Supporting Data Journalism through Compilers for Visual Inputs

Parker Ziegler // @parker_ziegler
Ph.D. Student, UC Berkeley

Strange Loop
St. Louis, MO • September 21, 2023
What does journalism have to do with compilation?

Aren't compilers for transforming programs, not charts?

What does this even mean?
Fixed Entities

Live Objects

Compilers
Data Journalism
The coronavirus arrived in the United States by early 2020, setting off wave after wave of infection and death in the months that followed.
Will global warming make temperature less deadly?

Harry Stevens
Data Journalism

“Telling stories with data”
Data Journalism

(Lots and lots of)

Programming
Data Journalism

Municipal Agencies

FOIA Requests

Police Departments

Academic Journals

Clean

Normalize

Filter

Transform
Data Journalism

Computational Notebooks
GIS Software
Web Frontends
In 2018, police in the city of Plainfield, N.J., started using software called PredPol to predict where crime would likely happen.

All 53 students are scattered; 50 of them have lost their homes.
Data Journalism

Programming
How can we design and build **programming tools** to support the working practices of **data journalists**?
How can we design and build programming tools to support the working practices of data journalists?
30 hours of video observation with Data Journalists, Social Scientists, and Earth and Climate Scientists.
A Need-Finding Study with Users of Geospatial Data

Parker Ziegler
penzieger@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Sarah E. Chains
schains@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

ABSTRACT

Geospatial data is playing an increasingly critical role in the work of Earth and climate scientists, social scientists, and data journalists exploring spatiotemporal change in our environment and societies. However, existing software and programming tools for geospatial analysis and visualization are challenging to learn and difficult to use. The aim of this work is to identify the unmet computing needs of the diverse and expanding community of geospatial data users. We conducted a contextual inquiry study (n = 25) with domain experts using geospatial data in their current work. Through a thematic analysis, we found that participants struggled to (i) find and transform geospatial data to satisfy spatiotemporal constraints, (2) understand the behavior of geospatial operators, (3) track geospatial data provenance, and (4) explore the cartographic design space. These findings suggest design opportunities for developers and designers of geospatial analysis and visualization systems.

CCS CONCEPTS

• Human-centered computing → Human computer interaction (HCI); Empirical studies in HCI; Interactive systems and tools.

KEYWORDS

geospatial data, GIS, geography, cartography, context, need-finding

ACM Reference Format:


1 INTRODUCTION

Geospatial data—data encoding the location and attributes of phenomena on the Earth’s surface [5]—is growing in scale and diversity at a tremendous rate [4]. Researchers estimate that geospatial satellites generate ~10PB of new imagery daily [8] to support the surface, cheap, power-efficient sensors that are used to collect environmental data. This explosion of geospatial data measuring real-time environmental conditions has created new challenges for researchers and practitioners to make sense of this data. Moreover, many of these data sources are proprietary and expensive to access. For example, using OpenStreetMap (OSM), a collaborative effort to create a free and editable map of the world, would cost between $250,000 and $10 million to produce [43]. Geospatial data has long played a pivotal role in the research of geographers and cartographers. However, recent advances in the community of geospatial data science have made it more available, and geospatial data science is now an important tool for understanding the world.
Lift visual styles and graphical forms from examples

Apply to new datasets
Lift visual styles and graphical forms from examples

Reverse engineering was as time-consuming as developing the visualization from scratch.

Data Journalists
Social Scientists
Earth and Climate Scientists
Data Journalists

Social Scientists

Earth and Climate Scientists

Sketch visualizations in **design software**

Explore a wide design space **without code**
After selecting a design, they still had to reproduce the visualization in code.
```javascript
const x = d3.scaleLinear()
  .domain(d3.extent(cars, d => d['weight (lb)']()))
  .range([marginLeft, width - marginRight]);

const svg = d3.create('svg')
  .attr('width', width)
  .attr('height', height)
  .attr('viewBox', [0, 0, width, height])
  .attr('style', 'max-width: 100%; height: auto;');

svg.append('g')
  .attr('transform', `translate(0,${height - marginBottom})`)
  .call(d3.axisBottom(x).tickSizeOuter(0));

svg.append('g')
  .selectAll()
  .data(dodge(cars, {radius: radius * 2 + padding}))
  .join('circle')
  .attr('cx', d => d.x)
  .attr('cy', d => height - marginBottom - radius - padding - d.y)
  .attr('r', radius)
  .append('title')
  .text(d => d.data.name);
```
Could this be a *compilers* problem?
Could this be a compilers problem?
PL RESEARCHERS

