Brunel Documentation

Introduction to Brunel

The Brunel project defines a highly succinct and novel language that defines interactive data visualizations based on tabular data. The language is well suited for both data scientists and more aggressive business users. The system interprets the language and produces visualizations using the user's choice of existing lower-level visualization technologies typically used by application engineers such as RAVE or D3.

The goal of Brunel is to be as easy as possible to integrate into an existing solution. It can be set up as a service (web or local) easily, and delivers results that can directly be added to web pages. The D3 driver, for example, produces javascript and CSS that can are placed directly into web pages with no additional steps needed. In addition the service oriented nature makes other delivery mechanisms simple; Jupyter notebooks (formerly iPython) allow access to Brunel from *Python*, *R* and a variety of other languages. We will be adding more delivery mechanisms as the project progresses -- in general it doesn't take more than a few days.

Language Basics

The Brunel language consists of a list of commands. These commands each control an aspect of the chart. In general the order of the actions is not important, but note:

- Some actions set properties like the coordinate system, but in case of conflict, the last one wins
- Some actions can be applied to multiple targets -- `bin(a,b)` is the same as `bin(a) bin(b)`
- The order can be important for actions that take lists of fields (for example the `x` and `y` commands)

Brunel commands work on a single "element" within a single chart -- in other words they work on a single graphical representation. To combine "elements", we use the composition operators which join together different elements. These operators allow side-by-side charts, overlaid elements and nested elements.

How This Guide is Organized

This document is available both as a simple file, and also as an online guide. The online version allows you to click on the Brunel examples and see them in operation on a sample data set. You can also edit the commands and hit return to modify the syntax.

Some Brunel Examples

```
x(winter) y(summer)

bar x(region) y(#count) sort(#count) + line x(region) y(#count) sort(#count)

bar color(region) y(#count) polar stack label(region)

x(region) y(summer) bin(summer) color(#count) label(#count) style('symbol:rect; border-
radius:15')

bubble label(abbr) size(population) color(region) sort(density) tooltip(state,
population, density)

x(boys_name) y(girls_name) chord size(#count) color(girls_name)

cloud x(state) size(density) color(summer) sort(summer)

bubble size(area) tooltip(state) sort(area) label(state:2) color(region) x(region)

bin(income) x(income) label(#count) tooltip(abbr) y(#count) bar list(abbr)
color(region) stack
axes(x) legends(none)

x(region, presidential_choice) treemap size(population) label(abbr) tooltip(state)
color(dem_rep:red-blue)
```

Basic Concepts

Position Commands: `x` and `y`

The commands `x` and `y` set fields to be used for the x and y dimensions. These dimensions may be physically flipped by using the transpose option, but that does not affect how they behave. Elements _ line*,* bar *and* area_ go along the x axis, and are intended to show how the 'y' field depends on the 'x' field. The Y field is considered as the "result" or "dependent" field and generally is thought of as the more important -- it's a rare chart that doesn't specify 'Y'.

```
x(winter) y(summer)

x(winter)

y(summer)
```

Diagram charts also use the position commands to specify how they are drawn; often they are defined as regular charts with the simple addition of a diagram keyword.

Basic Elements

The language allows you to specify any of the following element types, which define the base way the data is displayed.

point, bar, text, line, area, path, polygon The most important thing to note about the element is whether it is an **aggregating** element. The first three elements (*point*, *bar* and *text*) are not, and they generate one symbol per row of data. The second group of four elements create a single graphic shape from all the data, and so some care is needed when applying **color**, **label** or other aesthetics as must ensure the value of that field is the same for the whole graphic shape. If the aesthetic field is a categorical one, then Brunel handles that for you automatically by splitting up the shape and making one graphic shape per group.

