rule also results in swapping the horizontal and vertical axes as they are associated with the $x$ and $y$ position encoding channels. If a user wants to make some design changes in an axis that is the horizontal axis after transposing but is the vertical axis before transposing, defining a specifier for this rule might be confusing. A simple approach defaults to always specifying what is in the original view specification or what will appear in the transformed view. However, the former may not be useful for cases like making further changes to a newly added element, and the latter might make it difficult to compose a specification by requiring users to imagine the outcome status. As an overarching principle, our compiler **applies the current rule to a view that has been transformed by the previous rules (P7)** (e.g., line 1–8 and line 9–15 in Figure 9). This approach also implies that the compiler **applies the last declared rule (P8)** when there are two rules making changes to the same element for the same property, which is also a common practice with CSS specifications. Our compiler

(a) More specific

```
{ specifier: {
    role: "mark",
    datum: {
      cat: "Apparel" },
  action: "modify",
  option: {
    color: {
      value: "red"}}}
```

(b) Less specific

```
{ specifier: {
    role: "mark" }
  action: "modify",
  option: {
    color: {
      value: "gray"}}}
```

**Figure 12: Rules to change the color of marks (a) by specifying the mark for the "Apparel" category and (b) by generally changing the color of all marks (independent of the data).**

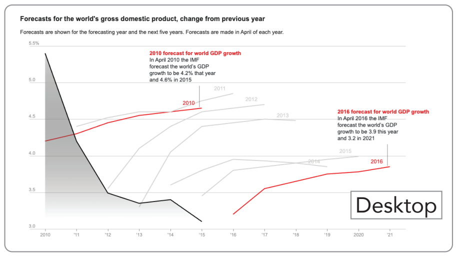
handles this principle by updating the target view specification for each transformation rule.

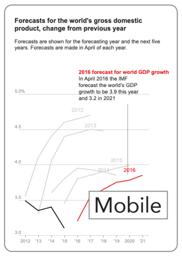
Next, our compiler **assigns higher priority to a more specific rule than a more generic rule for the same element (P9)** (note: not the same specifier)[1]. Here, the more attributes a `specifier` has, the more specific the rule is, inspired by CSS principles [23]. For example, suppose a user wants to change the color of a mark for the "Apparel" category (rule (a) in Figure 12) as well as changing the color of all bars (rule (b)). Here, the mark for "Apparel" is affected by both rules. Therefore, we recommend that generic color changes to other bars should not be applied to the mark for the "Apparel" category (i.e., rule (a) has higher priority than (b)). If a user does not want to apply a specific change (e.g., the custom color for the "Apparel" mark), then the user should omit the rule from the Cicero specification. Lastly, to enhance the degree of freedom in indicating the priority of rules, Cicero provides an `important` property for the same specifier, inspired by the `!important` keyword in CSS [23]. **Rules with the `important` property set to `true` have higher priorities than others (P10)** (i.e., compiled at the end). For example, a rule that changes the color of every axis label with `{important: true}` overrides another following rule that recolors a specific axis label[2]. We refer the reader to the Supplemental Material for the full details on how our Cicero compiler for the extended version Vega-Lite exhibits these principles.

## 6 REPRODUCING REAL-WORLD EXAMPLES

To demonstrate the expressiveness, flexibility, and reusability of Cicero and illustrate the above principles of our Cicero compiler, we present an in-depth walk-through of a mobile-to-desktop example (Bond Yields) using our extended version of Vega-Lite as the rendering grammar. We have twelve additional real-world inspired walk-through specifications that show the responsive changes step-by-step and two other detailed textual walk-throughs in the Supplemental Material that exhibit a variety of other transformations to visualization elements (i.e., data, marks, axes, title, labels, annotations, informational marks/emphasis, interaction, etc.) for both desktop-to-mobile and mobile-to-desktop transformations. The total of 13 example use cases includes three from Hoffswell et al. [13], four cases from Kim et al. [16], three recent responsive visualization cases (in our extended Vega-Lite), and two additional cases

[1]See the 'Justice Kennedy' case (desktop to mobile) in our Supplemental Material.
[2]See the 'Disaster Cost' case (desktop to mobile) in our Supplemental Material.

