How the TPL connects with the front-end

<Live Google Doc>

In this document we **explain in prose how the TPL connects to the front end and its visualizations**, but **see also Step 2 of How2TPL <Drive Link> for an abbreviated description**. We will first explain how the visualizations are prompted from the TPL, then we will explain how to add the interactivity part from the TPL, and lastly we will explain the external libraries that the front end uses.

Prompts are generated with the `[prompt]` tag or automatically when setting a variable equal to something (can be suppressed with the `[no_step]` tag). These variables are also what creates the visualizations.

When you initialize the variable, using `let`, you also set that new variable to an object type. Here is the breakdown of the currently implemented object types that have connected visualizations:

- `Parameter`: Parameter highlights the function arguments.
- `Variable`: Variables are visualized in the variable bank. Used for variables in code. In interactive, the user adds the values of the variables themselves.
- `ArrayIndex`: As of right now, unclear what this does. We use ArrayElement.
- `ArrayIndices`: As of right now, also unclear what this does.
- `ArrayElement`: ArrayElement highlights the array element at a specific index. In interactive, the user clicks the element themselves.
- `AstNode`: AstNode highlights the AST node, which can be a loop call, an if statement, etc. This highlighting is a lighter blue color and is more explicit than the regular line highlighting.
- `ScratchAstNode`: As of right now, also unclear what this does. We use AstNode.
- `Line`: Line highlights the specific line of the code. In interactive mode, the user clicks which line is next/ should be highlighted.

Once the new variable is initialized, you set it to an object returned by a function in `tplHelper.js` (a helper function). This action sets the prompt at this step to the name of the variable, and then uses type and the information returned by the helper function to visualize. For example, when the `Parameter` object is initialized, it is set to the object returned by `helper.get_array_parameters(state.args)`, which returns an object that has a name, type, and value. When the front end reads that the object is the parameter type, it then highlights the parameters.

In the `controller.js` file, the `state` variable is what houses this information. Each time the next button is clicked, the prompt changes. Each prompt has its own state, and each state has a call stack accessed by `state.variables.in_scope` which has every variable instantiated in the TPL where the prompt is located. Since the TPL functions almost like JavaScript code, you can control what variables are in this stack at what time using curly brackets, because the variables function as local variables instead of global variables.

Displaying the front-end aspects, like what should be highlighted at that state, is done by looping through the entire `call_stack` and visualizing accordingly.

The function `addHighlighting()` adds the highlighting to the code and the variable bank at each state also this way. It looks through each variable in the call stack and creates the corresponding highlighting.

The variable bank is displayed similarly, but has a couple of key differences.In the `addVariableBank()` function, the "Variable Bank" gets extracted from the call stack. This variable bank object keeps track of all of the variables that have been initialized at that time and should be displayed in the Variable Bank.

The interactivity is prompted by interactive tags in the TPL. They have the same basic form as `[prompt]` and `[no_step]`, but all fit the template `[interactive("<name>")]`. These tags are put over the variable initializations that are described above. Here is a breakdown of the interactivity tags:

- `[interactive("add_variable")]`: user inputs the value of the new variable into the variable bank
- `[interactive("update_variable")]`: user updates the variable in the bank to its new value
- `[interactive("add_array_index")]`: we don't use this so we are not certain, we use update_variable to update array index

- `[interactive("next_line")]`: user clicks the next line
- `[interactive("conditional")]`: user selects yes/ no radio button based on if conditional is true or false
- `[interactive("array_element_click")]`: user clicks the index in the array that is being referenced
- `[interactive("array_element_get")]`: this uses the scratch area. Still not certain what it does.
- `[interactive("evaluate_expression")]`: we do not use this. Looks like it's to evaluate the expression in a scratch box.

The user input is checked for correctness in the `checkAnswer` functions near the end of `controller.js`.

Lastly, in `controller.js`, we use two external libraries: jQuery and D3.js. jQuery is mostly being used at the beginning of the code to create the initial state of the problem. That is about the only time jQuery is used. The rest of the time we are using D3.js. D3.js is a powerful data visualization library, but we are very much not using it to its potential. For our purposes, we use it to dynamically change the html and css in order to display the visualizations and the highlighting. We also use D3.js to access the user input. The majority of the library calls are from the original Practicum, and we really didn't explore too much how to utilize them ourselves. But the calls that already exist do work as long as the object structures are the same.