CodeGraph

Project Repository

Hannah Swan
hswan@cs.utah.edu
u0939840

December 2, 2017

1 Background and Motivation

The motivation behind this project is selfish, though somewhat motivated by research. In computer science research, as well as industrial software, the code becomes increasingly complex over time. Code is organized based on programmer's styles and for any given project, often many programmers are contributing code over its lifetime. Even well documented programs can be difficult for new programmers to get started with, or for original code authors to pick back up and work later. While all the fine level details are exposed in the often numerous code files, the overall picture of how all these parts fit together can be difficult to visualize. It's also difficult to trace where any data may flow, what functions may affect it, or what other data it might affect.

2 Project Objectives

I would like to explore some ways to better visualize code. This is clearly a complex problem that is dependent on programming languages and techniques. For this project, I'll ignore many of the complexities in order to get an idea of what ways of visualizing code could be helpful, and considers the different context in which they would be helpful (as reference, for debugging, to understanding code for a new programmer, for refressing memory of an older programmer, or for programming assistance in editting, etc.)

I plan to start by looking simply at functions and variables. For each function/variable, what other functions/variables may it affect (what comes after it), or affects it(what comes after it)? Where could the data change? Where does the data flow through the program. How does the program flow?

3 Data and Data Processing

I'm planning on using a simple C++ CPU ray tracer as my raw data, but I'll also probably use some very small C++ program as test data. The idea here is that I'd like to be able to draw some of my graphs by hand for debugging purposes, but also want a larger data set that is more interesting to visualize. Obviously, in an ideal world, this project would include a parser for a program, but I think that is beyond the scope of the project. I plan to manually create my data: a list of all the functions, their variables, and their calls to other functions in the program. This data may

include the order of the function calls, whether or not each variable changes, and any dependencies a variable has within that function.

I'm planning on ignoring advanced programming techniques (such as recursion), and I'll treat classes simply as separate variables and functions.

4 Design

Figures 1, 2, 3, and 4 shows design ideas. In particular, I plan to visualize code through graphs. I'd like to be able to do more than just a call hierarchy graph, which are usually too complex to be helpful (as the program grows, anyway). I'd like to have particular views to help visualize different aspects of the program. For each of these views, I also want to make sure that only the needed information is shown, while everything else is ignored. I've explored this idea and pared down to a few different views that I think would be the most helpful.

Figure 5 shows a possible graph for viewing functions (this is similar to a call hierarchy.) Figure 6 shows a way of visualizing variable dependencies between functions. Figure 7 takes into account that the function calls have an order to them, which may be an important aspect to visualize. Finaly, figure 8 would focus on a particular path between two functions.

5 Features

5.1 Must have

At the very least, I'll need to have a graph (most likely a tree) showing, essentially, the call hierarchy. The interaction will work with a list of functions and/or variables, should show the path(s) of all functions/variables it is dependent on, is dependent on it, and/or both. I think it's also most important to add the path view, at least between a particular call, if not a chain of calls.

5.2 Optional

I'd like to have at least one more graph view. (In fact, I think it is a must-have, but I haven't decided which of the alternative views is the must-have.) I'm leaning towards the variable view, but it might become clearer as I progress that another view, like the data flow view, is more necessary. There are many more views that I've considered: a variant of the call hierarchy including call and return paths (where each node would have an enter and exit connection) or the addition of options for showing either all dependencies for a chosen functions/variables or all functions/variable that are have the selected function/variable as a dependency (i.e. all the paths to a function/variable or all the paths from a function/variable.) These kinds of views would be particularly helpful for debugging.

I'm also considering including several lists/text boxes with interactivity between then. For example, at the very least, I'll need a list of all the functions and some list of variables. These could be connected so that when a function is connected, the variable list is sorted or filtered by variables within or affected by that function, and vice versa. Ideally, though it's beyond the scope of the project, it would also be helpful to include a text box with the original code for a selected function, and possibly highlights all calls and declarations of the functions or selected variables. However, this would require the parser that I'll have to ignore.

Ideally, the graph views would have some sort of level of detail processing for zooming in and out, with the most zoomed in view showing the code, and the most zoomed out view simply showing one node for `main()`. Again, this is probably outside the scope of the project.