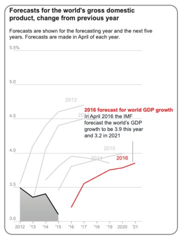


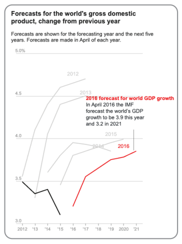


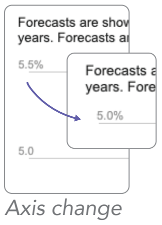
Cicero transformations

```
1  ...
2  { specifier: { role: "view" },
3    action: "modify",
4    option: { size: [365, 450] }},

5  { specifier: {
6      role: "data",
7      data: [
8        year: { leq: 2011 },
9        forecasted_year: { leq: 2011 }]
10   action: "remove" },

11 { specifier: {
12     role: "mark",
13     mark: "area" },
14   action: "remove" },

15 { specifier: {
16     role: "row",
17     field: "growth", }
18   action: "modify",
19   option: {
20     scale: {
21       domain: [3, 5] }}},

22 { specifier: {
23     role: "mark.label",
24     mark: "line",
25     text: {
26       startsWith: "2016 forecast for" }},
27   action: "reposition",
28   option: {
29     dx: { by: -10 },
30     dy: { by: -40 }}}
31 ...
```

Transformed







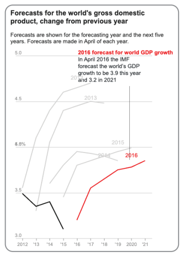


**Figure 13: A walk-through example case of Bond Yields from a desktop version (top left) to a mobile version (top right). Starting with the desktop version, we first resize the chart to fit to a mobile screen (line 2–4), remove a subset of data for earlier years (line 5–10), remove the area mark (line 11–14), update grid lines by rescaling the domain of the *y* position channel (line 15–21), and reposition the annotation (22–30).**

from the Vega-Lite example gallery that were not originally responsive but demonstrate the generalizability of our Cicero specifications to refine complex source views (in Vega-Lite). All 13 cases are listed in Figure 1 and provided in the Supplemental Material (https://osf.io/eg4xq).

### 6.1 A Walk-through Example: Bond Yields

The Bond Yields example[3] visualizes changes to both the actual and forecasted GDP growth rates over time. In the desktop version (Figure 13), the *x* position encodes the year from 2010 to 2021, and the

[3]https://www.wsj.com/graphics/how-bond-yields-got-this-low/

$y$ position indicates the GDP growth rate from 3.0 to 5.5. The area mark and black line mark represent the actual GDP growth rate from 2010 to 2015. The red and gray lines represent the five-year forecast of GDP growth rate for each year from 2010 to 2016; for example, the leftmost red line shows the estimated GDP growth rates for 2011 to 2015, as forecast in 2010. Transformations to produce the mobile version include (1) reducing the chart size, (2) removing the data points and labels for the forecast year of 2010 and 2011, (3) omitting the area mark, (4) truncating the $y$ axis, and (5) repositioning an annotation. The Cicero spec is shown in Figure 13a.

First, to resize the chart for a mobile phone, one can apply a `modify` action to the entire `view` (line 2–4). The `option` object indicates the `size` of 365 (width) × 450 (height) to ensure that the chart fits a mobile phone without requiring horizontal scrolling. Alternatively, one can use `{width: 365, height: 450}` in the `option`. Then, line 5–10 filters out (`remove`) the specified data points to simplify the view by reducing the information density. The `data` keyword in the `specifier` means ⟨year ≤ 2011 (for the actual GDP growth rate) OR forecast year ≤ 2011 (for the forecast)⟩. Filtering out the data points removes (1) the two simple line marks for the forecast year of 2010 and 2011, (2) the data annotation for forecast year 2010, and (3) the corresponding parts of the area and black line mark for the actual GDP growth rates because each of these elements is associated with the filtered data (**P2**). This association is determined by the original visualization structure; if the annotations were declared as non-data elements, then the annotation for the 2010 forecast would remain (**P1**).

The `remove` transformation in line 11–14 omits the area mark specified by the `mark` keyword. After filtering the earlier data, there is wasted space along the y-axis that unnecessarily compresses the data. To address this issue, the rule in line 15–21 changes the scale domain of the row field (`growth`) to `[3,5]`, resulting in the removal of the axis label and grid line for 5.5; the remaining elements automatically adjust to fill the newly vacated space (**P6**).

