CS 375 Geolang Project

Installation & Setup Instructions

Prerequesites

The installation for this project is fairly straightforward, with the only prerequisites being NodeJS, npm, and the Typescript Compiler.

To install Node & npm, simply follow this https://nodejs.org/en/download/ link and install the version appropriate for your machine.

Once Node & npm are installed, you can install the Typescript Compiler via the following command:

```
npm install -g typescript
```

This will globally install typescript and allow you to compile typescript code into javascript via the `npx tsc` command.

Setup

With these three things installed, you can go ahead and run:

```
npm install
```

To let npm install all the package depencencies needed for the project.

Once npm finishes installing the dependencies, you can run:

```
npm start
```

This will automatically compile the typescript into javascript, appropriately recompile the antlr grammer files as well as start the server.

Usage

Once the server is running, you can go to `localhost:3000` in a web browser to be immediately greeted by the editor page. Here you can select the COVID data demo, which demonstrates some basic mouse hoverability, geojson loading, as well as csv processing. You may also select the Animations Demo, which demonstrates how Geolang can create animations.

Project Structure

```
|- /dist   :: Webpack'd JS Files
|- /public :: All publically accessible resources
|-- /demos   :: Pre-written demo .geo files
|-- /geojson :: Two geojson files for US data
|- /src    :: All Original Source Code for the Project
|-- /antlr   :: Antlr Library and GeolangParser/Lexer Files
|-- /Geolang :: All Geolang Typescript Source Code
```

```
|--- GeolangMain.ts           :: Main Class File Run by the Webpage's Javascript
|--- GeolangAST.ts            :: Contains all AST Node Definitions
|--- GeolangASTBuilder.ts     :: Class that builds the AST
|--- GeolangP5Generator       :: Class that parses the AST and generates the P5
Javascript Source Code
|--- GeolangTokensProvider.ts :: Class the exports the tokenization of Geolang for
us in the Monaco Text Editor
|-- index.js     :: Loads GeolangMain into the Webpage
|-- P5GeoJSON.js :: The Core GeoJSON Library
|- /views  :: Holds the three webpages' source files -- empty editor, animation
demo, and the covid demo
|- index.cjs :: Initial Node JS File that manages server http requests
```

Project Remarks

Overall, I had a lot of fun with this project, and I certainly learned a lot! That being said, since a lot of this was a learning experience, there's definitely a few pitfalls that I ran into and a few situtations in which I had to band-aid things together for the sake of making a prototype. To highlight the things that causes the most concern and trouble, I detailed them below and the problems I ran into during my implementation.

Lack of Syntax Rigidity

Many of the existing Geolang Features function by supporting a very minimal set of instructions, but allowing the user to supply those instructions whatever arguments they wish, and then just transpiling it regardless. So for example, I did not implement any Geolang Syntax-Level support for P5's "arc" object, instead, I let user pass whatever string they want to the `Model` Command and accept any number of arguments for the `Scale` command. By doing so, I immediately allow support for the "arc" drawing command of p5 with no additional implementation.

However, as it's probably obvious, while this is great for prototyping and quick experimentation, it comes at the cost of having nearly zero effective error checking. During transpile time, there's little way for the transpiler to notice that `Scale` is missing a parameter or has two many, for example.

Running Scripts in a Webpage

This was one of the tricker problems I ran into. Currently, all scripts are run via the "eval" javascript function. This resulted in me running into issues where every time I'd try to re-execute the transpiled JavaScript, it would run into namespace override issues and wouldn't always sucessfully execute. As a band-aid fix to this, I switched to running p5 in a combination instance & global mode. I load p5 globally, to expose as much of the namespace as possible, but then transpile the code to a p5 instance. This way users have access to default p5 variables and functions such as "PI" or "millis()".

However, this itself comes with an additional handful of issues. Specifically, some of the globals exposed are incorrect since they aren't referencing the local p5 instance the user is working within. For example, the "width" & "height" globals represent the size of the global p5 canvas (which is (0 width, 0 height) since we don't want it to take up space on the webpage) and not the actual width and height of the canvas being used.

An additional issue caused by this is that I am effectively running a *new* p5 instance every time you hit run, which can significantly impact performance when running larger scripts over and over.

GeoJSON Loading

Given my current code architecture, there was no easy way to implement a "GeoJSON"-type'd object that can normalize a given GeoJSON geometry to the coordinate system origin. This results in GeoJSON objects not being easily scalable as any scaling I do also applies to the *distance* the object is away from the origin, completely changing its translation. With the right amount of time and architecture, this should be able to be solved fairly trivially.

Additionally, one feature that I wanted to implement but didn't have the time to, is a way of creating multiple geometry symbols from a given GeoJSON FeatureCollection. For example, in the COVID demo, I needed to have the state object, a text object, and it's outlining box, *for each state individually*. This seems significantly less trivial then the above.