if you do not specify an element, then the `point` element is the default. However a **diagram** knows which element it likes, and so will provide a suitable default. In general it is simplest not to define the element when using a diagram

```
point x(winter) y(summer) color(region) legends(none)

bar x(winter) y(summer) color(region) legends(none)

bar x(winter) y(summer) bin(winter) mean(summer)

text x(winter) y(summer) label(state) color(region) legends(none)

text x(winter) y(summer) label(state:2) color(region) legends(none) tooltip(state)

line x(winter) y(summer) color(region) legends(none)

line x(winter) y(summer) color(region) using(interpolate) legends(none)

area x(winter) y(#count) bin(winter) stack color(region)

path x(winter) y(summer)

polygon x(winter) y(summer)
```

Coordinates

Coordinate-based charts are one of the main two types of charts (diagrams are the other major division). In a coordinates-based chart we assign fields to map chart dimensions (the *X* and *Y* dimensions in a regular chart, or *angle* and *radius* in a polar chart). We can choose to show the axes or not, but that does not affect the existence of the dimensions. The dimensions define a mapping between the data used for one or more fields and an extent on a chart.

Basic Use of `x` and `y`

`x` and `y` work as you would expect when there is only one of each. Note that **Brunel** will choose a suitable scale transform (*linear*, *root* or *log*)) for you unless you explicitly request a different transform:

```
x(density) y(population)

x(density) y(population:linear)

x(density:root) y(population:root)
```

Multiple fields specified for `x` are reserved for future use -- they will allow clustering, or nesting of dimensions within the X axis.

Multiple fields specified for `y` define multiple series; one series for each field specified. This is particularly useful with time series where we have different comparable fields in different columns. When we have multiple `y` fields, the `#series` and `#values` special fields become useful. They allow you to refer to the name of the series and the value of whatever series is being shown without explicitly requiring a name:

```
line x(state) y(summer, winter) sort(summer)

line x(state) y(summer, winter) color(#series) + x(state) y(summer, winter)
color(#series)
tooltip(state, ": ", #series, " = ", #values)
```

`yrange`

The `yrange` command takes two fields and constructs a range between them. It is most often used with bar and area shapes. The statistics `range` and `iqr` are also often used with yrange, as they generate two values (a high and a low), exactly like two fields would.

```
area x(state) yrange(summer, winter) sort(winter)

area x(region) yrange(density) range(density)
```

Coordinate Transforms

- Transpose flips the X and the Y dimensions, so 'Y' reads horizontally and 'X' vertically. Note that this is not the same as simply swapping X and Y as the direction of the elements also changes.
- Polar maps the Y dimension to the polar radius, and the X direction to the polar angle
- Stack takes all items with the same X value and stacks them on top of each other

Currently, polar is poorly supported; stacked polar bars (pie charts) work, but not much else. This will be improved very shortly.

Examples:

```
bar x(region) y(#count)

transpose bar x(region) y(#count)

stack bar x(summer) bin(summer) y(#count) color(region)

stack transpose bar x(summer) bin(summer) y(#count) color(region)

stack polar bar x(summer) bin(summer) y(#count) color(region)
```

Constant and Special Fields

Constant Fields

Any place a field can be used, a constant can be used instead. Constants are defined without quotes as numeric values, or with quotes as a categorical value.

```
bar x(state) yrange(winter, 58.2) + bar x(state) yrange(summer, 58.2)

y(region) x('Hot') color(summer:red) style('symbol:circle') size(#count:100)
mean(summer)
sort(summer) + y(region) x('Cold') color(winter:blue) style('symbol:circle')
size(#count:100) mean
(winter)
```

As well as constant and regular fields, you can use the following "special" fields where a regular field is expected. These fields start with a '#' symbol and have special meaning, as described below. It is often helpful to aggregate rows using the `list` function to show a list of the rows aggregated into a single shape

#row

This field simply has the value of the row of the data, indexed starting at one.

```
text x(#row) y(winter) label(#row) color(#row)

path x(winter) y(summer) + text x(winter) y(summer) label(#row)

text y(region) x(0.5) label('Rows: ',#row) list(#row:20)
```

#count

This field has the value one, initially, but specifying it automatically causes summarization, with the effect that the effective value of the field is the count of the group. This is one of the most common ways of summarization. Using the `percent` summarization command on #count turns it into a percentage.

```
bar x(region) y(#count) sort(#count)

x(winter) y(summer) bin(winter) bin(summer) color(#count) style('symbol:rect')

x(winter) y(summer) bin(winter) bin(summer) color(#count) style('symbol:rect')
percent(#count)
label(#count)
```