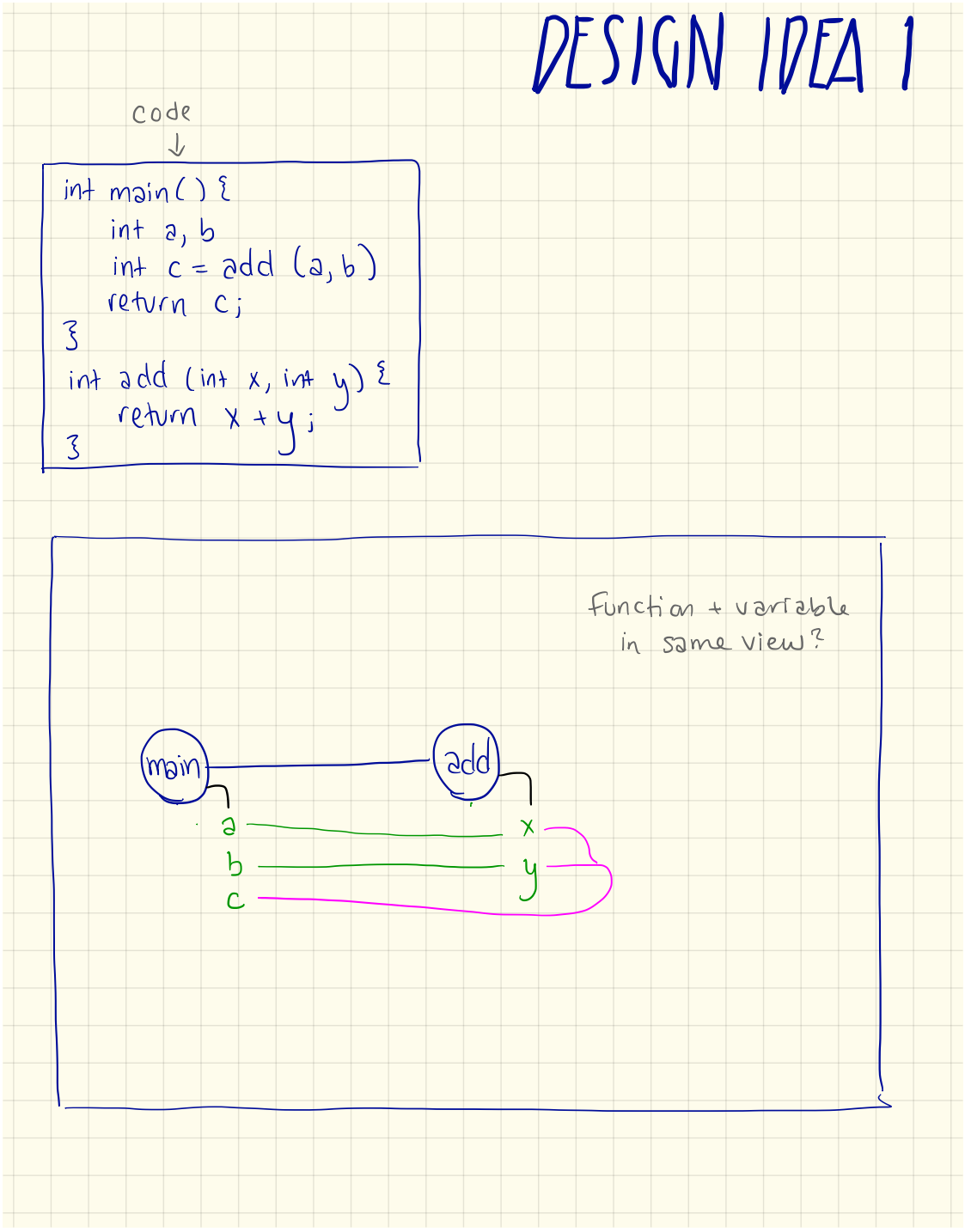

Figure 1: Design 1 Idea

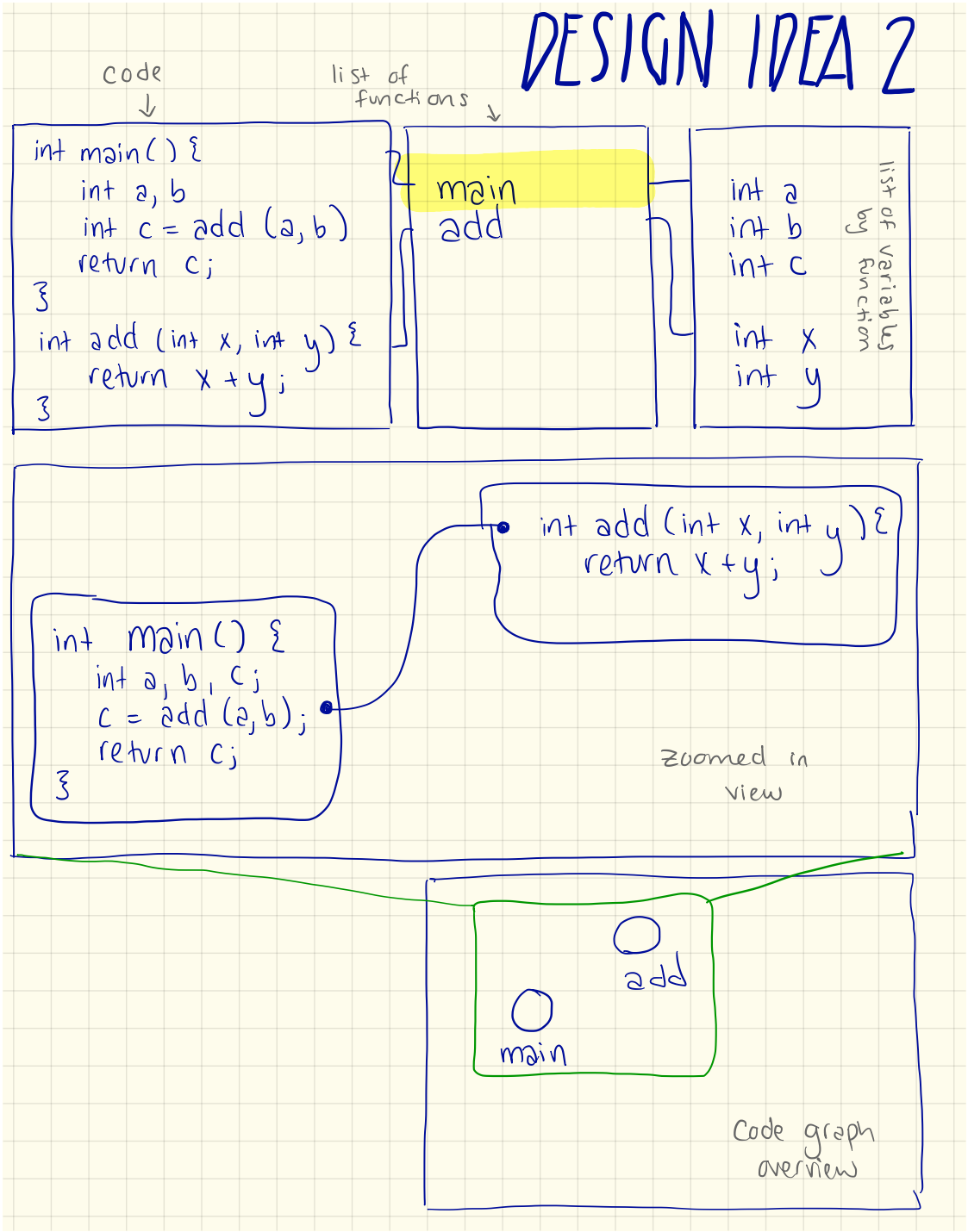

Figure 2: Design 2 Idea

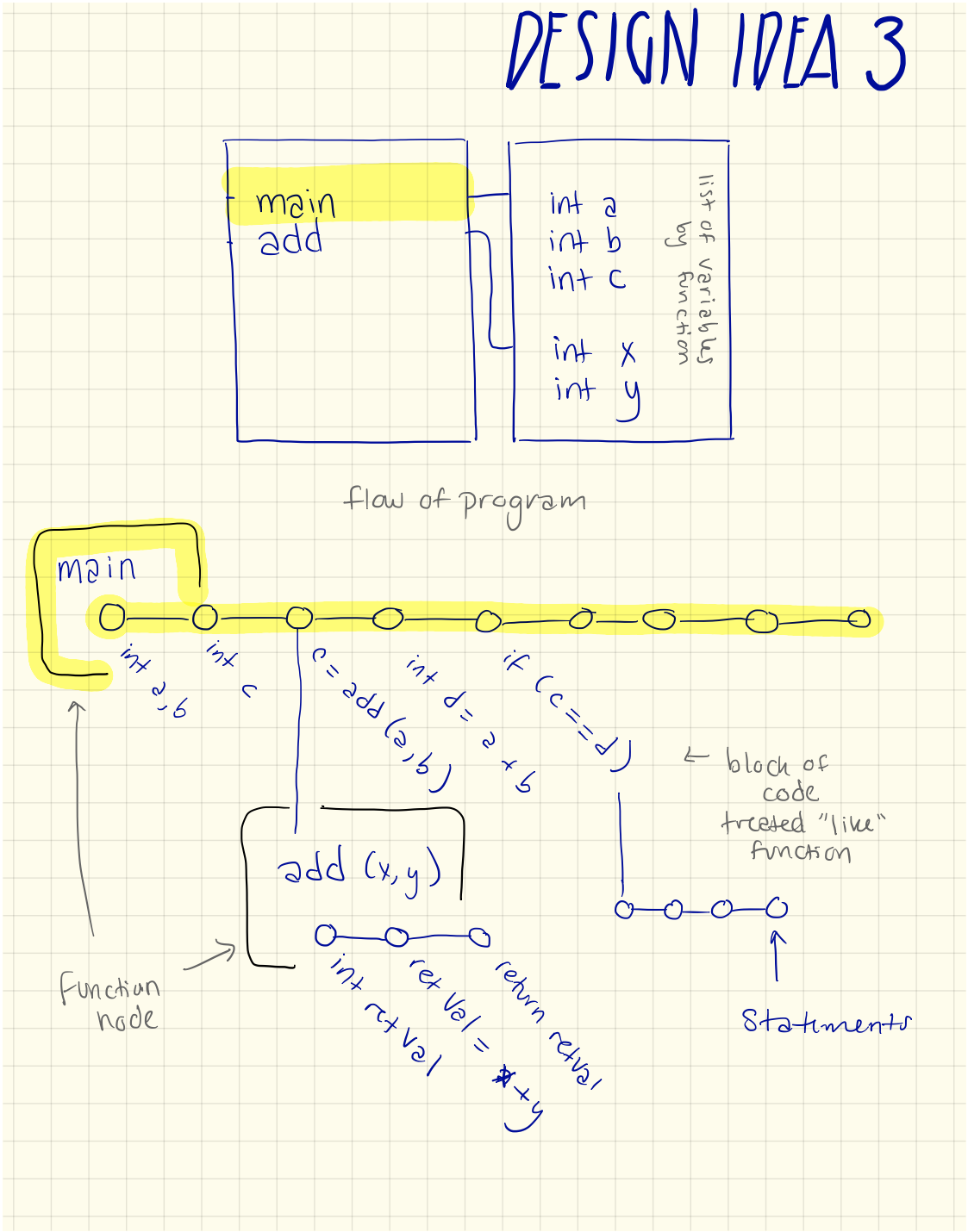

Figure 3: Design 3 Idea

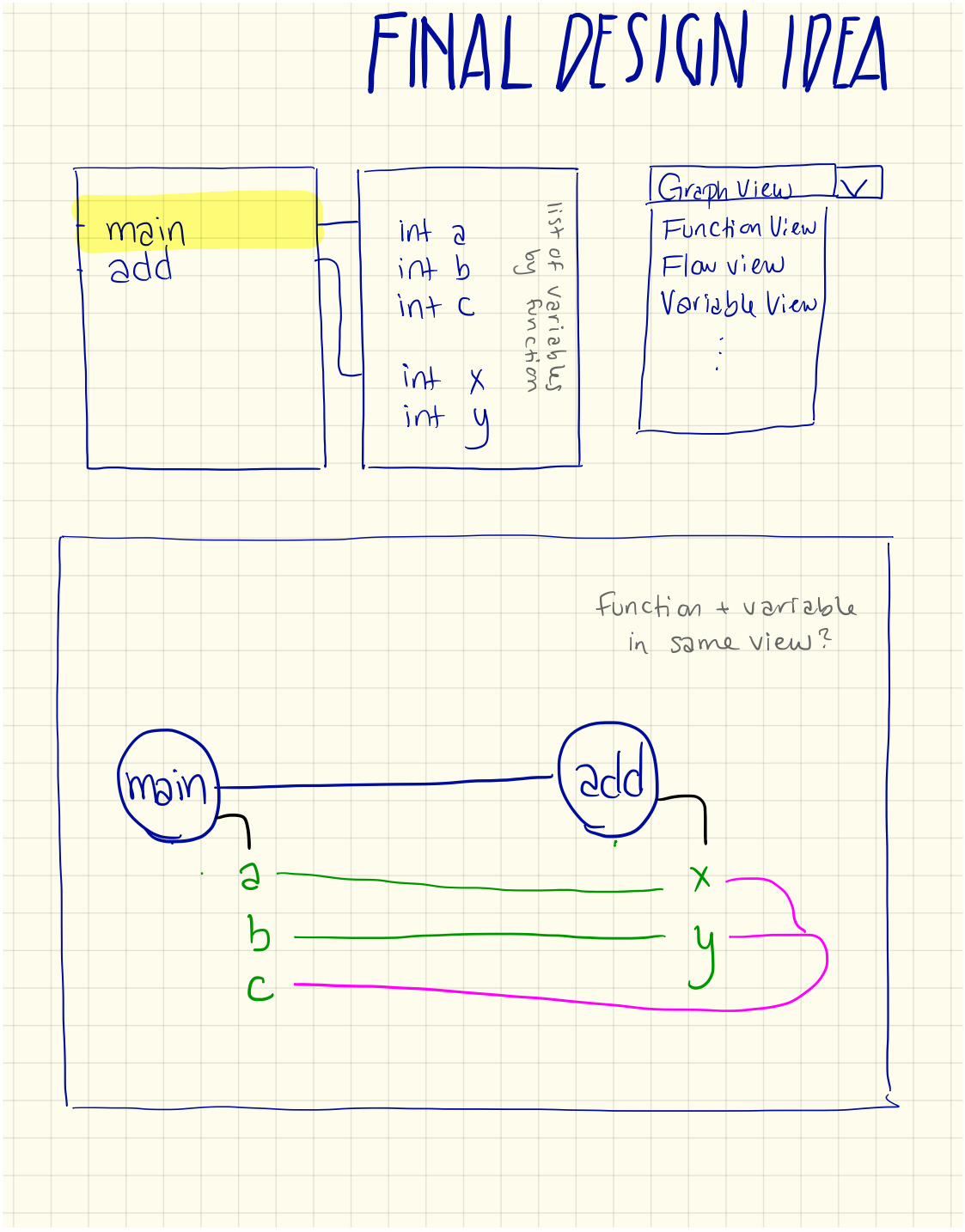

Figure 4: Final Design Idea

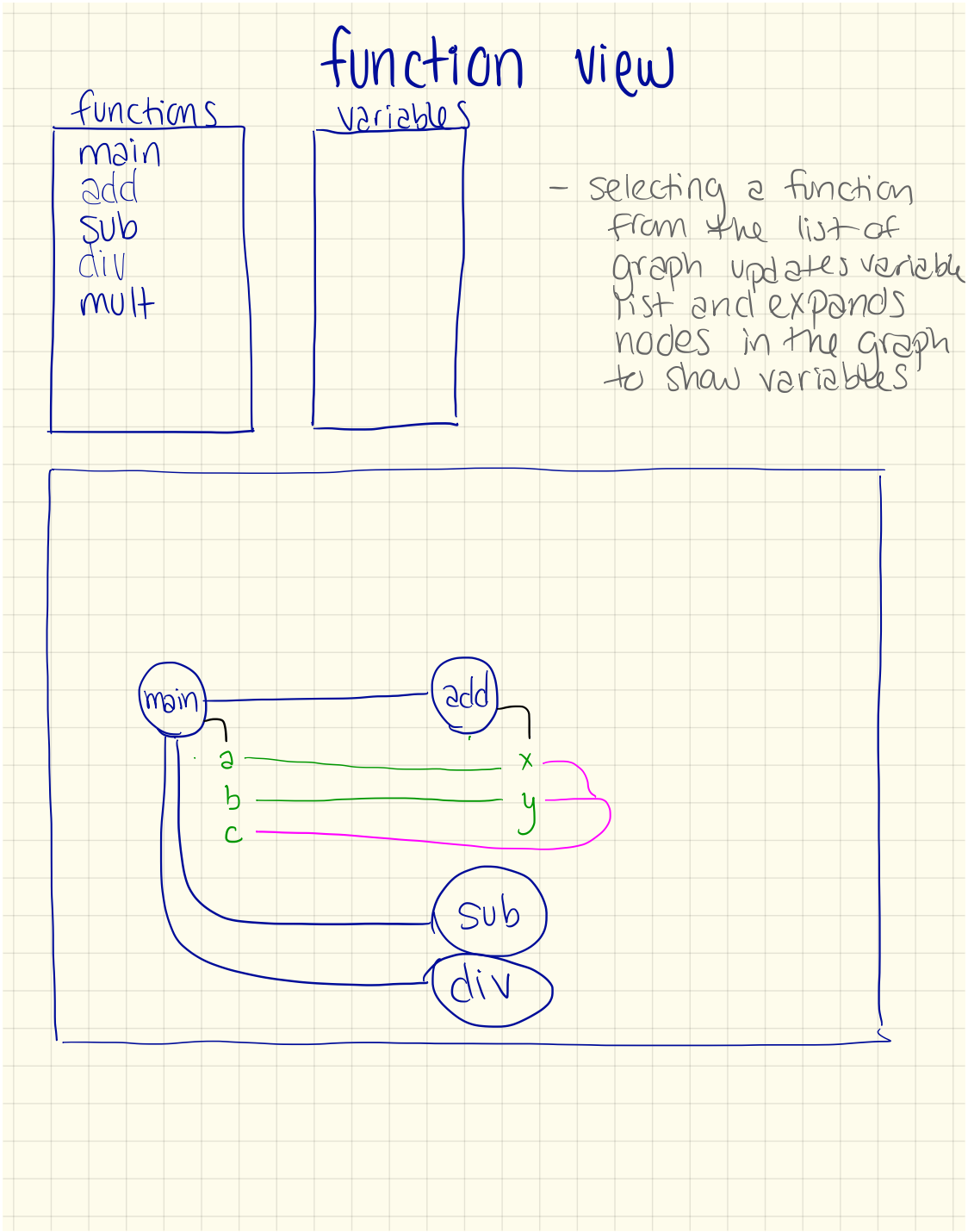

Figure 5: Function View

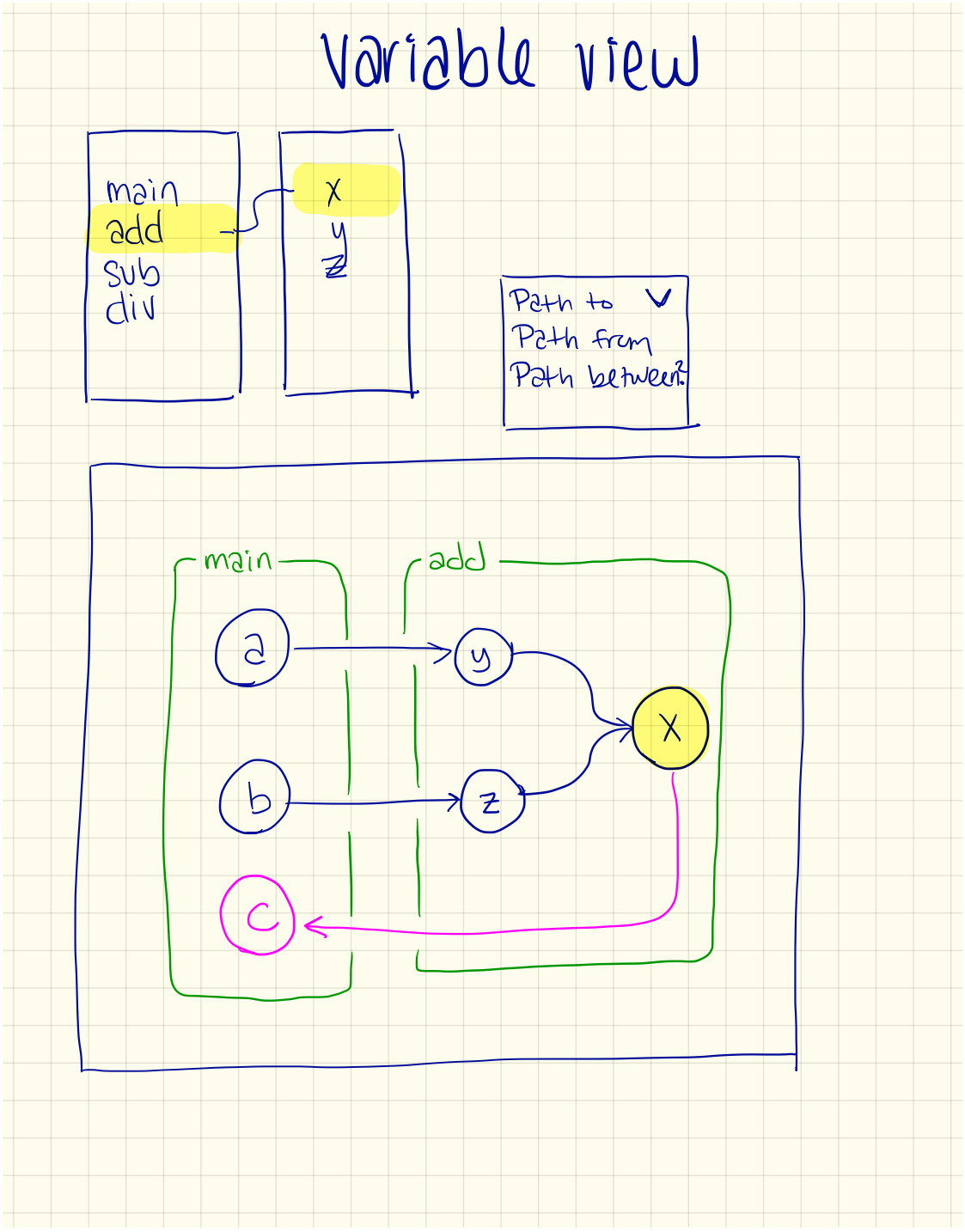

Figure 6: Variable View

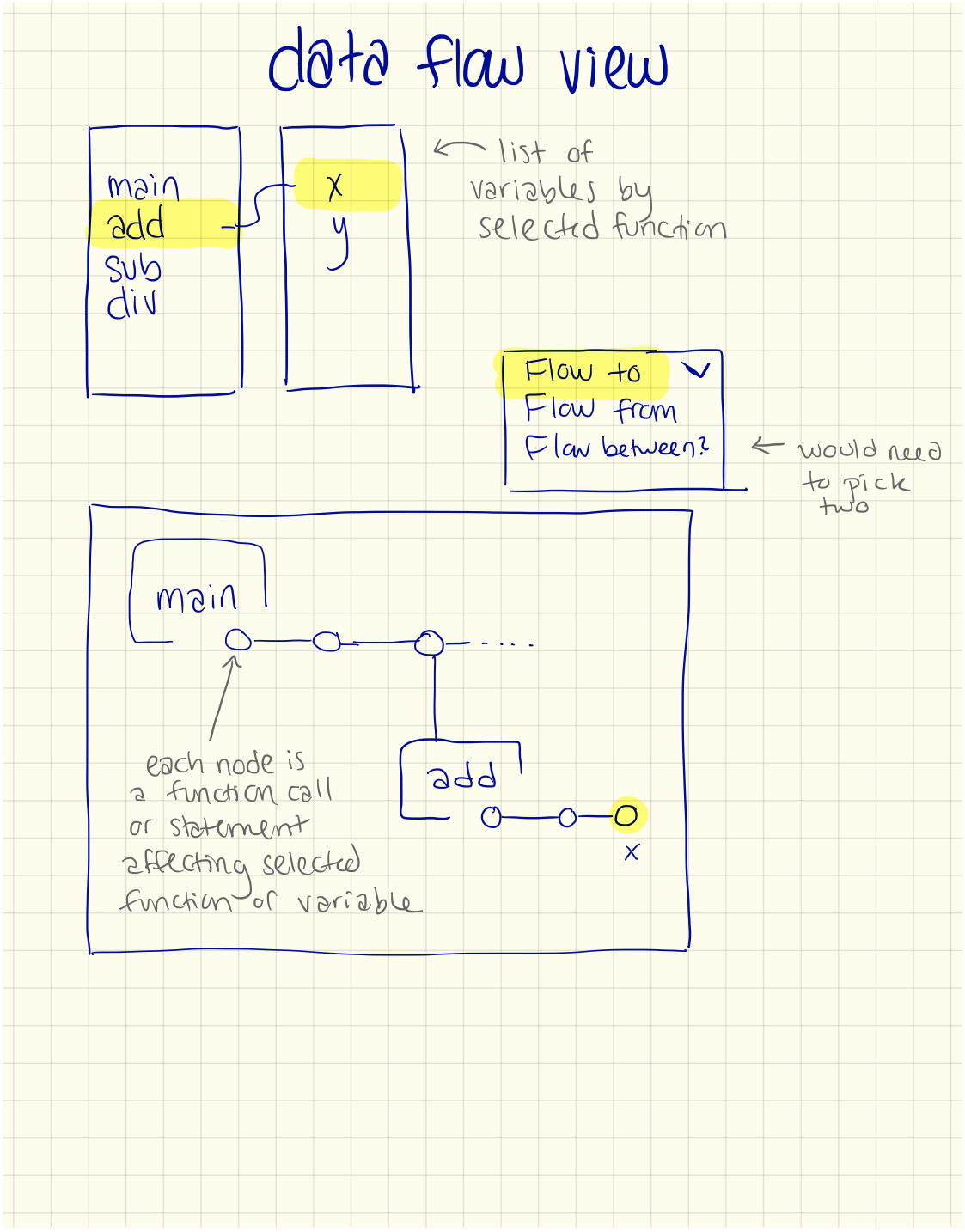

Figure 7: Data Flow View

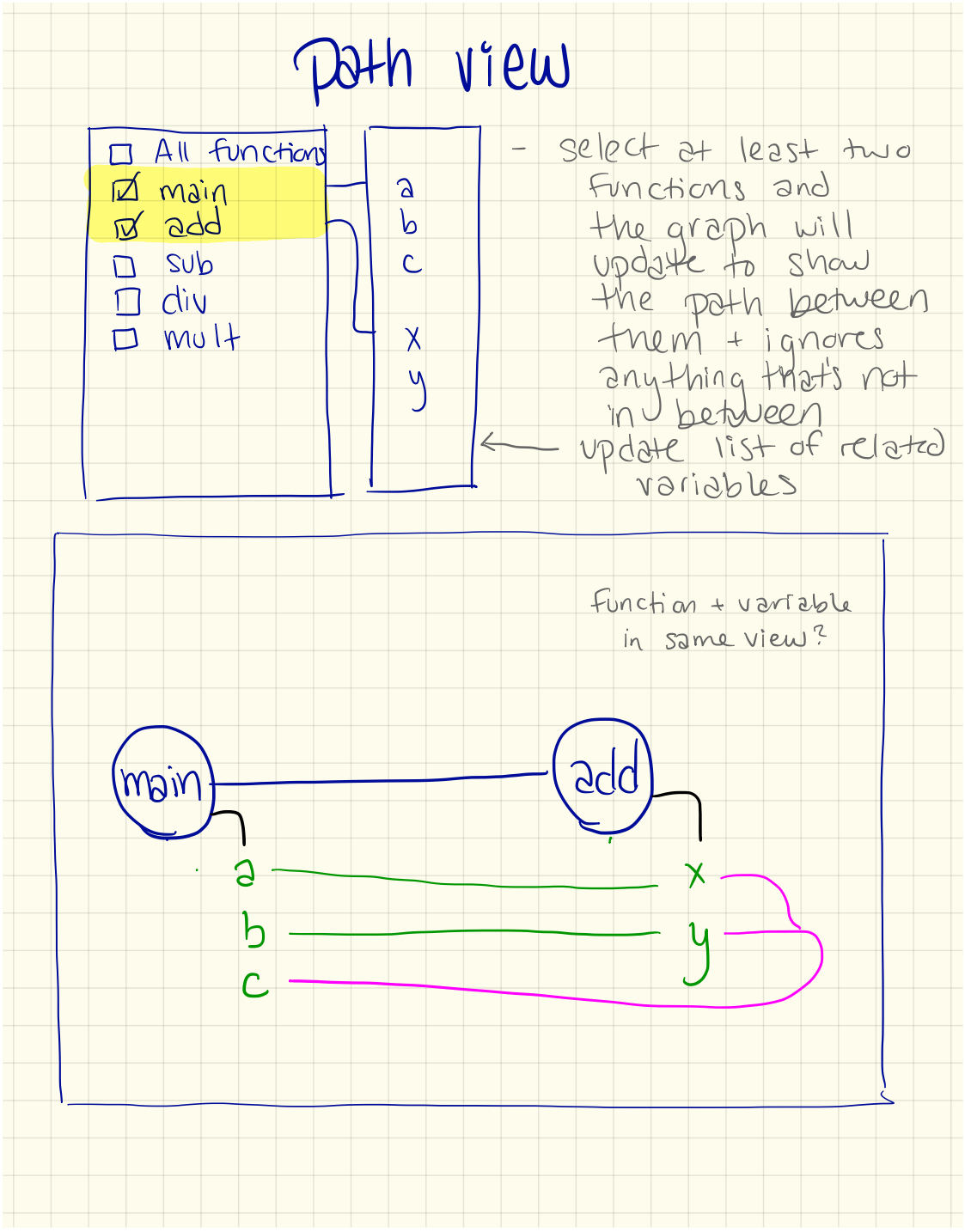

Figure 8: Path View

Another thing I'd like to consider is including a class view, or a view that includes classes. I imagine this will become clearer as the must-haves become reality.

6 (Rough) Project Schedule

6.1 Week One

Data processing and start building and displaying tree/graph (simple call hierarchy)

6.2 Week Two

