CIS*4650 ~ Checkpoint 3 Project Report

Isabella McIvor (1101334) , Huda Nadeem (11439431), Zuya Abro (1109843)

What has been done for this checkpoint

➢ Created a new visitor class similar to the ShowTreeVisitor and SemanticAnalyzer classes to perform a post-order traversal of the tree and generate assembly code based on the declarations and expressions in the tree.

➢ Implemented a -c flag to call the assembly code generator.

➢ Created new test files to test the parser, semantic analyzer and the assembly code generator.

Design process

The checkpoint required a lot more preparation than the past two as we had to do a lot of practice and analysis before beginning the design process. This involved going over the lecture slides several times and testing out the TM simulator with many examples to understand how it worked.

The first step of the design process was the setup. This involved creating the CodeGenerator.java class as the visitor class to generate the assembly code. We also had to modify the absyn classes that now required an extra parameter in their accept methods for the boolean flag. In addition, we needed to create instance variables to track a variety of attributes: mainEntry, inputEntry, outputEntry, emitLoc, highEmitLoc, ofpFO, retFO, and initFO.

When setting up, we were not completely sure what each variable was supposed to track. We figured that out as we implemented certain visit methods, but still then had trouble determining when and how to update and access each variable. Had we had redone this checkpoint, a change we might've made is to become more familiar with the variables at the beginning. An instance where it is apparent that this strategy would have been more efficient is when we had a bug in the SimpleVar and ArrayVar code generation. These were supposed to produce different results depending on whether it was on the right or left side of an AssignExp and that information was passed through the boolean flag in the parameter (boolean isAddr). We were unsure why the SimpleVar and ArrayVar visit methods were receiving a "false" result no matter if the AssignExp passed a true or false variable on its end. We discovered it was because the accept methods for each absyn class had hard coded "false" as the argument in the visit methods that it called. When we were setting up the absyn classes with the new boolean flag, we had hard coded them to false because we did not understand how they were supposed to be implemented. This logical error could have been avoided if we had spent more time determining what the boolean variable was meant to do before implementing it.

Another instance where we had some bugs was when calling the emitting routines. When we were not completely sure what the parameters represented, we were getting incorrect results. One bug was evident when generating the finale. In the emitRM_Abs method, we forgot that the implementation shown in class already calculated the distance between the current location and the location to jump to, so we calculated it in the parameter. This caused some incorrect calculations which also could have been avoided by looking over the emitting routines carefully.

In order to keep track of where local variables were located in the frame stack, we implemented a hash table (varTable) similar to one used in the semantic analyzer. This stored the name of the variable as the key and the variable declaration as the value. Since the VarDec objects were updated to have instances of “offset” and “nestLevel”, this simplified the process of finding the location of the variable in the stack, since we could use the lookupVar() method to get the frame offset within the VarExp and IndexExp visit methods. We chose a similar method to be able to keep track of the function locations so that we could easily find the location to jump to for a function call. This hash table (funTable) stored the function name as the key and the number of the line (from emitLoc) to jump to. When declaring a function, we stored this information in the hashtable and in the prelude, we stored the input and output information in the hashtable.

When testing out our AssignExp implementation, we noticed that the frame pointer was off by one or two for the individual variables. This was because we forgot to take into account that the frame offset should be decreased each time a local variable was declared. However, since the variable declarations did not pass the frameOffset parameter to the rest of the body, we could not simply increase the frameOffset parameter within the SimpleDec and ArrayDec visit methods. Our solution to this was to come up with an instance variable decOffset to measure only the amount of local declarations there were. Every time the tree visited a SimpleDec or ArrayDec it would increment the value of this variable. Then, in the CompoundExp visit method, after the decs it would add this offset to the overall frameOffset so that the CompoundExp visit method could pass it to the expressions. After leaving a function we would reassign the decOffset to 0.

Modifications from previous checkpoints

- ➢ In Checkpoint 2, we implemented the code for the global scope directly in CM.java. We changed the SemanticAnalyzer.java to include a wrapper visit method that would handle the global scope tasks to better hide the implementation details from our main class.
- ➢ Semantic analysis enhancements after implementing basic semantic checks, we came across lots of edge cases, with function calls and type checking. Our solution was to refine our visitor classes, adding more detailed checks and improving our handling in Absyn.

Summarizing any lessons gained in the implementation process

- ➢ Breaking down the compiler into smaller, manageable components (e.g., visitor classes) made it easier to handle the larger problem one at a time.
- ➢ Regular testing with simple files and increasingly making them more complex code helped identify bugs early. We learned the value of unit tests for each component and integrating testing for the entire compiler.
- ➢ Using version control (Git) was so helpful for coordinating work for my team members. We learned to make small, incremental changes and consistently merge/push our work to avoid conflicts.
- ➢ We learned how to make clear modular code through the use of object oriented principles such as inheritance, encapsulation and interfaces.

Discussing any assumptions

- ➢ The assembly code does not have to be exactly identical to the examples in the TM Simulator package, as long as it executes the same.
- ➢ While we want our code to detect as many errors as possible, our compiler may not catch every error in source code. Our error messages are there to be helpful and guide the user but may not always pinpoint the exact issue.

Limitations

- ➢ Our compiler supports a simpler version of the C program language. Structures within C like pointers, structs, are not supported, focusing on basic types, arrays, and functions.
- ➢ We are missing runtime errors for scenarios such as accessing an array at an invalid index.
- ➢ Call expressions cannot generate code for parameters.
- ➢ Missing generation for recursion.

Possible improvements

- ➢ Improving the compiler with clearer error messages and a more interactive development environment could significantly improve a user's experience
- ➢ Adding a more extensive C language implementation for structs and pointers, would make our compiler more versatile.
- ➢ Our implementation right now focuses on correctness rather than efficiency. We can work towards more optimizational structure to reduce the generated code's size and execution time.

Contribution/Assessment

Ella:

- ➢ Made modifications from previous checkpoints
- ➢ Implemented assembly generation for VarExps, VarDecs, OpExp, AssignExp, and WhileExp
- ➢ Implemented “-c” flag
- ➢ Contributed to the documentation

Huda:

- ➢ Contributed to WhileExp, CompoundExp, IfExp, and CallExp
- ➢ Contributed to the documentation
- ➢ Contributed to the test files [123457890.cm]
- ➢ Error checking with output

Zuya

- ➢ Contributing to test files
- ➢ Contributed to documentation