LITERALLY ANYTHING

IS THIS A COMPILERS PROBLEM?
How do we compile from a visual form to a textual symbolic representation?
How do we compile from a **visual form** to a **textual symbolic representation**?

```assembly
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32) (result i32)
    get_local 1
    get_local 0
    i32.add)
  (table (;0;) 1 1 anyfunc)
  (memory (;0;) 17)
  (global (;0;) i32 (i32.const 1049118))
  (global (;1;) i32 (i32.const 1049118))
  (export "memory" (memory 0))
  (export "__indirect_function_table" (table 0))
  (export "__heap_base" (global 0))
  (export "__data_end" (global 1))
  (export "add" (func 0))
  (data (i32.const 1049096) "invalid malloc request"))
```
How do we compile from a **visual form** to a **textual symbolic representation**?

Recover (or infer) **semantic information** from the visual form that we can represent **symbolically**.
How do we compile from a **visual form** to a **textual symbolic representation**?

Map **interactions** with the visual form to **program edits**.

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32) (result i32)
    get_local 1
    get_local 0
    i32.add)
  (table (;0;) 1 1 anyfunc)
  (memory (;0;) 17)
  (global (;0;) i32 (i32.const 1049118))
  (global (;1;) i32 (i32.const 1049118))
  (export "memory" (memory 0))
  (export "__indirect_function_table" (table 0))
  (export "__heap_base" (global 0))
  (export "__data_end" (global 1))
  (export "add" (func 0))
  (data (i32.const 1049096) "invalid malloc request")
)```
Map interactions with the visual form to program edits.

Recover (or infer) semantic information from the visual form that we can represent symbolically.

Lexer
Parser

reviz
cartokit
A compiler from **SVG subtrees** to **data visualizations**
A compiler from SVG subtrees to data visualizations, partial programs
Demo

The New York Times

Least Vaccinated U.S. Counties Have Something in Common: Trump Voters
Danielle Ivory, Lauren Leatherby and Robert Gebeloff

Can we automatically retarget existing visualizations at new datasets?
A user identifies a visualization they want to use as a visual input.

The New York Times

Least Vaccinated U.S. Counties Have Something in Common: Trump Voters

Danielle Ivory, Lauren Leatherby and Robert Gebeloff
They pass its svg subtree to
reviz compiles a **partial JavaScript program** using Observable’s Plot library.

```javascript
const plot = Plot.plot({
  color: {
    type: "categorical",
    range: ["#67371f", "#96cb02", "#e67e22", "#e67e22", "#e67e22"]
  },
  marks: [
    Plot.dot(data, {
      x: "??",
      y: "??",
      fill: "??",
      fillOpacity: 1,
      stroke: "??",
      strokeOpacity: 1,
      strokeWidth: 1
    }],
  ]
});
```
A user **fills in the holes** ("??") in the partial program to produce a new visualization.
What actually happens in this step?
It all starts with a walk.

Visit each node in the svg subtree.
It all starts with a **walk**.

```
<svg viewBox="0 0 493.5 450" width="493.5" height="450">
  <g aria-label="dot">
    <circle cx="366" cy="349" r="7" fill="#C67371" fill-opacity=".8" stroke="#C67371" />
    <circle cx="140" cy="167" r="7" fill="#A789D3" fill-opacity=".8" stroke="#A789D3" />
    <circle cx="121" cy="119" r="7" fill="#709DDE" fill-opacity=".8" stroke="#709DDE" />
  </g>
  <g aria-label="x-axis tick label" fill="none" stroke="currentColor">
    <text y="0.71em" transform="translate(56, 420)" stroke="currentColor">10</text>
    <text y="0.71em" transform="translate(56, 420)" stroke="currentColor">20</text>
  </g>
</svg>
```
It all starts with a **walk**.
Read **geometric** and **presentational** attributes off of each DOM node and its computed styles.

```
<circle cx="366" cy="349" r="7" fill="#C67371" fill-opacity=".8" stroke="#C67371" />
```

**Styles**

<table>
<thead>
<tr>
<th>Style</th>
<th>Computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>stroke-width</td>
<td>2px</td>
</tr>
<tr>
<td>stroke-opacity</td>
<td>1</td>
</tr>
</tbody>
</table>