#series and #values

These special fields are used when there are multiple 'y' coordinates given. The `#series` field gives the names of each Y field used, and the `#values` field gives the corresponding value of that field.

```
bar x(state) y(summer, winter) stack color(#series) label(#values) tooltip('Value for
', #series, '
is ', #values)
```

#all

This field works for labels and tooltips, and is syntactic sugar for adding all the fields as a list

```
bar x(summer) y(#count) color(region) bin(summer) stack axes(x) percent(#count)
label(#count)
tooltip(#all)
```

Aesthetics

"Aesthetics" is the term we give to commands that modify properties of the graphical object that are not related to position. Currently Brunel supports `color` and `size` aesthetics. Also included in this section is the `split` command, which is really more of a data grouping construct, but it acts in many ways like an aesthetic.

When we use a field as an aesthetic, it has a significant side effect; the data is split into groups, one for each value of that aesthetic. Multiple aesthetics make groups based on the combinations of values. For elements that show one shape for each data row, that makes no difference, but for elements (like area and line) that aggregate multiple rows of data into one shape, it makes a strong difference. For this reason, when using aggregating elements, it is common to use categorical fields for aesthetics. Alternatively, we can bin a numeric field to ensure it has fewer different values.

Color

Color is probably the most used aesthetic. When we map a field to

```
color(winter) x(winter) y(summer) style("size:200%")

color(region) x(winter) y(summer) style("size:200%")

color(winter) x(winter) y(summer) bin(winter) style("size:200%")
```

In the above examples, you can see that the mapping from field to color is dependent on the type of the field. There are three different classes of mapping used:

- **nominal** : Used by default for categorical data, this mapping has no order and tries to create a spread of different hues to distinguish as many different hues as possible
- **diverging** : Used by default for most numeric data, this mapping assumes that high and low values are of interest and so creates a two-ended color range that highlights those values
- **continuous** : Used for counts and binned data, this mapping goes from low to high with one range of color

Some examples are given below:

```
color(winter:nominal) bar x(winter) y(#count) bin(winter)

color(winter:diverging) bar x(winter) y(#count) bin(winter)

color(winter:continuous) bar x(winter) y(#count) bin(winter)
```

In addition to these classes of mappings, explicit color mappings can be used from the following list: `blue`, `green`, `red`, `yellow`, `gray`, `blue-red`, `green-red`, `green-blue`, `white-black` , `red-blue`, `red-green`, `blue-green`, `black-white` :

```
color(winter:gray) x(winter) y(summer) style("size:200%")

color(winter:blue-red) x(winter) y(summer) style("size:200%")

color(winter:green) x(winter) y(summer) style("size:200%")
```

As described above, aesthetics interact with the element type. Because elements such as lines need one color only, when we use color on such an element, it splits into pieces, one for each "group" created by the aesthetic.

```
line x(winter) y(summer) color(region)

polygon x(winter) y(summer) color(region)
```

Currently, only the first color field is used. In the future, we will enhance this so that if two fields are set, the first is used for HUE, and the second for SATURATION and BRIGHTNESS. When three fields are used, they will set the red, green and blue components.

```
color(region, #count) x(summer) size(#count) bin(summer) bubble
```

Size

The size aesthetic works very similarly to color, except it modifies the size of the element. It can be given an optional parameter which is a percentage to increase size by for the maximum value of the field. This defaults to 100.

```
point x(state) y(density) size(density)

point x(state) y(density) size(density) style("size:200%")

point x(state) y(density) size(density:1000)
```

`Size` works differently for different elements; point elements can be sized as you would expect; bars are sized on their widths only (so `width` and `size` have the same effect. Lines, paths and edges have their stroke sizes modified. Size has no effect on areas and polygons.

```
point x(state) y(density) size(density)

bar x(region) y(density) size(#count) mean(density) label(#count)

line x(state) y(density) size(region)

line x(state) y(density) size(region:1600)
```