Add list of variables and interactivity with the graph (highlighting relevant paths with relevant colors)

6.3 Week Three

Alternate graph view: functions as nodes, but rearranged to show data flow/flow of the program vs. call hierarchy

6.4 Week Four

Alternate graph view: variable graph (probably by function)

6.5 Week Five

I'll pick the most needed optional feature(s) for the final week

7 Process Log

7.1 Week One

I underestimated how difficult it would be to get my data. Instead of using the ray tracer I originally planned, I decided to go even simpler. I wrote a simple C++ program that calculates roots to a quadratic polynomial. I included many "helper" functions that are unnecessary in an actual program, but has the benefit of being easy to understand for these test designs. I hand wrote the data for this simple program. Basically, each function is an object, and it holds some other objects or list of objects.

For example, here is a selected function from my simple program.

```
float subtract(float x1, float x2) {
        float x3 = negate(x2);
        float y = add(x1, x3);
        return y;
}
```

This code was specifically written to have easy to understand function calls and variables for this experimental design. Here is the JSON data for this function:

```
{
        "key": "subtract",
        "value": {
```

```
                "arguements": ["x1", "x2"],
                "returnVariable": "y",
                        "functionCalls": [{
                        "key": "negate",
                        "arguements": [ "x2" ],
                        "returnVarEq": "x3",
                        "line": 62
                },{
                        "key": "add",