**Computed**

- `cx` → 366
- `cy` → 349
- `r` → 7
- `fill` → "#C67371"
- `fill-opacity` → 0.8
- `stroke` → "#C67371"
- `stroke-opacity` → 1
- `stroke-width` → 2px
Store collected attributes in **attribute sets**.

\[
\begin{align*}
\text{cx:} & \{ \text{cx → 366}, \text{cx → 398}, \text{cx → 422}, \text{cx → 543}, \ldots \} \\
\text{cy:} & \{ \text{cy → 349}, \text{cy → 218}, \text{cy → 265}, \text{cy → 121}, \ldots \} \\
\text{r:} & \{ \text{r → 7} \} \\
\text{fill:} & \{ \text{fill → “#C67371”}, \text{fill → “#A7B9D3”}, \text{fill → “#709DDE”}, \ldots \} \\
\text{fill-opacity:} & \{ \text{fill-opacity → 0.8} \} \\
\text{stroke:} & \{ \text{stroke → “#C67371”}, \text{stroke → “#A7B9D3”}, \text{stroke → “#709DDE”}, \ldots \} \\
\text{stroke-opacity:} & \{ \text{stroke-opacity → 1} \} \\
\text{stroke-width:} & \{ \text{stroke-width → 2px} \}
\end{align*}
\]
Apply a set of **predicate functions** associated with a **visualization type**.

\[
\begin{align*}
\text{cx}: & \{ \text{cx} \rightarrow 366, \ldots \} \\
\text{cy}: & \{ \text{cy} \rightarrow 349, \ldots \} \\
\text{r}: & \{ \text{r} \rightarrow 7 \} \\
\text{fill}: & \{ \text{fill} \rightarrow \text{"#C67371"}, \ldots \} \\
\text{fill-opacity}: & \{ \text{fill-opacity} \rightarrow 0.8 \} \\
\text{stroke}: & \{ \text{stroke} \rightarrow \text{"#C67371"}, \ldots \} \\
\text{stroke-opacity}: & \{ \text{stroke-opacity} \rightarrow 1 \} \\
\text{stroke-width}: & \{ \text{stroke-width} \rightarrow 2\text{px} \}
\end{align*}
\]
Compute the **ratio of predicates** returning **true** for each **visualization type**.

**Bar Chart** (0/3 predicates)
- `hasMarkType("rect")`
- `hasConsistentGeomAttr("width")`
- `hasXScaleType("width")`

**Scatterplot** (2/2 predicates)
- `hasMarkType("circle")`
- `hasConsistentGeomAttr("r")`

**Strip Plot** (2/3 predicates)
- `hasMarkType("circle")`
- `hasConsistentGeomAttr("r")`
- `hasSiblingsWithConsistentCyAttr`

**Bubble Chart** (1/2 predicates)
- `hasMarkType("circle")`
- `hasDivergentGeomAttr("r")`
Merge the inferred **visualization type** with the **attribute sets** to produce the **intermediate representation** (IR).

**Scatterplot** (2/2 predicates)

```json
{
  type: "Scatterplot",
  r: [7],
  fill: ['#C67371', '#ccc', '#709DDE'],
  fill-opacity: [0.8],
  stroke: ['#C67371', '#ccc', '#709DDE'],
  stroke-opacity: [1],
  stroke-width: ['2px']
}
```
The **frontend** of the **reviz** compiler

---

**Input Visualization**

**Intermediate Representation**

```json
{
  type: "Scatterplot",
  r: [7],
  fill: ["#C67371", "#ccc", ","709DDE"],
  fill-opacity: [0.8],
  stroke: ["#C67371", "#ccc", ","709DDE"],
  stroke-opacity: [1],
  stroke-width: ["2px"]
}
```
The **backend** of the `reviz` compiler

```
const plot = Plot.plot({
  color: {
    type: "categorical",
    range: ["#C67371", "#ccc", "#709DDE"]
  },
  marks: [
    Plot.dot(data, {
      x: "??",
      y: "??",
      fill: "??",
      fillOpacity: 1,
      stroke: "??",
      strokeOpacity: 1,
      strokeWidth: 1
    })
  ]
});
```