Size can also take two fields. For this definition the two fields modify width and height, so this way of specifying size is most suited to a point element with a rectangle type.

```
x(longitude) y(latitude) size(population, density) style('symbol:rect; size:300%')
```

Split

The split aesthetic does not modify the appearance of items at all -- all it does is to split up a single item (like an area)into multiple ones. Effectively it is used just for creating groups. Multiple fields can be used to split by

```
polygon x(winter) y(summer) split(region)
```

Data Transformations

In Brunel, we define data transforms on fields, and the system coordinates all of these into a final set of transformations. An important point is that a transformation completely replaces a field. This means that if you bin a field, for example, you no longer have access to the unbinned values. In practice this limitation is not often a difficulty as when we combine visualizations, we can use different transforms *within* each visualization.

Sort

The sort action can be applied to any list of fields, and has the result that when a categorical field is being used in the data set, then we set the order of that field's categories so that the ones corresponding to high values of the sort fields are shown first. We can set an optional parameter 'ascending' to change to show smallest values first. When multiple fields are used in the sort, the first field is the most important -- the others are used only to break ties.

Here are some sort examples:

```
x(state) y(summer) sort(summer) color(region) legends(none)

x(state) y(summer) sort(region) color(region) legends(none)

x(state) y(summer) sort(region,summer) color(region) legends(none)

x(state) y(summer) sort(summer:ascending) color(region) legends(none)
```

Bin

For numeric data, the bin action takes a set of numeric values and transforms them into an ordered set of categories representing ranges of the data. This is done adaptively, so the bins will be different for different data sets. Binning for dates is done based on calendar ranges and so bins for dates may not be of equal numbers of days (for example when we bin by months)

For categorical data, the bin actions bins all categories with small counts into a single "Other" category. "Small" is defined by default that the non-binned data will comprise about 95% of the total data (i.e. we try and aggregate the lowest 5%). Note that this means that if there are lot of very small count values the "Other" category will be large.

Binning does not automatically aggregate or summarize the data. There will still be the same number of data rows after binning. Bin has an optional parameter which is the desired number of bins.

```
x(summer) y(winter) bin(summer)

x(summer) y(winter) bin(summer:3)

x(summer) y(winter) bin(summer, winter) style("opacity:0.1")

x(summer) y(winter) bin(summer:10, winter:10) style("opacity:0.1")

x(summer) y(winter) bin(summer, winter) size(#count)
```

Aggregation

In Brunel, data can of course be passed in pre-aggregated (and this is necessary for very large data sets), but to get fast local interactivity, we need to be able to aggregate and re-aggregate in the client. We provide a simple system for aggregation, with the following features:

- Aggregation is performed when a summary function is defined (see list below) or the special field `#count` is used.
- When aggregating, #count and any fields defined by a summary function are treated as responses, and all other fields are used to define the groups or 'dimensions' for the summaries.
- All other fields are dropped

The following summary functions work for all types of field (categorical and numeric)

- **count** : The number of rows in the group
- **valid** : The number of rows that are not missing and (if numeric or date) of the correct format
- **unique** : The number if unique categories or values in that group
- **list** : A concatenated list of the unique values for the group. Takes an optional integer parameter that limits the number of items to display (this defaults to 12)
- **mode** : The most common value (ties broken by the row order)
- **mean** : The mean value. Note that for categorical data, this silently changes to the mode. Although this is unusual, it is very helpful for use when you are unsure if data is numeric or not

```
bar x(region) y(#count) label(#count)

bar x(region) y(population) count(population) sort(population) label(region:3, ": ", population)

bar x(region) y(population) valid(population) sort(population) label(region:3, ": ", population)

bar x(summer) bin(summer) y(region) unique(region) label(region)

bar y(1) bin(summer) color(summer:red) list(region) label(region) stack axes(none) legends(none)

bubble label(region:8) list(region) size(summer) bin(summer:20) tooltip(region)

bar x(summer) bin(summer) y(#count) mode(region) label(region:9)
```