Lastly, the `reposition` rule in line 22–30 moves the mark label. Because there are many text elements associated with data marks (e.g., year names for each line), a specific `text` query is needed to select the label to move. For this rule, one can use the `startsWith` operator (line 25–26) to select elements with text starting with the specified string. Then, the `option` object changes the relative horizontal and vertical position (`dx` and `dy`, respectively) using the `by` operator which adds the specified value to the original value (i.e., moving the element by 10px left and by 40px upward).

## 7 POTENTIAL APPLICATIONS FOR CICERO

Declarative grammars are particularly valuable for their utility in applications like visualization recommenders and authoring tools. In particular, they can function as a common representation method for different intelligent tools with similar purposes [41]. Visualization systems often use their own "internal representation" methods for their specific purposes [41]. Suppose we have two recommender models for different parts of a visualization (e.g., one for chart types and the other for annotations and emphases) that use heterogeneous representation methods. If they are translated to Cicero, then their recommender outcomes could be effectively combined to a user-side application. In this section, we describe how we used Cicero to represent a design space of responsive transformations in a prototype design recommender for responsive visualization as a proof of concept. We further discuss how Cicero might support mixed-initiative authoring tools.

### 7.1 Responsive Visualization Recommender

As a case study for potential applications for Cicero, we developed a recommender prototype for responsive visualization transformations using Answer Set Programming (ASP), which represents knowledge in terms of facts, rules, and constraints [6]. Our recommender takes a source visualization specification expressed in our extended version of Vega-Lite along with configuration preferences (e.g., intended screen size, strategies that a user wants to avoid, and a subset of data that can be omitted) which could hypothetically be provided by a user. Our recommender is intended to provide a diverse set of recommendations rather than showing several "optimal" visualization with slight differences. We encoded a set of common responsive visualization strategies motivated by prior work [13, 16] in ASP. Given the inputs and encoded strategies, Clingo [10], an ASP solver, generates a search space of responsive transformation strategy sets (corresponding to responsive visualization designs). To rank these strategy sets, we encoded heuristic-based costs that apply to individual strategies, and normalize and aggregate these costs to rank strategy sets representing design alternatives. We implemented three types of costs that apply to individual strategies: "popularity" costs based on the frequency of the strategy in prior analyses of professionally-designed responsive visualizations [13, 16]; density costs, where strategies that reduce information density are assigned lower cost than those that do not in a desktop-first pipeline, and vice versa in a mobile-first pipeline; and message preservation costs, where strategies (e.g., axis transpose, disproportional rescaling) are assigned costs based on the extent to which prior work proposes that they affect the implied "message" of a visualization [16, 17].

In this pipeline, each recommended strategy set in the ASP format (e.g., `do(transpose_axes).`) are translated to a Cicero spec (e.g., `{specifier: {role: "view"}, action: "transpose"}`). While inference engines or models (e.g., ASP, ML, etc.) often employ their own abstract expressions for computational purposes, systems need to translate such abstract expressions (e.g., to JavaScript, Python, etc.) before utilizing them. For instance, ASP can efficiently perform logic problems, but the ASP expressions cannot be directly used to execute actual tasks without translation. In the context of responsive transformation, directly using ASP codes to transform a visualization design specification (i.e., running JavaScript codes for each ASP code) is likely to complicate the translation, lacking modularization. For example, whenever a recommender adds a new transformation strategy, the system has to look at every detail of different use cases, and doing so may not be consistent with the existing transformation strategies. This inconsistency in turn makes it more difficult to debug and extend the recommender. Instead, if we can translate those abstract transformations to systematic expressions like the Cicero grammar, then implementing recommenders for responsive visualization only needs to focus on generating a search space by modularizing the translation process.