Intermediate Representation

Partial Program
Start with the IR and an empty program, signified by an **evaluation hole**.

```json
{
  type: "Scatterplot",
  r: [7],
  fill: ['#C67371', '#ccc', '#709DDE'],
  fill-opacity: [0.8],
  stroke: ['#C67371', '#ccc', '#709DDE'],
  stroke-opacity: [1],
  stroke-width: ['2px']
}
```

**Intermediate Representation**  **Partial Program**
Apply a set of **rewrite rules** to transform **key-value pairs** in the IR into **program fragments**.

```
`Plot.dot(data, {
  x: "\${PROGRAM_HOLE}",
  y: "\${PROGRAM_HOLE}",
  \${EVAL_HOLE}
});`
```

**Intermediate Representation**

```
{
  type: "Scatterplot",
  r: [7],
  fill: ["#C67371", "#ccc", "#709DDE"],
  fill-opacity: [0.8],
  stroke: ["#C67371", "#ccc", "#709DDE"],
  stroke-opacity: [1],
  stroke-width: ["2px"]
}
```

**Partial Program**
Apply a set of **rewrite rules** to transform **key-value pairs** in the IR into **program fragments**.

```
{
  type: "Scatterplot",
  r: [7],
  fill: ["#C67371", "#ccc", "#709DDE"],
  fill-opacity: [0.8],
  stroke: ["#C67371", "#ccc", "#709DDE"],
  stroke-opacity: [1],
  stroke-width: ["2px"]
}
```

```
`Plot.dot(data, {
  x: "${PROGRAM_HOLE}",
  y: "${PROGRAM_HOLE}",
  r: "7",
  ${EVAL_HOLE}
})`
```

```
`Plot.dot(data, {
  x: "${PROGRAM_HOLE}",
  y: "${PROGRAM_HOLE}"
})`
```
Continue applying rewrites until we’ve read all key-value pairs in the IR.
Continue applying rewrites until we’ve read all key-value pairs in the IR.

```
{
  type: "Scatterplot",
  r: [7],
  fill: ["#C67371", "#ccc", "#709DDE"],
  fill-opacity: [0.8],
  stroke: ["#C67371", "#ccc", "#709DDE"],
  stroke-opacity: [1],
  stroke-width: ["2px"]
}
```

Intermediate Representation

```
'Plot.dot(data, {
  x: "${PROGRAM_HOLE}",
  y: "${PROGRAM_HOLE}",
  r: 7,
  fill: "${PROGRAM_HOLE}",
  fillOpacity: 0.8,
  stroke: "${PROGRAM_HOLE}",
  strokeOpacity: 1,
  strokeWidth: "2px"
})`

Partial Program
evalGeomAttr

evalPresAttr

evalType

evalColor

evalR

Input Visualization

Attribute Sets

Intermediate Representation

Rewrite Rules and Codegen

Partial Program

Output Visualization
Working Practice

Lift visual styles and graphical forms from examples

Apply to new datasets
Lift visual styles and graphical forms from examples.

Reverse engineering was as time-consuming as developing the visualization **from scratch**.
Working Practice

Lift visual styles and graphical forms from examples

Automaticaly compile partial programs from examples

Hours

Apply to new datasets

Feed in any dataset

Milliseconds
Don’t commit to a specific visual form before seeing your data in it. A given visual form — say the pie chart or treemap — isn’t “good” or “bad” in an absolute sense, but it may or may not be appropriate to your data and the specific question you want answered. **The only way to know whether a form is effective is if it communicates: you must put your data in it and see.**

Mike Bostock • “10 Years of Open-Source Visualization”
Recover (or infer) **semantic information** from the visual form that we can represent **symbolically**.

Map **interactions** with the visual form to **program edits**.
cartokit

A *direct manipulation* programming environment for *interactive cartography*
Can we author programs through **direct manipulation** of their outputs?
A user uploads their geospatial data to **cartokit**.
cartokit renders and displays the data while simultaneously generating a program.
A user styles the map via **direct manipulation** while **cartokit** recompiles a matching program.
A user can **copy**, **modify**, and **deploy** the compiled program as desired.
How do we get from interactions to program edits?
Map property controls in the UI to an intermediate representation.