The following summary functions produce results only for numeric data

- **sum** : Sum of all values in the group
- **percent** : The percent of the sum of this group as a percent of the sum of all groups with the same 'x' value
- **median** : Median value of the group
- **min**, **max** : Lower and upper values of the group
- **range** : Distance between min and max values
- **q1**, **q3** : Lower and Upper quartiles
- **iqr** :Distance between q1 and q3 -- the interquartile range
- **stddev**, **variance**, **skew**, **kurtosis** : statistical measure for the group

Note that `iqr` and `range` produce a range -- two values. If used with 'y' the result is the distance between them, but if used with 'yrange' it will generate the actual range. See the examples below for how this works

```
bar x(region) y(population) sum(population) sort(population)

bar x(summer) bin(summer) color(region) y(#count) percent(#count) stack label(#count)

area x(region) y(density) mean(density) sort(density)

bar x(region) y(summer) range(summer) sort(summer)

bar x(region) yrange(summer) range(summer) sort(summer)

area x(region) yrange(dem_rep) iqr(dem_rep) + line x(region) y(dem_rep) median(dem_rep)
```

Interactivity

Brunel allows a number of interactive features. If not specified, some interactivity is available by default, but, as always, specifying what is wanted will force that option. Note that interactivity is specified on a per-chart basis, so each chart can have a different mode of use.

Pan/Zoom

By default this is set **on**, but this feature will only apply to *numeric* dimensions of charts that do not specify a diagram. When this feature is active, a chart can be panned and zoomed by dragging or double-clicking on any **blank** area of the chart. Holding down the shift key while double-clicking zooms the chart out instead of in.

```
text x(winter) y(summer) tooltip(state, region) label(abbr)

x(latitude) y(summer) | x(latitude) y(winter) interaction(none)
```

Selection

Each data set has a special field `"#selection"` that can be used in the same way as any other field -- for color, coordinates, etc. In general this feature not be useful unless you have multiple charts and at least one of them states that they use selection interactivity. When clicking on elements of that chart, those selections will then be propagated through to the other charts in that system.

Selection takes on two possible values, an 'x' for unselected and a check mark for selected. One common use case is to map the value to color to show the selected parts from one chart as highlighted in the other chart.

```
x(winter) y(summer) color(#selection) size(#selection:200) | y(region) size(#count)
interaction(select) color(#selection) x('') axes(none) label(region)
sort(#count:ascending)

treemap x(region, presidential_choice) tooltip(#all) size(population) color(#selection)
label(abbr)
| bar x(boys_name) y(#count) stack color(#selection) interaction(select) transpose
axes(x) | bar
x(girls_name) axes(x) y(#count) stack color(#selection) interaction(select)
color(#selection)
transpose axes(x) legends(none) stack
```

Interaction(filter)

Instead of using the `#selection` field as a regular field, it can also be used to filter the data going into the chart, by specifying `interaction(filter)`.

```
x(boys_name) y(girls_name) label(#count) interaction(filter) | bar y(population) stack
color(#selection) split(region) sort(population) label(region) sum(population)
interaction(select)
```

Filter

The `filter` command is an interaction command which will create one or more interactive controls (sliders, check boxes,...) beside the visualization and allow dynamic filtering using those items. As is true for all interactivity, the filtering happens client-side and sorting and summarization happen afterwards and so will respect the filtered data.

```
x(population) y(violent_crimes) color(dem_rep) size(water:600)
filter(presidential_choice, water)
```

Diagrams

Diagrams take the existing X and Y values and re-interpret them using a layout as a diagram, rather than by using the positional values in a coordinate system.