                        "arguements": [ "x1","x3" ],
                        "returnVarEq": "y",
                        "line": 63
                }],
                "conditionals": [],
                "arguementsChanged": [],
                "variables":
                [
                        {
                                "key": "x3",
                                "value": {
                                        "parentVars": ["x2"],
                                        "parentFunctions": ["negate"]
                                }
                        },{
                                "key": "y",
                                "value": {
                                        "parentVars": ["x1","x3"],
                                        "parentFunctions": ["add"]
                                }
                        },{
                                "key": "x1",
                                "value": {
                                        "parentVars": [],
                                        "parentFunctions": []
                                }
                        },{
                                "key": "x2",
                                "value": {
                                        "parentVars": [],
                                        "parentFunctions": []
                                }
                        }
                ]
        }
}
```


We can see that even for a simple function, the data can be very complicated. In this case, there are no conditional (if) statements that would cause branching, and there are no arguements

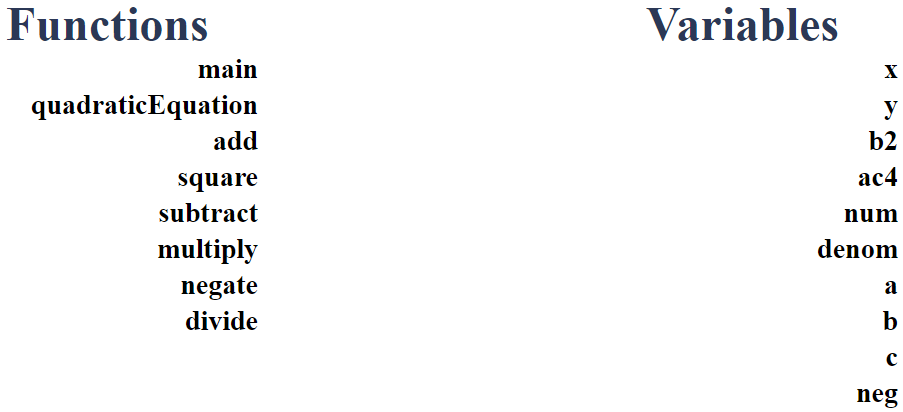

Figure 9: Function and Variable Lists

changed.