```javascript
const ir = {
  center: [-106.1086, 37.7531],
  zoom: 4,
  basemap: {
    url: "https://tiles.stadimaps.com/styl...",
    provider: "Stadia Maps",
  },
  layers: [
    {
      type: "Fill",
      style: {
        fill: {
          color: "#8638e5",
          opacity: 0.75,
        },
        stroke: {
          color: "#ffffff",
          width: 1,
          opacity: 0.5,
        },
      },
    },
  ],
};
```
When a user alters a property in the UI, dispatch an update to the IR.
When a user alters a property in the UI, **dispatch an update** to the IR.

```javascript
const ir = {
  center: [-106.1086, 37.7531],
  zoom: 4,
  basemap: {
    url: "https://tiles.stadiamaps.com/styl...
  },
  layers: [
    {
      type: "Choropleth",
      style: {
        fill: {
          attribute: "years_2020_2039",
          scale: "Quantile",
          count: 8,
          scheme: d3.schemeBuPu,
          thresholds: [-210.21, -23.03, ...],
          opacity: 0.75,
        },
        stroke: {
          color: "#ffffff",
          width: 1,
          opacity: 0.5,
        },
      },
    },
  ]
};
```
Begin **code generation**, starting with dependencies.

```javascript
const ir = {
  center: [-106.1086, 37.7531],
  zoom: 4,
  basemap: {
    url: "https://tiles.stadiamaps.com/styl…",
    provider: "Stadia Maps",
  },
  layers: [
    {
      type: "Choropleth",
      style: {
        fill: {
          attribute: "years_2020_2039",
          scale: "Quantile",
          count: 8,
          scheme: d3.schemeBuPu,
          thresholds: [-210.21, -23.03, ...],
          opacity: 0.75,
        },
        stroke: {
          color: "#ffffff",
          width: 1,
          opacity: 0.5,
        },
      },
    },
  ];
```

```javascript
`import mapboxgl from "mapbox-gl";
${transformTable.size > 0
  ? 'import * as turf from "@turf/turf";
    : ""}
${codegenFileImports(ir)}
${codegenFns(ir, transformTable)}
${codegenMap({ map, ir, uploadTable, transformTable })}`
```
Pass execution to **codegen functions** to produce **program fragments** for distinct parts of the IR.

```javascript
const ir = {
  center: [-106.1086, 37.7531],
  zoom: 4,
  basemap: {
    url: "https://tiles.stadiamaps.com/styles/",
    provider: "Stadia Maps",
  },
  layers: [
    {
      type: "Choropleth",
      style: {
        fill: {
          attribute: "years_2020_2039",
          scale: "Quantile",
          count: 8,
          scheme: d3.schemeBuPu,
          thresholds: [-210.21, -23.03, ...],
          opacity: 0.75,
        },
        stroke: {
          color: "#ffffff",
          width: 1,
          opacity: 0.5,
        },
      },
    },
  ],
};

`import mapboxgl from "mapbox-gl";
${transformTable.size > 0
  ? 'import * as turf from "@turf/turf";
  : ""}
${codegenFileImports(ir)}
${codegenFns(ir, transformTable)}
${codegenMap({ map, ir, uploadTable, transformTable })}`
```
Pass execution to **codegen functions** to produce **program fragments** for distinct parts of the IR.
Pass execution to **codegen functions** to produce **program fragments** for distinct parts of the IR.

```javascript
const ir = {
  center: [-106.1086, 37.7531],
  zoom: 4,
  basemap: {
    url: "https://tiles.stadiamaps.com/styl...",
    provider: "Stadia Maps",
  },
  layers: [
    {
      type: "Choropleth",
      style: {
        fill: {
          attribute: "years_2020_2039",
          scale: "Quantile",
          count: 8,
          scheme: d3.schemeBuPu,
          thresholds: [-210.21, -23.03, ...],
          opacity: 0.75,
        },
        stroke: {
          color: "#ffffff",
          width: 1,
          opacity: 0.5,
        },
      },
    },
  ],
};
```
Each codegen function produces its own program fragment, with callers deciding insertion points.
The **program fragments** are assembled into an output JavaScript program.

```javascript
import mapboxgl from "mapbox-gl";
import waPoTemperatureRegions from "./wapo-…";

mapboxgl.accessToken = "pk.eyJ6fh2gsd6g289d…";

const map = new Map({
  container: "map",
  style: "https://tiles.stadiamaps.com/styl…",
  center: [-106.1086, 37.7531],
  zoom: 4
});

map.on("load", () => {
  map.addSource("wapo-temperature-regions", {
    type: "geojson",
    data: waPoTemperatureRegions,
  });

  map.addLayer({
    id: "wapo-temperature-regions",
    source: "wapo-temperature-regions",
    type: "fill",
    paint: {
      "fill-color": [
        "step",
        ["get", "years_2020_2039"],
        ...
      ]
    }
  });
});
```
The **program fragments** are assembled into an output JavaScript program.