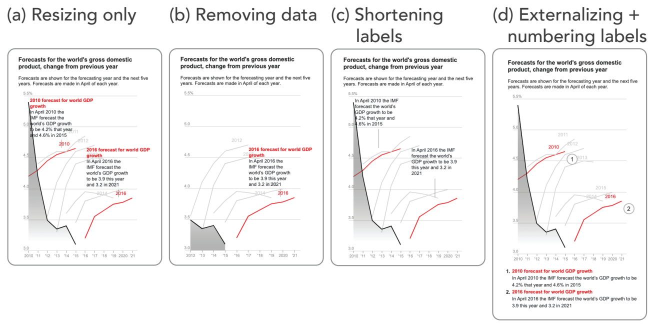


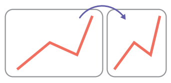
**Figure 14: Selected examples among top seven recommendations for Bond Yields case from desktop to mobile. The original design is shown in Figure 13.**

This process is similar to how Draco translates ASP expressions to Vega-Lite [33] and then renders a visualization [28].

Below, we illustrate example recommendations (Figure 14a) using our walk-through example (Section 6). We provide further details on our prototype recommender implementation, and describe example recommendation cases below and in Supplemental Material (https://osf.io/eg4xq). We emphasize, however, that our goal in developing the prototype recommender is to demonstrate the feasibility of using Cicero in such an approach, rather than to argue for the specific implementation of the cost model we used. In other words, our recommender should be interpreted as a proof of concept of our approach, rather than as an ideal recommender.

*7.1.1 Example: Bond Yields.* To generate candidate mobile views for the Bond Yields case, we include in the configuration the target size of a mobile view and a subset of data that can be omitted (referring to the original design). The first recommendation (Figure 14a) is simply resized to the target size. For this change, our ASP recommender returns `do(set_width,365).` and `do(set_height,450).`, and these abstract descriptions are translated to corresponding Cicero rules: `{specifier: {role: "view"}, action: "modify", option: {width: 365, height: 450}}`. In the second recommendation (b), the suggested omission is applied, similar to the original mobile view except for the remaining area mark and axis value for 5.5%. Our ASP engine expresses the transformation in an abstract way (`do(add_filter,f0).`, where `f0` is a pointer to the user-suggested data filter statement), and then it is converted to a proper Cicero rule, `{specifier: {role: "data", data: [...]}, action: "remove"}`. The data annotations for the forecast years of 2010 and 2016 are shortened by removing the first line (the red text) in the third recommendation (c). For this change, our recommender converts an ASP rule, `do(remove_text_line,t2,0).` where `t2` is a pointer to the annotations (or mark labels), to a Cicero rule: `{specifier: {role: "mark.label", field: "forecasted_year", index: 2}, action: "remove", option: {items: {index: 0}}}`. The fourth recommendation (d) externalizes the same data annotations below the chart with numbering for reference to the data marks. For this transformation, ASP rules, `do(externalize,t2).` and `do(numbering,t2).`, are translated to a Cicero rule: `{specifier: {role: "mark.label", field: "forecasted_year", index: 2}, action: "modify", option: {external: true, number: true}}`. If the ASP rules were not compiled into our modularized Cicero grammar, the required changes to the original visualization specification would need to directly dissect many different parts of the specification, such as data, annotations, and axes. By modularizing this computation, Cicero can provide a more systematic representation of those changes, which helps extend and debug our recommender.

```
1 // A0&2: decrease X axis range
2 { specifier: { role : "view" },
3   action: "modify",
4   option: { width: 375 }}
5
6 // A9: decrease font size
7 { specifier: { role : "text" },
8   action: "modify",
9   option: { fontSize: { prod:  0.8 }}}
```



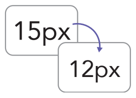


**Figure 15: Expressing transformation strategies of MobileVisFixer [40] in Cicero. Line 2–4: decreasing the range of the $x$ axis by reducing the width of the chart. Line 7–9: decreasing the font size using `prod` keyword.**

*7.1.2 Generalizability for Recommenders.* Cicero can enhance modularization of responsive visualization tools by connecting tool-specific expressions and visualization grammars. For example, our recommender prototype uses ASP [6] to encode expressions with the Clingo solver [10]), and the Cicero compiler connects recommendations expressed in ASP to visualizations in our extended Vega-Lite. Future work might start to leverage Cicero with machine learning-based recommenders. For instance, Cicero can express reusable transformation rules in MobileVisFixer [40] that translates non-responsively designed visualizations to mobile views. As shown in line 2–4 of Figure 15, Cicero expresses 'reducing the range of $x$ axis' by expressing the change to the chart width (e.g., 375 pixel for mobile screens). Using the `prod` keyword in line 9, one can express reducing the font size of all the text elements relatively. In the Supplemental Material, we provide a list of reusable Cicero expressions for MobileVisFixer [40] rules of which the meanings are clearly defined.

