CIS*4650 Group 6 Checkpoint 3 Report

Amanda Hahn (ahahn01@uoguelph.ca, 1150580)

Nicholas Waller (nwaller@uoguelph.ca, 1122924)

Total Work Completed

We believe we have met all of the requirements for this checkpoint. We modified the previous absyn definitions per the spec given, implemented our final visitor class Java file, CodeGenerator.java, which utilized 2 main functions, emitRM and emitRO to emit register memory (RM) and register only (RO) instructions. These functions then print out formatted lines of TM assembly code into our output .tm file. Both A1 and A2 go into their respective file formats when triggered using their flags, and a third flag, -c, has been implemented to run code generation. There were many cases that were handled, such as recursion (including mutual recursion), local and global variables, loops, conditionals, and more. Finally, we implemented runtime error checking for out of bound array indices, reporting an error and halting if they occur.

Design Changes over Time

In general, we have done very well to maintain the structure of the compiler over time, and the work done for checkpoint 1 and checkpoint 2 remained relatively static. There were only a few instances where this wasn't the case. One example is with the length of arrays. For our checkpoint 2 submission, we allowed any non-negative integer. We realized this causes issues in logic for our checkpoint 3 submission, since for checkpoint 3, we check if the size is 0 to see if it was passed in as a parameter or if it was locally defined. We also had an issue in SemanticAnalyzer.java from our checkpoint 2 submission, where, instead of holding a reference to the relevant declaration in dtype, we only stored type info. For this reason, we created a new parameter in CallExp and VarExp which will always hold a direct reference to the

relevant declaration (FunctionDec, ArrayDec, or SimpleDec). We chose to do this instead of fully modifying our C2 implementation to ensure nothing was broken between checkpoints. Finally, we had one small bug that was missed in C2 where implicit int to bool conversions did not work when passing an int into a bool function parameter.

Design Process

The design process of the entire compiler was a large learning process. There were many steps of the process that were completely foreign to us, as it reached into knowledge beyond that which was taught during our previous low-level programming courses. Firstly, we had a strong foundational understanding of context-free grammars and regex's, but have never implemented them in a way where they match the output of a program. By using these 2 languages in a very unique way, we were able to generate the Abstract Syntax Tree (AST) for the program, and write our first visitor class to display it. We then learned about semantic analysis, and put that to good use to create a .sym file, which showed all of the scopes for variables, and utilized a symbol table to perform semantic analysis and type checking. Finally, for this checkpoint, we implemented a final visitor class which allowed us to generate our output assembly code. For creating this assembly code, we incrementally created instructions. We started with things like the prelude and finale, IntExp, SimpleDec, and FunctionDec, such that we were able to run a very basic void main() function with a single piece of data assignment. From there, we went into some of the longer or more involved visitor classes (other than the function one). This included OpExp, which required mapping out 14(!) different operators to their assembly. Other visitors that were implemented include

CallExp, IfExp, WhileExp, and others. Because there are infinitely many programs, in theory, there are an infinite amount of programs that you need to be able to handle. Because of this, we tried to reduce things down into individual cases. These cases included things like nested loops, recursion and mutual recursion, very long if statements, variable redeclarations, and other cases, which are all represented within our test files.

Troubles Faced

Immediately, one of the things we noticed is that we had not used assembly in a long time. This meant that as we were starting, we were very out of practice for reading, writing, and understanding the assembly code which needed to be generated, making for debugging a slow process until we gained familiarity. Additionally, since almost every program generated with our code resulted in hundreds of lines of assembly code–several of which had loops or recursion causing them to be run many times over–made it very difficult to trace through what was happening in the TMSimulator and find the location where the error occurred.

Furthermore, we found that there was a very large learning curve when it came to converting the output of the semantic analyzer to the code. Wrapping our heads around the fact that, given an AST we needed to generate assembly code which works with an entirely different memory space (the dMem stack), as well as how the needed offsets work, took quite a bit of time. This meant as we each started coding we needed to make sure there was appropriate difficulty “ramp up” to the functionality we were working on, giving us time to adjust to the new way of thinking about our code.