```javascript
import mapboxgl from "mapbox-gl";
import waPoTemperatureRegions from "./wapo-...";

mapboxgl.accessToken = "pk.eyJ6fh2gsd6g289d...";

const map = new Mapboxgl.Map(
    container: "map",
    style: "https://tiles.stadiamaps.com/style...",
    center: [-106.1086, 37.7531],
    zoom: 4
);

map.on("load", () => {
    map.addSource("wapo-temperature-regions", {
        type: "geojson",
        data: waPoTemperatureRegions,
    });

    map.addLayer({
        id: "wapo-temperature-regions",
        source: "wapo-temperature-regions",
        type: "fill",
        paint: {
            "fill-color": [
                "step",
                ["get", "years_2020_2039"],
            ]
        }
    });

    L.mapboxgl.accessToken = "pk.eyJ6fh2gsd6g289d...";
    const map = new L.map("map")
        .setView([-106.1086, 37.7531], 4);

    L.tileLayer(
        "https://tiles.stadiamaps.com/tiles/alida...",
        { maxZoom: 20,
            attribution: '&copy; <a href="https://stadiamaps...",
        }).addTo(map);

    L.geoJSON(waPoTemperatureRegions, {
        style: (feature) => {
            const attr = feature.properties["years_2020_2039"];

            if (attr < -23.03) {
                return {
                    fillColor: "#762a83",
                    fillOpacity: 0.75,
                    color: "#FFFFFF",
                    weight: 0.5,
                };
            } else if (attr < 9.42) {
                ...
            }
        }
    }).addTo(map);
```

```javascript
import L from "leaflet";
import waPoTemperatureRegions from "./wapo-...";

const map = L.map("map")
    .setView([-106.1086, 37.7531], 4);

L.tileLayer(
    "https://tiles.stadiamaps.com/tiles/alida...",
    { maxZoom: 20,
        attribution: '&copy; <a href="https://stadiamaps...",
    }).addTo(map);

L.geoJSON(waPoTemperatureRegions, {
    style: (feature) => {
        const attr = feature.properties["years_2020_2039"];

        if (attr < -23.03) {
            return {
                fillColor: "#762a83",
                fillOpacity: 0.75,
                color: "#FFFFFF",
                weight: 0.5,
            };
        } else if (attr < 9.42) {
            ...
        }
    }
}).addTo(map);
```
```javascript
import mapboxgl from 'mapbox-gl';
import waPoTemperatureRegions from './wapo-temperature-regions';
mapboxgl.accessToken = 'pk.eyJ1Ijoi…';