## 7.2 Mixed-initiative Authoring Tools

Users of visualization authoring tools may prefer different levels of customization and automation [25]. Tools like Microsoft Power BI [9], which automates design recommendations by converting a source visualizations using a set of default strategies, allow quick visualization creation, but can limit design expressiveness. In contrast, while the prototype proposed by Hoffswell et al. [13] and DataWrapper [1] do not have automated recommendation features, they enable more customization in making responsive designs.

Mixed-initiative authoring tools can provide a balance of automation and customization capabilities, by allowing authors the ability to make manual responsive transformations or accept recommender-suggested transformations. Mixed-initiative authoring has been applied in exploratory data analysis (e.g., Voyager [38] and Dziban [20]) and dashboard design (e.g., LADV [21]) settings. While our prototype recommender takes as input a representation of users' preferences, a next-generation authoring tool might aim to reason about responsive transformations that the user makes so as to recommend further or alternative transformations. For example, imagine creating the Bond Yields case (Section 6.1) without data filtering. After resizing, the target visualization might look dense (Figure 14a1) although it maintains more takeaways compared to the actual design. Then, a user might decide to externalize the annotations instead of removing data. Following this manual change, a mixed-initiative authoring tool might suggest numbering the externalized annotations to support finding data references.

A mixed-initiative approach stands to reduce computational complexity by looking at the current state of edits rather than reasoning over a larger space of transformation combinations. Within a mixed-initiative authoring pipeline for responsive visualization, Cicero can be used to represent both system-recommended transformation strategies and user-driven manual edits, which can make such systems easier and more efficient to handle different sources of transformations (system and user). In addition, when an author updates the source visualization, Cicero can be used to reapply previous rules that are generalized to the updated chart (i.e., rules with the specifiers that can make queries from the updated chart).

## 8 LIMITATIONS

While Cicero and the Cicero compiler for our extended version of Vega-Lite can reproduce real-world use cases that represent a diverse set of transformations, future work should apply Cicero and future Cicero compilers to a bigger set of use cases to improve them and further extend the expressiveness of the grammar. For example, future work might focus on expressing complex user interactions (e.g., pan+zoom for a 3D visualization) with `specifier`s, inspired by declarative grammars for interactive visualizations (e.g., `trigger`, `signal`, and `event streams` in Vega [14, 34]), to better facilitate the application of such technologies to Web contexts where they have largely been underutilized [13, 16]. Another interesting future direction could be expressions for bounded dynamic behavior—the sizes or arrangement of elements dynamically change up to a certain limit, such as `max-width` and `flex-wrap` in CSS—in `option`s. As it is a Web browser that implements CSS specifications, additional expressions for bounded dynamic behavior will be useful only if a rendering grammar supports such behavior. Furthermore, new design and evaluation studies for intelligent responsive authoring tools with Cicero might be useful to extend both Cicero and prior approaches in responsive visualization tooling [13, 16, 17, 19, 29, 40, 42, 44].

Next, to demonstrate the full potential of Cicero in Web-based communicative visualizations, we chose to implement an extended version of Vega-Lite that can more easily express common techniques for narrative visualizations, such as externalizing annotations and applying word wrap to text labels. These capabilities are not straightforward to implement in Vega-Lite [13], so the resulting capabilities of a Cicero compiler for Vega-Lite may likewise be limited in what can be expressed in rendered visualizations. As such grammars continue to develop, the corresponding compiler can be refined to support additional responsive functionalities. Furthermore, future work might need to apply these techniques to a larger class of declarative systems, such as extensions based on ggplot2 [37] or Vega [34], to efficiently implement the corresponding Cicero compilers with a better understanding of their capabilities.

Finally, a Cicero specification defines a set of transformations to create a single responsive version and itself is not intended for direct rendering. As multiple responsive versions are necessary for different device types, an authoring system could bundle multiple Cicero specifications as a family using the `metadata` object in the specifications to decide when to apply each of them.

