Names: Kaylee Bigelow and Jonah Stegman

Date: March 25, 2021

Course: CIS*4650

Checkpoint Two - Documentation

What has been done

To start this assignment we fixed our code from the previous submission. This was done by revising the grammar so that local declarations and statements lists could come in any order. In addition, we simplified some other areas of the grammar to make it more efficient. The third part we fixed was the regex for comments in our flex file.

Once these were finished we created the semantic analyzer file and started working on the symbol table. The flag for the semantic analyzer was added to cm.java and all the classes were put in semanticAnalyzer.java. From there we made our hashmap functions and our print function. Then we added all variables when they were defined in assignExp to the hashmap and also all functions to the hashmap. The variables for each level were printed out right before leaving any scope. After this was working prints were also added to show when the analyzer entered and exited a block, global scope or function. The function name was also printed when entering a function. The final thing that was done was when printing out the function type the types of its parameters were also printed with it (i.e. test(int, int) -> int).

Finally we added the type-checking. The type-checking was the hardest part of the assignment. While the types were already passed along for the use of the symbol table there were many edge cases that had to be caught. We started out with the simple cases such as checking both sides of an operand then made these cases more complex. The edge case that often led to more work were around arrays. Often arrays had to be handled with care to ensure that they were handled as expected.

After these had all been complete we did error checking and more testing. Through that process we attempted to break our code, fixing any errors we encountered along the way.

This assignment was completed mainly through pair-programming, thus it is hard to distinguish who did what parts of the assignment. However through pair-programming we feel that we each did an equal amount of work on the assignment.

Techniques and Design Process

Before starting on this assignment we decided how we would do type checking and in what order the different aspects of the assignment had to be done. The ordering of the assignment was fairly easy to design. First we had to get our previous code to an improved state, then figure out how to store/pass types, create the symbol table, and finally implement type checking.

The major part of our design process included coming up with a way to organize type checking. In the end we decided that it was best to work incrementally. Thus we started with checking two sides of an operand, then multiple variables in an math equation, working towards assignments, ending on returns and function calls. We felt that this order allowed us to understand the type checking at each point, making the following steps easier to execute.

Overall, the main technique we used throughout the assignment was to work incrementally. If we take on too big of a section at one time it leads to confusion and often having to redo aspects of the type-checking.

Lessons gained

Throughout this process we learned valuable lessons concerning how code is processed. We came to realize how complex the process of type checking can be especially if more types were to be added. From this we take away to not underestimate implementation of error conditions. Often as programmers we overlook edge cases and error conditions but this assignment showed us that not every case that we think is an edge case is an edge case.

Once again we have learned how much work goes into creating a compiler, and how appreciative we should be that they are readily available on this day.

Assumptions and Limitations

While completing this assignment we made a few assumptions along the way. The first being that to avoid having repeated errors concerning void variables, we save all void variables as the type int. When the user creates a void variable the error will be shown to the user thus allowing them the chance to fix it. However, saving the variable as an int allows for easier debugging on the user's behalf as the output will not be flooded with the same error again and again in different locations.

Another assumption that we made is that the user cannot set two arrays equal to one another. This assumption was made as the semantic analyzer does not store the values of variables thus the array sizes cannot be verified in all cases. When the user attempts to set two arrays equal to each other, an error will be shown so that the user can fix their mistake.

The final assumption that was made is when a function is declared with the parameters set to void, during a call the user does not need to send in void, they may simply leave the parameters blank. This was done as it is the standard in many languages and makes it easier on the user.

Possible Improvements

A possible improvement to our semantic table would be having the ability to recognize different types such as char. This would take a lot of work and require changes to the abs tree and flex files as well but it would allow for much more complex programs.

Another possible improvement would be to check array bounds if the size of the array was statically set (i.e. int a[5];). These cases would be possible sometimes to check bounds as the size is known. This could allow for additional errors to be caught before runtime.

A third improvement could be the addition of function headers in the semantic analyzer. Currently, they would produce an error saying that the function was redefined but this could be changed and fixed to allow for calling functions that are declared below but have function headers.