Lessons Learned

As with the other assignments, we have a newfound appreciation for compilers as a whole like GCC. We realized that, even for a language as restricted as CM, coding a simple compiler took a large amount of time. This is also while considering a single assembly language, a single architecture (x32 vs x64 would be a consideration for GNU), and other things that would make a compiler like GCC significantly more complicated. Especially because of the limitations listed below as well, we have a very strong appreciation for optimizations being done in GCC, since you will typically write better assembly code in C than assembly if you are using GCC.

Furthermore, throughout this whole project, the constant consideration of error cases and how to recover from them has given a huge amount of insight and respect for the work done in more strictly typed and error checked languages, like Ada. C- took a relatively limited approach to error handling, recovering enough to continue processing, but not allowing the compiler to continue past the current step. Despite this, the limited error recovery we have done (including popping off the stack until tokens are synched up for recovery, implicitly deciding the results of mismatched types to prevent type errors from propagating to further levels, and runtime errors such as arrays out of bound).

Assumptions and Limitations

Overall, our assignment matches the structure that Prof. Song has requested within the assignment description and within the class slides. Despite this, there are a couple of small assumptions/limitations made based on time constraints. One such

example is the fact that we did not implement short circuiting operations. Short circuit operations would've been hard, as they would require jumps to the end of the statement while ignoring the rest of the checks/statements and not accepting the child part as a whole and instead accepting parts at a time until one comes up that you expect to allow a short circuit (for example, true goes into the block prematurely for or statements, false skips past the block prematurely for and statements). This does lead to some issues with other languages, but is not a large deal for ours. This is because typically short circuiting is done to check for null and then check if it is another value, but since pointers are not part of C-, this does not matter. Despite this, operations like (i != 0 && 100/i < 10) will not work and cause the program to crash, despite the fact that other languages would allow this.

What Would We Do Differently/Possible Improvements

Our code, we believe, does exactly what it set out to do, and does it in a way that makes perfect sense. Despite this, we believe that there is 1 glaring issue with our implementation, and that is how many instructions we have compared to the professor. We do this for 1 very simple reason; to simplify the Java logic for certain expressions. For example, we often preserve the state of the registers, such as in our helper function "checkBounds". Because of this, our program size for GCD.tm seems to be ~1.5x the professor's, but ours works the exact same as his. Despite this, there are definitely areas the stack is written to redundantly, and that is a definite improvement.

One possible improvement for this project as a whole would be a slight modification to the source language (C-) itself. Currently, int to bool conversions are

valid, but bool to int conversions are invalid. Since, other than this, we handle booleans and integers the same way C does (bools being integers of 0 or non 0 for false and true respectively), it makes things very difficult at times when writing code to test in C-, since custom conversion / print functions need to be used to convert between them.

Group Contributions

To begin with, as with the other checkpoints, Amanda did a ton of preliminary research into what each component of the assignment will be, and split it accordingly into 2 coding sections. Amanda did a large amount of the code for this assignment, setting up the visitor class structure, updating SemanticAnalyser to have the new “reference” variable, and implementing around half of the total visitor classes and the related functions for implementing those. She also created the prelude that goes into each program, such as finding main, jumping around i/o code, and more. The visitor classes that were implemented by Amanda include, but are not limited to, the huge OpExp visitor, AssignExp, IntExp, BoolExp, and ExpList. Nicholas then started working, where he did the implementations for arrays and conditional jumps (such as if statements and loops). We both then worked together on finding any bugs or errors in the program, and wrote the testing files together. After that, Nicholas started working on documentation and described the work that was completed. Overall, we both believe that we did our fair share, and communicated incredibly appropriately when we had less or more time. We do not believe that either one of us contributed significantly less than the other.