7.2 Week Two

I finished hand writing the data, and started adding the list views. By default, a list of all functions in the program shows up. Then, if any elements in this list are clicked, a list of all the variables for a function is populated. This is shown in figure 9, where `quadraticEquation` was clicked.

7.3 Week Three

I decided to implement a graph view for functions. However, I had a specific view in mind that, as far as I know, the default layouts would not fulfill for me. I wanted the graph sorted by the return value of the function, if it had one. My thought is that it would be helpful to see the return value of a function as the root of a tree, with the arguements as leaf nodes. However, this means that some varaibles would have to appear as multiple nodes in this tree. Additionally, I'd have to assign the positions myself. I wrote a function that would build the tree, starting at the return value and working it's way through "parent" variables. (I realize the terminology is unintuitive. In the tree, parent variables are children nodes.)

This took quite a bit of debugging, as I made a mistake while forming the links. I was mistakingly referencing the original array, instead of my new list of nodes meant for my tree structure. However, I did eventually solve the issue, and we can now tree graphs for functions with return values. This can be seen in figure 10, once again for the `quadraticEquation` functions. Like the variable list, this tree is drawn when the mouse clicks on the function name.

7.4 Week Four

The problem with my current graph view is that it only works for functions with a meaningful return value, which is not every function. In particular, our `main` function does not have a meaningful return value, so there is no logic for how to draw the variable graph. In this case, I decided to run a force simulation. This also took quite a bit of debugging as I was unfamiliar with this type of layout, and also had to reassemble the data to make it work with the simulation. (In particular, the "links" aren't explicit in my data).

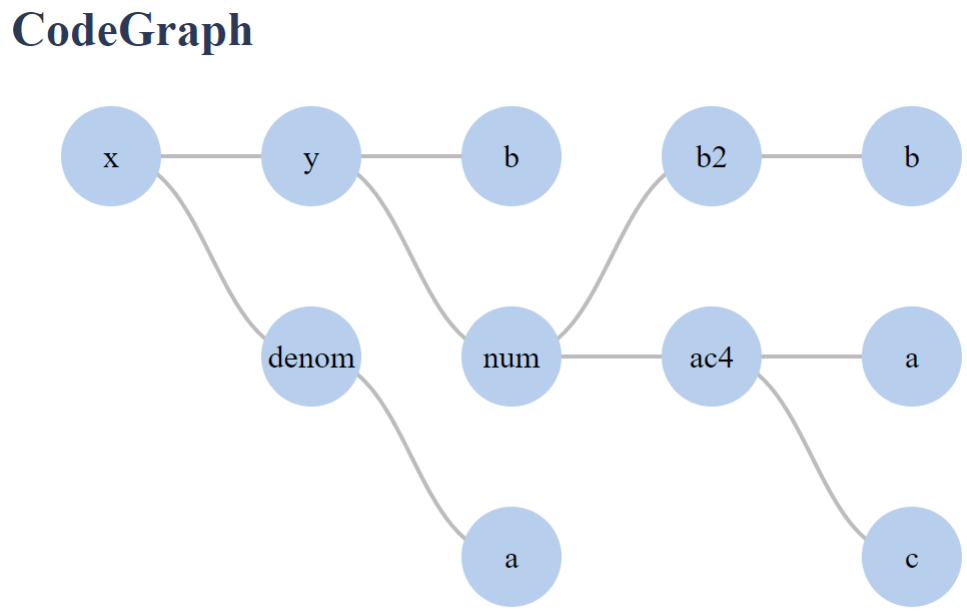

Figure 10: Variable Tree

Shown in figure 11 is the simulation result for the variables in the `main` function, which appears on the mouse click for `main`.

What is interesting about this simulation is that we can see quickly which variables are used often. However, it's also unfortunate that some of the links end up being hidden - like the one between `root2` and `z`. For this reason, I considered using the line value to apply an additional force on the nodes. (That is, the line value being the line in which these variables appear.) However, this is not data I thought to include when I hand wrote the data, and since the writing of the data was costly in time, I didn't want to go back to add it. If I ever continue to work on this, I would like to try to add this information in and see if it helps the simulation.

7.5 Week Five

At this point, there was still a lot of things I wanted to include, but I think I underestimated how difficult it would be working with such complex data. I decided that the most important features that were missing was a graph view with the functions, and highlighting nodes in the variable graphs. I believe this is particularly necessary for the variable tree views, since there can be repeated nodes.

Figure 12 shows the function graph. This was accomplished in a similar way as the variable graph simulation, though the way the data was assembled for the simulation differed slightly. This was also set to be the first, default view. I was pleasantly surprised by this simulation. What is nice about this simulation is that the most "important" function migrated to the middle, while my unnecessary "helper" functions were pushed towards the edge. I'm curious whether this kind of pattern would hold for larger programs, and whether it might help to eliminate unnecessary functions.

