CSE 401 Project Report, Spring 2024

Group ID: ae

Andy Stanciu, Victor Cheng

andys22, chenvic

Overall, the MiniJava project went well! We didn't have any moments of major confusion, and the development process was surprisingly pretty linear and bug-free. All language features work, and there do not remain any language features left to be implemented.

We're not sure if we'd have done anything differently had we been able to do the project again. We're satisfied with how everything came together, especially with how we managed to modularize and abstract different aspects of our compiler— the SymbolContext and SymbolTable API, singleton utility classes like Logger and Generator, etc. If there's one thing we're missing, it's more documentation and commenting. Around 50% of our APIs are missing substantive documentation— the only place where we did a pretty good job was the symbol context APIs and a few of the semantics visitors. The only thing we wish could have been changed about the project is the time allocation for the codegen portion. While we believe there was plenty of time to complete code generation, we think it would be nice to allocate a little more time for people to have the chance to take on extensions (small or big). At least for us, it wasn't until the last phase of the project that we fully understood which extra language features are/aren't feasible, so a little bit of extra time (maybe by speeding up the scanner or parser checkpoints) would have been nice.

Andy and I worked exclusively synchronously and in-person. It's very difficult to say who exactly did what, but overall, I (Victor) was responsible for the majority of the semantic visitor logic, and Andy tackled the symbol context/table APIs and a little bit more of the codegen visitor logic. We contributed an approximately equal number of tests, and we had a lot of fun discussing design decisions and catching each other's bugs live.

In our opinion, we developed a pretty extensive set of tests for all phases of the project. In our test/resources directory, we divided our tests into Scanner, Parser, Semantics, and CodeGen subdirectories. Within each, we created numerous test source files that each target a specific language feature or small subset of features. Our naming conventions weren't the most consistent, but we made sure it's all there and every language feature is being accounted for at every stage of the compiler. For our codegen tests, we took a little bit of a different approach (rather than creating individual Junit tests for each source file). Since we were a little bit stubborn and didn't want to work on attu, we wrote a bash script "test_compiler.sh" located in the repository's base working directory. This script sends over all our codegen test source files to attu (as well as our compiler's assembly output). From there, attu compiles them all using gcc and the Java compiler, compares their output, and responds with a report of which tests passed and which ones failed (with a diff if they failed). Kind of extra, but a fun little way to get sidetracked while working on the project…

We ended up implementing a decent bit of extra features in our compiler. None of them are too crazy (we unfortunately ran out of time to get garbage collection and nested scoping complete, so we decided to not merge those changes). Still, we're proud of the extra features we managed to incorporate:

- **Better assignment.** we thought it was a little silly how the MiniJava grammar defined an assignment statement as "*identifier* = expression" (and thought the whole idea of an

"ArrayAssign" nonterminal was a little redundant and clunky. So instead, we generalized assignment to be "*assignable* = expression", where Assignable is an interface that IdentifierExpression, ArrayLookup, and our new Field AST node all implement. This allows us to assign to qualified identifiers, instance variables, more complex array subscript expressions, and in essence, anywhere that actual Java would allow you to assign.

- **All assignment operators: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=.** Once we added one, we reasoned that we might as well just add them all. We created helper methods and generalized the semantics, as well as code generation, of an assignment operator to prevent any of our code from getting redundant or ugly.
- **Post/pre increment/decrement operators: ++i, --i, i++, i--.** In our grammar, these look like "assignable++" or "--assignable", as you should be able to increment/decrement any location where you are allowed to assign. Also, we didn't have enough time to turn these into expressions (or assignment statements either, for that matter), so unfortunately assignment chaining, incrementing/decrementing in an expression, and other fancy ways of using assignment operators in expressions isn't supported.
- **The rest of the operators that exist in Java: /, %, &, |, ^, ~, <<, >>, >>>, <=, >, >=, ||, ==, !=.** Once again, we only added a few at first, but then decided to just go all in. Just like for the assignment operators, we created helper methods to generalize semantics for binary operators, unary operators, bitwise operators, etc. The toughest operators to implement, at least for codegen, were unsurprisingly / and %.
- **Field access.** As we alluded to earlier, we added the Field AST node to allow for fields (instance variables) of objects to be accessed (and assigned to!). This was mostly motivated by us being frustrated that MiniJava did not support a statement as simple as "this.x = 0". Anyways, now we can do stuff like obj.obj2.obj3.a = 1, assuming a is an integer of course. The field AST node's semantics are somewhat similar to a method call's semantics, though are significantly simpler actually.
- **Instanceof operator.** This one was very fun to implement! The code generation made us realize how inefficient instanceof really is (and how it ought to be avoided in code as much as possible). We took advantage of the parent pointers in our vtables to get this working properly, and had some fun writing some recursion in assembly that chased the parent pointer until it was either null or it's address matched the one we're expecting.
- **Ternary expressions.** Kind of random, but we thought it would be cool and seemed easy enough. Semantic validation was a little tricky since for a ternary "(cond) ? e1 : e2", we need to ensure e1 and e2 are both of the same type. The code generation looked pretty similar to an if-else statement with the labels.

That's all! We had a lot of fun implementing our MiniJava compiler and wish we could have spent a whole extra quarter to continue expanding it. Guess we will have the summer to do that…