## 9 CONCLUSION

We contribute Cicero, a declarative grammar for specifying responsive transformations from a source to a target visualization. By enabling flexible, expressive, and reusable specifications of visualization transformations, Cicero paves the way for intelligent responsive visualization authoring tools, by providing a concise set of action predicates that enable encoding diverse transformations, flexible specifier syntax for handling the behavior of transformations, and reusability of transformation rules. To demonstrate the utility of Cicero in the context of intelligent visualization tools, we leverage Cicero for a prototype design recommender for responsive transformations. Future work can employ Cicero for a range of responsive visualization authoring tools designed for specific declarative grammars with custom compilers for those grammars.

## REFERENCES

[1] 2012. DataWrapper. https://www.datawrapper.de/ Last accessed: Jun. 2, 2021.
[2] Keith Andrews. 2018. Responsive Visualisation. In *MobileVis '18 Workshop at CHI 2018*. ACM.
[3] Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu. 1996. The Visual Design and Control of Trellis Display. *Journal of Computational and Graphical Statistics* 5, 2 (1996), 123–155. https://doi.org/10.1080/10618600.1996.10474701
[4] Elliot Bentley. 2021. The Web as medium for data visualization. In *The Data Journalism Handbook: Towards a Critical Data Practice*, Liliana Bounegru and Jonathan Gray (Eds.). Amsterdam University Press, 182–192. https://doi.org/10.5117/9789462989511
[5] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185
[6] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. 2011. Answer Set Programming at a Glance. *Commun. ACM* (2011). https://doi.org/10.1145/2043174.2043195
[7] Jay Bryant and Mike Jones. 2012. *Responsive Web Design*. Apress, Berkeley, CA, 37–49. https://doi.org/10.1007/978-1-4302-4525-4_4
[8] E. Di Giacomo, W. Didimo, G. Liotta, and F. Montecchiani. 2015. Network visualization retargeting. In *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*. 1–6. https://doi.org/10.1109/IISA.2015.7388095
[9] Roy Gal. 2017. Responsive visualizations coming to Power BI. https://powerbi.microsoft.com/en-us/blog/responsive-visualizations-coming-to-power-bi/ Last accessed: Jun. 2, 2021.
[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. (2014). arXiv:1405.3694 https://arxiv.org/abs/1405.3694.
[11] Google. 2019. Using Google Charts. https://developers-dot-devsite-v2-prod.appspot.com/chart/interactive/docs Last accessed: Sept. 11, 2020.
[12] Jane Hoffswell, Alan Borning, and Jeffrey Heer. 2018. SetCoLa: High-Level Constraints for Graph Layout. In *Computer Graphics Forum (Proc. EuroVis)*. https://doi.org/10.1111/cgf.13440
[13] Jane Hoffswell, Wilmot Li, and Zhicheng Liu. 2020. Techniques for Flexible Responsive Visualization Design. In *ACM Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3313831.3376777
[14] IDL. 2017. Documentation–Vega. https://vega.github.io/vega/docs/ Last accessed: Sept. 9, 2021.
[15] M. R. Jakobsen and K. Hornbæk. 2013. Interactive Visualizations on Large and Small Displays: The Interrelation of Display Size, Information Space, and Scale. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2336–2345. https://doi.org/10.1109/TVCG.2013.170

[16] Hyeok Kim, Dominik Mortiz, and Jessica Hullman. 2021. Design Patterns and Trade-Offs in Authoring Communication-Oriented Responsive Visualization. *Computer Graphics Forum (Proc. EuroVis)* 40 (2021), 00–00. Issue 3. https://doi.org/10.1111/cgf.14321

[17] Hyeok Kim, Ryan Rossi, Abhraneel Sarma, Dominik Mortiz, and Jessica Hullman. 2021. An Automated Approach to Reasoning About Task-Oriented Insights in Responsive Visualization. *To Appear IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2021). https://arxiv.org/abs/2107.08141.

[18] Younghoon Kim and Jeffrey Heer. 2021. Gemini: A Grammar and Recommender System for Animated Transitions in Statistical Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2021). https://doi.org/10.1109/TVCG.2020.3030360

[19] Juliana Leclaire and Aurélien Tabard. 2015. R3S.js–Towards Responsive Visualizations. In *Workshop on Data Exploration for Interactive Surfaces DEXIS 2015*. 16–19.