For example, in figure 12, we can see that the `divide` function is only called by `quadraticEquation`

CodeGraph

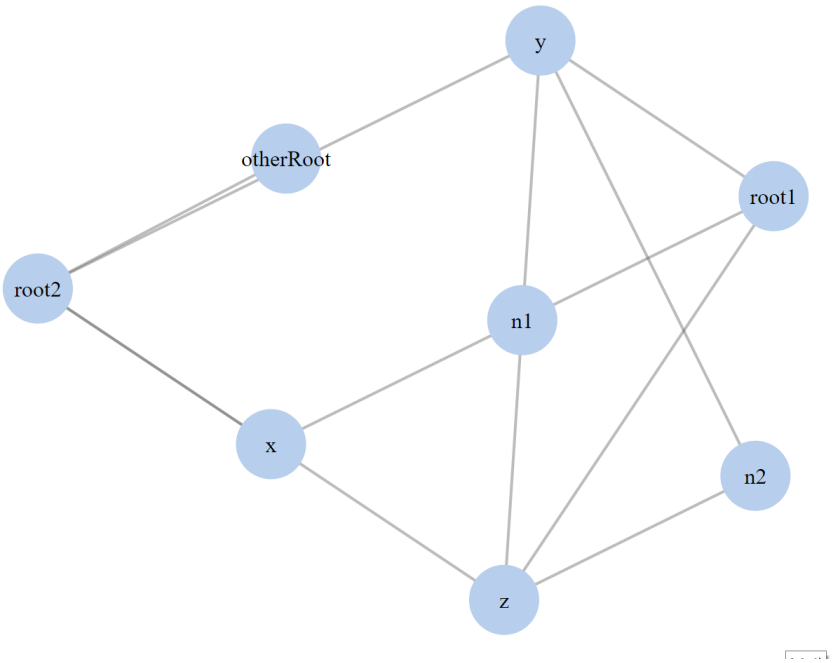

Figure 11: Variable Graph - Force Simulation

CodeGraph

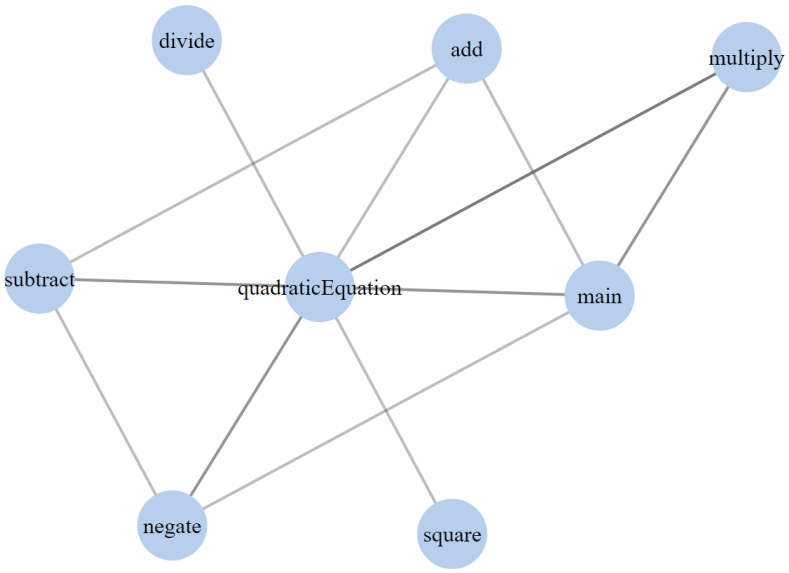

Figure 12: Function Graph - Force Simulation

function and, based on the lightness of the link, it is only called once. So, it's probably unnecessary.

Also, in this case there are multiple links between functions if they are called more than once. In retrospect, I should have included a number of calls in my data, and used this as a link strength.

Figure 13 shows the highlighting when the variable `a` is clicked on, within the list for the `quadraticEquation` function.

8 Evaluation

Mostly I learned more about how this information could/should be displayed. The displays that I've included have some useful features in that we can see some unnecessary variables. For example, in figure 13, we can see that the `denom` variable is only dependent on `a`, and only used to calculate `x`, so it is an unnecessary variable.

This leads us to believe that we could "collapse" this part of the tree, or, in other, more practical terms, we can eliminate this variable in the code.

I also learned a lot about what should be included in the data, which makes me glad that I didn't spend the time to write a parser. I realized that I included a lot of information in my data that I never used, and may not have needed. However, there was also information that I wish I included, such as line information. I would have used this to help position the nodes in the graph view, as well as forces in the simulations, to make variables drift to one side or the other depending on how early they show up in the function or where they are used.

Again, there are many more things I wish I could have played with. For example, after doing

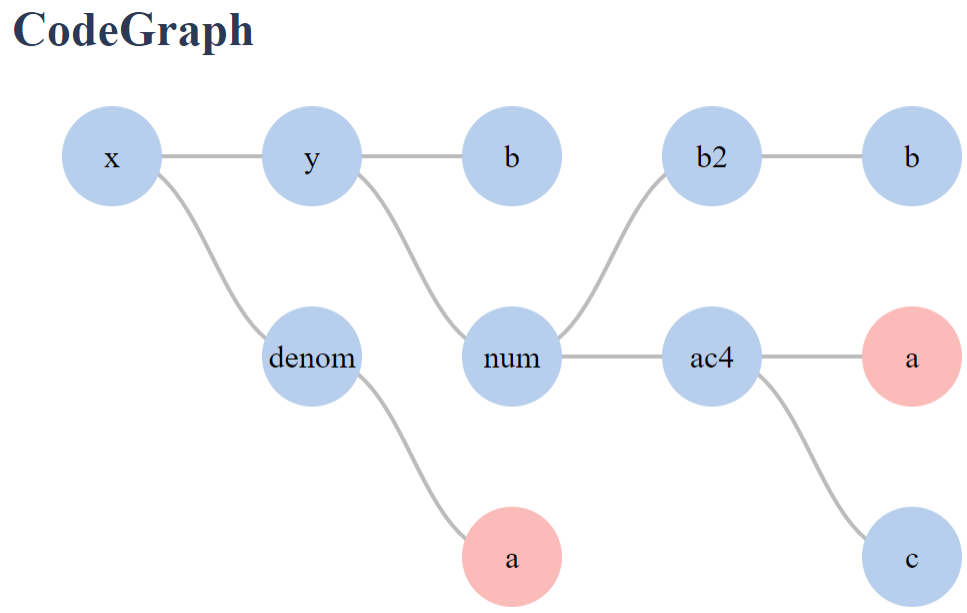

Figure 13: Highlighted Variable Nodes

the varaible tree view that was built based on the return value, I wanted to redraw these "trees" based on different variables. For example, starting from the arguments, and working towards the return variable. Or, selecting one of the middle variables and collapsing any repeat nodes for that particular variable.

Another important view would include both functions and variables. I had this kind of view in mind as I was working, so some of the building blocks are here. I'd like to start with the default function view and expand some function nodes (on a click), to show the variable views inside, and show connections with the variables between functions. This was why I started by sorting the variable views by return and arguement variables.

Overall, I enjoyed working on this project and expect I'll return to the idea at some point to explore more ways to visualize code.