const map = new mapboxgl.Map({
  container: 'map',
  style: 'https://tiles.stadiamaps.com/style/
});
map.on('load', () => {
  map.addSource('wapo-temperature-regions', {
    type: 'geojson',
    data: waPoTemperatureRegions,
  });
  map.addLayer({
    id: 'wapo-temperature-regions',
    source: 'wapo-temperature-regions',
    type: 'fill',
    paint: {
      'fill-color': [
        'step',
        ['get', 'years_2020_2039'],
      ],
    },
  });
});
```
Working Practice

Sketch visualizations in **design software**  
Explore a wide design space **without code**
Working Practice

Sketch visualizations in **design software**

Explore a wide design space **without code**

After selecting a design, they still had to **reproduce the visualization in code**.
Working Practice

Sketch visualizations in **design software**

Explore a wide design space **without code**

Milliseconds
Map **interactions** with the visual form to **program edits**.

Recover (or infer) **semantic information** from the visual form that we can represent **symbolically**.
Compilers for Visual Inputs

Map interactions with the visual form to program edits.

Recover (or infer) semantic information from the visual form that we can represent symbolically.

Lexer
Parser

reviz

cartokit
Linework

“London Underground Tube Map” · Harry Beck
Shaded Relief

“Walensee und Seeztal” · Eduard Imhof
Perspective

“Wrangell Mountains Shaded Relief Map” · Tom Patterson
Examples

Many, Many (Many) Hours of Painstaking Work

Outputs
Many, Many (Many) Hours of Painstaking Work
import * as d3 from "d3";
const context = canvas.getContext("2d");
regionsGeo.features.forEach((feature) => {
  context.beginPath();
  path(feature);
  const c = color(
    props.years_2080_2099
  );
});

import mapboxgl from "mapbox-gl";
import cancerRegions from "./cancer-regions";
mapboxgl.accessToken = "pk.eyJ6fh2...";
const map = new Map({
  container: "map",
  style: "https://tiles.stadiamaps...",
  center: [-106.1086, 37.7531],
  zoom: 4,
});
map.on("load", () => {
  map.addSource("cancer-regions", {
    type: "geojson",
    data: cancerRegions,
  });
});

import * as Plot from "Plot";
const plot = Plot.plot({
  y: { grid: true },
  marks: [
    Plot.line(covidDeaths,
      Plot.binX(
        { y: "sum" },
        { x: "date" }
      ))
  ],
  Plot.ruleY([0])
});

Many, Many (Many) Hours of Painstaking Work
import * as d3 from "d3";
const context = canvas.getContext("2d");
regionsGeo.features.forEach((feature) => {
  context.beginPath();
  path(feature);
  const c = color(props.years_2080_2099);
});

import mapboxgl from "mapbox-gl";
import cancerRegions from "./canc-...";
mapboxgl.accessToken = "pk.eyJ6fh2...";
const map = new Map({
  container: "map",
  style: "https://tiles.stadiamaps..., center: [-106.1086, 37.7531], zoom: 4,
});
map.on("load", () => {
  map.addSource("cancer-regions", {
    type: "geojson",
    data: cancerRegions,
  });
});

import * as Plot from "@obs...";
const plot = Plot.plot({
y: { grid: true },
marks: [
  Plot.line(covidDeaths, Plot.binX({ y: "sum" }, { x: "date" }),
  Plot.ruleY([0])
],
});
Fixed Entities
import * as Plot from '@obs…';

const plot = Plot.plot({
  y: { grid: true },
  marks: [
    Plot.line(covidDeaths,
      Plot.binX({
        y: "sum",
        x: "date"
      })),
    Plot.ruleY([0])
  ]
});

import * as d3 from 'd3';

const context = canvas.getContext('2d');

regionsGeo.features.forEach((feature) => {
  context.beginPath();
  path(feature);
  const c = color(props.years_2080_2099);
});

import mapboxgl from 'mapbox-gl';
import cancerRegions from './canc…';

mapboxgl.accessToken = "pk.eyJ6fh2…;"

const map = new Map({
  container: "map",
  style: "https://tiles.stadiamaps…",
  center: [-106.1086, 37.7531],
  zoom: 4,
});

map.on('load', () => {
  map.addSource("cancer-regions", {
    type: "geojson",
    data: cancerRegions,
  });
});
import * as Plot from '@obs...';
const plot = Plot.plot(
  y: { grid: true },
  marks: [
    Plot.line(covidDeaths,
      Plot.binX({
        y: "sum",
        x: "date"
      })),
    Plot.ruleY([0])
  ]
);

import * as d3 from 'd3';
const context = canvas.getContext('2d');
regionsGeo.features.forEach((feature) => {
  context.beginPath();
  path(feature);
  context.lineWidth = 2;
  const c = color(props.years_2080_2099);
});

import mapboxgl from 'mapbox-gl';
import cancerRegions from './canc-...';
mapboxgl.accessToken = 'pk.eyJ6fh2...';
const map = new Mapboxgl.Map(
  container: 'map',
  style: 'https://tiles.stadiamaps...",
  center: [-106.1086, 37.7531],
  zoom: 4,
);
map.on('load', () => {
  map.addSource('cancer-regions', {
    type: 'geojson',
    data: cancerRegions,
  });
});
Supporting Data Journalism through Compilers for Visual Inputs

Get in touch!

Parker Ziegler // @parker_ziegler
Ph.D. Student, UC Berkeley

peziegler@cs.berkeley.edu
parkie-doo.sh

Strange Loop
St. Louis, MO • September 21, 2023

github.com/parkerziegler/reviz
observablehq.com/@parkerziegler/hello-reviz

github.com/parkerziegler/cartokit
alpha.cartokit.dev