Treemap

This uses all the position fields to create a recursively divided space where each field splits the previous set of rectangles up into smaller ones so as to fill the space completely

```
treemap x(region) label(abbr)

treemap x(region) size(population) sum(population) label(state) list(state)
sort(population)
color(population:green)

x(region,summer) bin(summer) treemap label(abbr) list(state) size(#count)
color(summer:red)
legends(none) sort(summer)
```

Cloud

This ignores all positions and places the rows in a tag-cloud layout. If a label is defiend, it uses that text, otherwise it uses the rows

```
cloud color(population) size(population)

cloud label(abbr) color(population) size(population)

cloud label(abbr,":",region:3) color(population) size(population)
```

Chord

This uses the first two positional fields as categories in a chord plot

```
x(boys_name) y(girls_name) chord color(girls_name) legends(none)
```

Bubble

Like cloud, 'bubble' ignores the positions and simply places all items together as if they were bubbles. Unlike cloud, bubble uses multiple fields to form a hiearachy, like treemap does

```
bubble size(population) color(region) label(state) legends(none)

bubble x(region) size(population) color(region) label(abbr) legends(none)

bubble x(region, presidential_choice) size(population) color(region) label(abbr)
legends(none)
tooltip(#all)
```

Label and Tooltip

Labels and tooltips are text that is associated with each graphical item. They have the same command format, but the outputs are different. Labels appear all the time on the display and move with the element in an animated display. Tooltips only appear when the item is hovered over with the mouse (or the equivalent gesture on a touch interface), and are transitory.

Care must be taken when using labels and tooltips with elements like line, path and area that aggregated data, as any fields used to label should be consistent within each group. In general this means that fields used for `color` and `split` make good choices. Otherwise you may end up with extra groups created to ensure group labels are consistent.

```
y(population:log) label(state)

area x(population) y(density) bin(population) sum(density) stack color(region)
label(region)
axes(none)
```

When specifying labels and tooltips, the parameters are a mixture of fields and string literals. These are all concatenated together to produce the final string. Specifying the `#all` field shows every field used in the rest of the chart If multiple fields, and only fields, are present in the definition, that triggers a special formatting mode.

- `Labels` show all the fields, separated by commas
- `Tooltips` show the field names and the field values, each on its own line

```
bubble size(#count) color(region) label(region) tooltip(region)

bubble size(#count) color(region) label(#all) tooltip(#all)

bubble size(#count) color(region) label(region, "(", #count, ")") tooltip(#all)

bubble size(#count) color(region) label(region, violent_crimes, income) tooltip(#all)
mean(violent_crimes, income)
```

Each label field can have an optional integer parameter that makes a "shortened" form of the field. This can be useful to provide a rough idea what the name is, with a tooltip giving the full name. If you are using the `list` summation method, you can get a variety of labeling effects by manipulating both the number of list items produced and the number of characters to show in the label.

```
treemap label(state:3) tooltip(#all) color(violent_crimes) size(population)
sort(population:ascending)

treemap size(population) color(region) label(state:50) list(state) sum(population)
tooltip(state)

treemap size(population) color(region) label(state:40) list(state:5) sum(population)
tooltip(state)
```

Axes and Style

Axes

The axes command controls which axes are displayed. Legal values are `none, all, x, y, auto`

```
bar x(region) y(#count) axes(none)

bar x(region) y(#count) axes(all)

bar x(region) y(#count) axes(x)

bar x(region) y(#count) axes(y)
```

Legends

The legends command controls which axes are displayed. Legal values are `none, all, auto`

```
color(winter) size(summer) x(state) y(summer) legends(auto)

color(winter) size(summer) x(state) y(summer) legends(all)

color(winter) size(summer) x(state) y(summer) legends(none)
```

Style

This command is used to change the style of the chart. Its parameter can be a very long string, and consists of CSS-like descriptions of styles. If there are no tags that indicate what the style applies to, it defaults to the element being show. The possible styles include `fill, outline, symbol,width, height, size, font-family, font-size, font-weight, stroke-width` .

```
x(region) y(#count) style('fill:red')

x(region) y(#count) style('fill:white; outline:red')

x(region) y(#count) style('size:400%')

x(region) y(#count) style('symbol:rect')

x(region) y(#count) style('symbol:rect;border-radius:5')

x(region) y(#count) style('size:30;stroke-width:3')

treemap color(region) size(#count) label(#all) style('.label {font-size:24px}')

bar x(region) y(#count) label(#all) style('.label {font-weight:bold; fill:white;
text-shadow:none})')

x(region) y(#count) style('.axis {fill:red; font-weight: bold')

x(region) y(#count) style('.axis.tick line {stroke-width:5;stroke:red}.axis.title
{font-size:30px
;fill:cyan}')
```

