CSE 401 Project Report, Autumn 2022
Group BB

Jackson Atkins, Jeffrey Kaufman
jratkins, jeffkauf

December 5th, 2022

What Features Work

Individually, every language feature described in the MiniJava language specification works. This includes arithmetic expressions, control flow (in the form of if/else statements and while loops), object creation, dynamic dispatch, arrays, and inheritance (including method overrides). We are very confident in our scanner, parser, and semantic analysis implementations, especially as we have incorporated feedback on previous parts of the project in the final implementation. And while our code generation does work and allow for the combination of most language features, we are aware that some more complex programs run into problems which we describe next.

What Features Don't Work

We were unable to find any features of the MiniJava specification that do not explicitly work with our compiler. That being said, we are aware of at least one existing issue with the code generation aspect of the compiler. Certain sample programs (namely `LinkedList.java` and `TreeVisitor.java`) encountered errors during execution, which, after verifying the produced ASTs were correct, we were able to narrow down to the generation of the assembly code. While we were unable to determine the exact cause, we are fairly certain that a combination of an incorrect stack alignment and/or the overwriting of registers is to blame. What proved to be most difficult about this issue was how it only seemed to occur in more complex cases—when testing simple programs that only utilized one aspect of the MiniJava language, programs would execute correctly without error. Only when multiple aspects of the language are combined does the compiler seem to have issues.

Test Suite

In order to test each part of the project, we developed tests for each part with the goal of trying to achieve as much coverage as possible. For the scanner, parser, and semantic analysis parts, this consisted of automated testing using the JUnit test framework. To test the code generation, we compared the output the output of programs when compiled with our compiler with the output of the same when the programs were compiled using the Java compiler and executed using the Java Virtual Machine. We utilized both the provided sample programs as well as programs we created to fill any gaps that we felt were left out by the given set of programs.

In writing our unit tests, we focused much of our attention on ensuring edge cases worked as intended. For the scanner, this meant ensuring that the correct tokens were output by the compiler regardless of the of whether the input was syntactically valid. For the parser, we compared the abstract syntax tree (AST) outputted by the compiler with the expected AST. We especially focused on programs that relied on the proper precedence and associativity rules to be successfully parsed, as we determined that was the area of greatest possibility that the compiler may fail due to the ambiguity that can only be solved by the aforementioned rules. Our tests of the semantic analysis portion contain two main parts: one which receives a syntactically correct program and ensures that the outputted symbol table shows the correct fields, methods, and types, and one which receives an assortment of programs that contain the different semantic errors we needed to detect.

Extra Features & Extensions

We did not add any extra features or extensions outside of the given MiniJava specification, nor did we add extra error checking or optimizations (with the one exception that we did ensure we implemented O(1) dispatching using the virtual method table organization presented in lecture).

Division of Work

Work was split relatively evenly between the two of us throughout the duration of the project. As each part of the project was released, we discussed what aspects of the part we felt most comfortable implementing and went from there. We would then each begin implementations on the agreed upon work, but found that during this phase, it was helpful to focus more broadly on the overall structure of the part we were implementing and worrying less about the fine details and trying to debug some of the more complex issues. This helped us to get a good start on each part and to have a general outline of the structures and algorithms we used. Having a general idea of the structures and algorithms we were using helped us when we would later meet up in the Gates Center to sit down and discuss the more stubborn issues we faced. This ability to bounce

ideas off one another worked well in helping solve those harder issues and really dialing our implementation into conforming to the MiniJava specification.

Jackson took the lead in implementation of the scanner and semantic analysis portions of the project, while Jeffrey focused on the implementation of the parser. We both dealt significantly with the implementation of the code generation, especially in the regular debugging that was required during this phase. Jeffrey took the lead on building the test suite used for the scanner, parser, and semantic analysis portions of the project, and Jackson took the primary role in creating the tests for the code generation portion. While we both may have implemented various aspects of the project, in the end, we both feel as though we both took responsibility for the success of the entire project. Even if we didn't implement a certain piece of the puzzle, we both took the time to review the implementation to check for possible errors or oversights by the other partner to ensure that our compiler worked as intended. We both agree that our best work was done when we were able to meet together in the Gates Center as it was the ability to discuss, diagram out ideas on the whiteboard, and ask questions of one another that resulted in the most stubborn of bugs being corrected throughout the project.

Conclusions

Overall, we both agree that we are happy with our work. Although we definitely ran into some stubborn bugs, we were able to overcome the vast majority of them through working together. As we briefly touched upon earlier, we both found that we discovered and fixed the most issues when we were able to meet together in the Gates Center, as not only was having one another there in person made communication easier, it allowed for a more involved development process where we could really ensure that the other partner fully grasped and understood each part of the project. That being said, we obviously wish that we were able to solve the issue with the code generation. Given more time, we are confident that we could complete a version of our compiler that worked with any valid MiniJava program.

Debugging the incredibly complex programs described above when working on the code generation portion of the project proved to be quite frustrating. We both found it difficult to track variable values through the thousands of lines of assembly generated by our implementation. We both think it would have been useful to spend more time in lecture discussing the different ways of implementing data structures and algorithms in assembly, as while CSE 351 did prove helpful to have taken, much of the more complex ideas found in a language (such as control flow and object-oriented programming) like Java proved extremely difficult to implement in assembly.