CSE 401 Project Report, Autumn 2020

Group ID: AX

Names: Julian Gallegos, Karamvir Rai

UWNetIDs: julianga, krai96

- What language features work (arithmetic expressions, if/while, object creation, dynamic dispatch, arrays, etc.).

  Our scanner and parser seem to be fully functioning. Our semantics checker may have some issues with false negatives, but otherwise its structure is implemented. For assembly generation, arithmetic expression, object creation, local and instance variables, vtables (even for derived methods), and if/while conditionals are all working in isolated test cases.

- What language features weren't implemented, don't work, or don't quite work; and briefly what progress you were able to make on these, if any.

  We are not 100% on our implementation for error checking. We had many issues from our initial submission for semantics checking that we had to go through, but we believe we fixed all the issues listed in our last project feedback. Fixing the issues opened up a new issue with our isArithmeticExpression and isBooleanExpression helper methods for checking the type associated with an expression. We realized that our implementations would run checks with these methods under the assumption that the expression actually was an int expression or boolean expression respectively. This made sense to us until we realized we would have to print out errors upon properly confirming the expression type, but many of these errors could only be caught while

running the expression type checker methods, and if there was an error, there was no way to confirm if the expression actually was of the appropriate type. Additionally, if the expression was not actually of a certain type, then any error messages printed out would be garbage information as it wasn't actually an error with the semantics, but an error our code would print out just while checking type. This is not an issue for many parts of our code, like all of the arithmetic ast nodes (e.g. Plus), where it is safe to assume the expression HAS to be an arithmetic expression, so we can output error messages no problem. Where the issue rises is when we have to compare our method return type to the type of the expression being returned. We attempted to fix this by including boolean arguments to the expression type checking methods, but as we were inching closer to the project deadline, we did not carefully test this, and it is possible some false negatives may occur now. Aside from semantics checking, we have many of the assembly generation code working in isolated tests, but when generating code for the sample minijava programs, some of them segfault, and we were not able to find the source of the issue before the deadline. If we had more time, we would look further into how "this" is being passed along by rdi and the stack, as we believe the most likely culprit for the segfault is in the way we handle calls into methods, and perhaps we are losing track of the pointer for making proper calls.

- A summary of the test programs you've tried and the results. Note the word "summary" - a lengthy dump of your test results is not helpful. We want to get an idea of how extensively you tested your code and how you organized the testing.

We don't have any of our test programs saved unfortunately, but we did alter our Test.java many times to fit different specific cases, such as checking that a derived class vtable properly overrides its parent class methods while still adding the other methods it does not derive, all while keeping the order of the methods in the vtable consistent, then expanding test cases to test the same situation with a super class that

extends to another super class. We didn't aggressively test files until we got feedback from our Semantics project, it was embarrassing to see how many points we lost from edge cases that we didn't consider. From there, we would think of cases that might provoke and error in our code, write a simple test case in Test.java, run it, and then evaluate the output. This approach did take us a long way, since it made isolating and testing edge cases that could cause errors easier. If we could go back, we would have been more organized by creating a separate folder which contains all of the test classes. That way we could go back and try a previous test again after making changes to our codebase.

- Any extra features or language extensions supported by your compiler: additional Java language constructs not part of basic MiniJava; extra error or semantics checking; assembly code formatting, comments, or other features; clever code generation or register allocation; optimizations; etc.

  Our compiler does not support extra features or language extensions other than the basic requirements listed on the project specifications. Adding extensions to the project would have been an interesting thing to do if we had more time on our hands.

- How work was divided among team members. If you split up the work, who was responsible for what, or if you worked together on everything, a description of how you organized that and how it went.

  We worked together the whole time through zoom, switching who codes and who oversees/does extra thinking fairly regularly. Whenever we thought up a solution for our work, we made sure to carefully explain our logic to each other, and would only write our code into the project once we agreed upon a solution. Before even finding a solution, we also made good use of the "rubber ducky" approach, where we would explain our thoughts to each other until we came up with a solution. This was very

helpful during our project, plus by working together there was never any fear of someone losing track of what is happening in the project. I think splitting up the work would have been particularly challenging come the semantics portion of the project, where we really had to understand a lot of moving parts in the project. That said, there are places in the project that we may have rushed a solution without organizing it neatly, leaving many parts of the project feeling messier than we would have desired. This might have been avoided if we did not work together the whole time, as we would have likely taken more care in organizing and properly explaining our code through comments.

- Brief conclusions: what was good, what could have been better, what you would have done differently or would have liked to have seen changed about the project.

Given the strange circumstances around having to do this quarter completely online from home, and working together on our homework over zoom, we think we did ok. If we could go back to the beginning of the quarter and do one thing differently, it would certainly be starting these projects a little earlier, but also going into zoom office hours more often whenever we got stuck. If we had done so, we would not have spent so much time trying to make sense of issues we encountered. Furthermore, one thing that we would have done differently if we had the chance to start again would be to make sure that we write neater code. It is obvious when examining our code that there is some redundancy in logic, a lack of documentation, and other issues that make parsing through our codebase difficult. We wrote some private methods on the fly not knowing that we already had functions that did the same thing, so we could have done some refactoring. That was something we noticed in the codegen part of the project, but by then it wouldn't have been a good allocation of time to go back and make things more simple. Overall, the project itself was structured really well and gave us a good understanding of all the components and processes that go into building a

compiler. The project deadlines were fairly spaced out and it never seemed too rushed. Also, given the uniqueness of this quarter, it was nice that we were given extra late days multiple times. This really helped alleviate some stress. Finally, while we did not do a perfect job, we feel that by working together in the way that we did, we were able to learn much from not only the course, but each other, and we are very grateful for this experience.