Names: Kaylee Bigelow and Jonah Stegman

Date: April 4, 2021

Course: CIS*4650

Checkpoint Three - Documentation

What has been done

The first thing that was done was updating the NodeType class to be able to store the offset value in the node. This was needed so that the compiler can know where each variable is stored. From there other parameters were added to different Exp types. These allowed for more in depth information to be stored about the classes. Next was added emit functions to generate the code in the correct format. Once all of these beginning steps were done we moved onto creating the instructions.

There were many different types of instructions to generate, starting with the prelude, and finale. From there other instructions we generated in the order of functions, mathematics, assignments, if-statements, while-statements, function calls, and finally arrays. Each of these code generation steps had many steps themselves which often lead to finding bugs in previously completed sections. After the code was generated some array out of bounds error checking was added in. This was left to the final step as it required the greatest knowledge of how to generate instructions.

Overall, we completed a working code generator which can do all of the main blocks of code that are possible in the C- language. There are some aspects missing that will be discussed below, however we are proud of how much of this compiler we were able to accomplish this term.

What was changed from previous checkpoints

There were a few changes that were made to previous checkpoints during the completion of checkpoint three. The first change was fixing parameters to allow for arrays to be able to be passed. This was done by updating the cup file to allow for a flag to be set if a variable was an array. Through the flag both the semantic analyzer and show visitor tree were able to understand whether a parameter was an array or not. Since the size of a parameter array is unknown the semantic analyzer displays the value as -1. The value of -1 was chosen as an array that is initialized to -1 is invalid so that no code should have a negative value predefined for an array. This allows our code to tell that this is an array that size has not been set vs. a normal array that would have its size set either statically or through a variable. Because of this change with the flag some adjustments had to be made to the error checking for type matching in calls to functions. They were changed to now just compare the types to each other and if it got an array to ignore the size of the array when comparing (i.e. int[2] compared to int[-1] will say the types match).

A second improvement that was done was including the predefined input and output functions in the hashmap. This was pivotal in making checkpoint three work as input and output are called in almost any program. In addition, adding them to our hashmap allows us to check if a program ever tries to redefine them and then can throw the appropriate error.

A third improvement that was done was making it so checkpoint two's semantic analyzer always runs when the -c flag is given. However, only the errors are printed to the screen. The .sym file is still only created when the -s flag is set. This allows for semantic errors to be shown when compiling the code.

Techniques and Design Process

The main technique that was used in this project was incremental design and coding. This technique allowed for us to check that every step along the way was completed correctly before moving on to the next step. Without this technique we believe that we would have ran into many more errors along the way, which in turn would have slowed our progress. The other technique we used was testing early and often. This technique, similar to the above, allowed the number of errors we encountered at the end to be lessened. We were able to catch errors at each stage, and thus able to identify them. Finally, we used the technique of pair-programming. We found this allowed us to be on the same page, and work together to understand the problems as they arose. Because of this technique it also allowed us to point out potential problems that could arise before they do. In addition, this technique enabled us to switch between developers as not only were we on the same page but we also knew what all the code did.

The design process that we took was based off of the Incremental Development slides that we were provided with for this assignment. We laid out each step and then decided on the order in which they should be done. When we started a step we would first lay out a plan of what needs to be done in this part of code generation. This allowed us to update our plan if our understanding changed based on a previous task. As well, it allowed us to plan out the tests we would run to check if this step was working. In our design plan we accounted for this testing time for each step of the way to verify that we completed a step before moving on.

Lessons gained

There were three main lessons that were learned throughout checkpoint three. We learned how to convert code to tm code. By this we mean that we learned the processes involved in converting code to simpler code. This allowed us to fully understand processes like backpatching and memory management. We learned how calculations for jumps work in memory and how to store things in memory and how their offsets allow you to access those values.

Another lesson that was learned through this process was how complex recursion actually is compared to a simple call. The way the calculation needs to be done to allow for the offsets to be correct was a hard lesson to figure out.

A third lesson that was gained was to create multiple branches and merge often. This allowed use to revert back our changes when mistakes were made; as they were made often. This process allowed us to save tons of time fixing issues and resolving old problems that we previously had solved.

The final lesson that was learned was how to appreciate how hard compilers are to write and how lucky we are that the compiler for java and c already exist for us.

Assumptions and Limitations

There were a few assumptions that we made when making the code generator for this assignment. The first is that arrays start at 0, as they do in C. In general we treated arrays as they are done in the C language.

Another assumption we made was that if the tm code can be produced it will be. This means that if recoverable errors exist that are shown in the parser and semantic analyzer the code will still be outputted to the file.tm. We were unable to make the code determine which errors were recoverable and which weren't. This is why it is possible that the codeGenerator.java program could crash after some semantic analyzer errors are thrown.

The limitations that existed in checkpoint one and checkpoint two still apply for checkpoint three. Not all errors are caught by the semantic analyzer such as missing semicolons and missing brackets. Some limitations that exist in checkpoint three specifically are that when storing numbers there is a maximum bounds as the space stored is finite and not flexible. This means that if you run the fac program with a value of 100 the result will go out of bounds and be set to 0.

Another limitation is with passing parameters. There is a finite number of parameters that can be passed as there is a finite amount of memory that can be used. This means that if more than that many parameters are passed they will write over other variables that are stored in code.

A third limitation is that arrays can only be accessed with integers. While the code will not break, no result will be calculated for the array if they are declared or accessed with a math equation, or a variable. This limitation leads to another limitation as array out of bounds errors can only be checked if the array is accessed with an integer. Since arrays cannot be accessed with anything besides an integer error checking for those cases is not done.

A final limitation is that when calling a function a math equation cannot be used as a parameter. Math can be done outside of a function call and the result may be passed into the call, but it cannot be calculated inside of the call. This limitation is due to time, given more time this limitation can be overcome.

Possible Improvements

Some possible improvements for this compiler could be to allow for more types to exist such as char's or strings. This would allow for more complex programs to be able to be produced. If strings or chars were added then you would also be able to allow for output to print strings allowing for more effective runtime error messages to be given.

A second improvement could be having more memory assigned to each variable thus allowing the storage of larger values. This could fix the limitation that was explained about fac.cm.

A third improvement could be the implementation of prototypes. This would improve the readability of programs that were run through the compiler. In addition, prototypes would allow for calls to functions that are above where the function actually is. This would make understanding the flow of a program easier for some to understand.

A fourth improvement is to make arrays more flexible. Currently, as discussed in the limitations, arrays can on be created and accessed with integers. Given more time this could have been improved to allow for math equations, and variables to be used to access or create an array.