[20] Halden Lin, Dominik Moritz, and Jeffrey Heer. 2020. Dziban: Balancing Agency & Automation in Visualization Design via Anchored Recommendations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. 12. https://doi.org/10.1145/3313831.3376880

[21] Ruixian Ma, Honghui Mei, Huihua Guan, Wei Huang, Fan Zhang, Chengye Xin, Wenzhuo Dai, Xiao Wen, and Wei Chen. 2021. LADV: Deep Learning Assisted Authoring of Dashboard Visualizations From Images and Sketches. *IEEE Transactions on Visualization and Computer Graphics* 27, 9 (2021), 3717–3732. https://doi.org/10.1109/TVCG.2020.2980227

[22] MDN. n.d.. Cascade and inheritance. https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Cascade_and_inheritance Last accessed Aug 13, 2021.

[23] MDN. n.d.. Specificity. https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity Last accessed Dec 15, 2021.

[24] MDN. n.d.. Using media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries Last accessed Sept 4, 2021.

[25] Gonzalo Gabriel Méndez, Uta Hinrichs, and Miguel A. Nacenta. 2017. *Bottom-up vs. Top-down: Trade-Offs in Efficiency, Understanding, Freedom and Creativity with InfoVis Tools*. Association for Computing Machinery, New York, NY, USA, 841–852. https://doi.org/10.1145/3025453.3025942

[26] Microsoft. 2011. Power BI. https://powerplatform.microsoft.com/en-us/.

[27] S. Mohorovičić. 2013. Implementing responsive web design for enhanced web presence. In *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1206–1210.

[28] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). https://doi.org/10.1109/TVCG.2018.2865240

[29] Lisa C. Rost. 2020. Create better, more responsive text annotations (yes, also on maps). https://blog.datawrapper.de/better-more-responsive-annotations-in-datawrapper-data-visualizations/ Last accessed: Jun. 2, 2021.

[30] Lisa C. Rost and Gregor Aisch. 2020. Our new Tables: responsive, with sparklines, bar charts and sticky rows. https://blog.datawrapper.de/new-table-tool-barcharts-fixed-rows-responsive-2/ Last accessed: Jun. 2, 2021.

[31] Cedric Sam. 2018. Ai2html and Its Impact on the News Graphics Industry. In *MobileVis '18 Workshop at CHI 2018*. https://mobilevis.github.io/assets/mobilevis2018_paper_20.pdf.

[32] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum* 33, 3 (2014), 351–360. https://doi.org/10.1111/cgf.12391

[33] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2017). https://doi.org/10.1109/TVCG.2016.2599030

[34] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. In *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis '15)*. https://doi.org/10.1109/TVCG.2015.2467091

[35] Archie Tse. 2011. ai2html. http://ai2html.org/.

[36] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* 3, 32 (2018), 1057. https://doi.org/10.21105/joss.01057

[37] Hadley Wickham. 2010. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28. https://doi.org/10.1198/jcgs.2009.07098

[38] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 649–658.

[39] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting visual analysis with partial view specifications. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (2017), 2648–2659. https://doi.org/10.1145/3025453.3025768

[40] Aoyu Wu, Wai Tong, Tim Dwyer, Bongshin Lee, Petra Isenberg, and Huamin Qu. 2021. MobileVisFixer: Tailoring Web Visualizations for Mobile Phones Leveraging an Explainable Reinforcement Learning Framework. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 464–474. https://doi.org/10.1109/TVCG.2020.3030423

[41] Aoyu Wu, Yun Wang, Xinhuan Shu, Dominik Moritz, Weiwei Cui, Haidong Zhang, Dongmei Zhang, and Huamin Qu. 2021. AI4VIS: Survey on Artificial Intelligence Approaches for Data Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2021), 1–1. https://doi.org/10.1109/TVCG.2021.3099002

[42] Aoyu Wu, Liwenhan Xie, Bongshin Lee, Yun Wang, Weiwei Cui, and Huamin Qu. 2021. *Learning to Automate Chart Layout Configurations Using Crowdsourced Paired Comparison*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445179

[43] Yingcai Wu, Xiaotong Liu, Shixia Liu, and Kwan-Liu Ma. 2012. ViSizer: a visualization resizing framework. *IEEE Trans. Visualization & Comp. Graphics* (2012). https://doi.org/10.1109/TVCG.2012.114

[44] ZingSoft. 2009. ZingChart. https://www.zingchart.com/.