When there are multiple elements in a chart, if you use the simple form of style without a target, it will choose the current element only as the target.

```
line x(state) y(summer) style('stroke:red') + line x(state) y(winter)
style('stroke:blue')
```

Style Details

The styles defined are generally CSS style statements, and are placed into the resulting code as a local style sheet. However note:

- Brunel reads the styles and uses the values for initial layout.; for example it may reserve space depending on font size. This expands the list of possible style attributes, notably adding `symbol` and `size` as attributes that can be specified.
- However, Brunel does not support all the complexities of CSS. If you stick to element names and paths and combining them with commas and hierarchies, you should be fine.
- Local overrides (with the style command) are given them CSS "!important" tag. This means they will completely override any other definition. So if you do `style('* {fill:blue}')` it will turn all of brunel blue...
- SVG CSS is used, so we do not use "color" for color -- instead use "fill" or "stroke"

```
line x(state) y(summer) style('stroke:red') + line x(state) y(winter)
style('stroke:blue')
```

Group Hierarchy for CSS

The hierarchy for CSS purposes is given below. This is not written in stone and may change slightly in the future. Be careful of depending on it too strongly

```
  svg.brunel#id             -- the top level svg container with an id as given by the
application
    g.chart1                -- the first chart
      g.interior            -- items inside the coordinate space

        g.element1          -- first element
          g.main            -- main items in the element
            ???.element     -- this is where the lines, paths, rects and so on will be
found
          g.labels          -- labels for the shapes in '.main'
            text.label      -- all the labels

        g.element2          -- second element
          ...               -- main, labels and any other special items (diagrams
sometimes add a group)

      g.axis                -- axes for the first chart
        g.xaxis             -- x axis
          text.title        -- axis title
          g.tick            -- group for a tick mark (many of these)
            line            -- tick mark line
            text            -- tick mark label
        g.yaxis             -- y axis
          ...

      g.legend              -- axes for the first chart
        text.title          -- legend title
        g.tick              -- group for a legend swatch (many of these)
            rect            -- swatch item
            text            -- swatch label
```

Chart Composition

Chart Composition actions are those that take two charts and combine them to make a single one. The model for how they work is:

(actions defining chart A) + (actions defining chart B)

In the grammar, though, the parentheses are not required, or even allowed. The three composition methods are `tile`, `overlay` and `nest`

Composition is a work in progress. Currently `tile` and `overlay` work, but cannot be combined. `nest` does not work at all. When completed, and all work well, the precedence of the combinations will be `nest` binds tightest, and `tile` weakest. Thus `A | B + C < D | E` will result in three charts: "A", "B + C < D" and "E". The middle chart will have one element "B" and the second element "C" will have element "D" nested within it.

`|` (tile)

This is the simplest form of composition. It tiles the available space and puts the charts into the space. The default layout can be changed by giving each chart an `at` action to indicate its location (in percentages: left, top, right, bottom)

Examples:

```
bar x(region) y(income) mean(income) | bar x(region) y(violent_crimes)
mean(violent_crimes)
x(density) y(income) at(0,0,100,100) color(region) | polar stack bar y(#count)
color(region) at
(60,45,100, 90) legends(none)
```

`+` (overlay)

This method of composition attempts to combine the coordinate systems of the charts, placing one on top of the other and having them share axes. Because both charts contribute to the same coordinate space, it is important to ensure that those coordinates are compatible. Below are some examples showing how this works

Examples:

```
bar x(region) y(water) mean(water) + line x(region) y(under_18) mean(under_18) label("%
under 18")

bar x(region) yrange(income) range(income) + bar x(region) yrange(income) iqr(income) +
point
x(region) y(income) median(income) style("fill:white")
```

`>` (nest)

This method of composition places one chart inside the other. To do this the two charts must have a hierarchical nature -- the first chart should represent an aggregation of the second, so the nesting makes sense. When that is defined, the second chart will be replicated as small multiples within the other chart.