CSE 401 Project Report, Spring 2023
at
David Stumph and Javier Tomas
dstumph@uw.edu, jtomas21@uw.edu

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

All language features work, but some of them were difficult to get right. Inheritance in particular was tricky, and some of the logical operators were as well.

- 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 implemented all the required parts of the project, and got them all working. We created extensive test cases to verify this, as shown below.

- 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 tested all parts of our compiler, including tests for each part throughout the course.

- ➢ Scanner: We tested the scanner for all legal characters according to the MiniJava grammar. We wrote tests to make sure all types, statements, expressions, and keywords were represented correctly. We also wrote tests to check for unexpected and invalid characters. Finally, we wrote tests to make sure our minijava.jflex file captured comments(multi-line and single line) and delimiters correctly.
- ➢ Parser: When testing the parser, we wrote tests to make sure identifiers were captured correctly, indenting was correct, and line numbers were correct. Our parser tests were able to capture field names and types, parameter names and type, as well as class and method names and return types. We tested for arrays, if statements, while statements, classes, and operators. Our parser tests were there to ensure that the AST representation of the code was correct, and nothing got mixed up.
- ➢ Semantics: The tests for semantics had to cover all possible operations and validate whether it would be a legal expression at that particular part of the code. We checked for type compatibility, whether an identifier was being used in the correct scope, whether an operation could be performed on that identifier (using . notation in a class). Our semantics tests also checked for duplicate identifiers, making sure not to allow two identifiers of the same type to be present in the same scope. Semantics tests also check to make sure that an identifier could only contain data of its type, saving a boolean into an identifier that has type int should not be allowed. We tested for inheritance cycles and

that method overrides were handled properly in MiniJava.java. We also wrote a test to check that subclass assignment was allowed. We also wrote a test to make sure method parameters were not taking a non-existent type (cannot take argument of type Foo if Foo is not a valid class in the program). Upon discovery of an error, our semantics would report the error with a short description along with the line number where the error occurred. On error, MiniJava.java would exit with exit status of 1.

➢ CodeGen: For testing Codegen, we created a bunch of small programs that had just a couple different kinds of operations in them. We created separate tests for arrays, method calls, conditionals, loops, logical operators, and mathematical operators. This gave us a lot of granularity in our testing, and when a test failed, it was fairly easy to narrow down. We also added all the sample programs to our test suite for bigger stress testing, to try and catch any bugs missed by our smaller tests.

- 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.

The only extra feature, if you could even call it that, was making better use of registers in the generated code than what the lecture slides showed. We used the xorq instruction to evaluate !x, for instance, which was made possible by storing all booleans as either 0 or 1. This is unlike C, which accepts any integer as a boolean, and considers 0 to be false and all other values to be true.

- 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 pair-programmed almost the entire project. The only area where we split effort was writing test cases, since quantity was more important than quality in that area. We collaborated entirely over Zoom, since we didn't live near each other. For pair-programming, we each took turns actually writing code, and watching the other person write the code and helping out. It made it go quicker than splitting work would have, since there wasn't really anywhere where more than one thing needed to be done at once.

- 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.

There were a lot of cases we didn't consider during the Semantics assignment, mostly because we didn't have a full understanding of what MiniJava entailed by that point. The grammar didn't contain any information about what valid type checking needed to look like, so we just guessed,

and it didn't turn out that well. A link to relevant parts of the Java specification would have been nice, but we realize that figuring it out was the point of the assignment.

The rest of the project went fairly well. Being able to run programs compiled by the compiler was very satisfying, and the Scanner, Parser, and Codegen phases of the project were more straightforward.

The one thing I'd change about MiniJava would be to have integers be 32-bit, so it would be completely compatible with regular Java. With 64-bit integers, MiniJava will produce different output if overflowing occurs than regular Java would, so we just had to make sure our test cases didn't expose these differences.