

Project Report on CSE P 501, Autumn 2009

Mark McWiggins

I implemented a MIPS code generator, targeting the SPIM simulator. (I was excited by how fast SPIM executed the code that I generated until it dawned on me that it wasn't doing any of the checking that slows Java down ...)

All language features work well enough to compile and execute the 8 test programs available on the MiniJava site, as well as a ninth program **foundation.java** that I (belatedly) built to work out some problems I found with the **if** and **while** expression evaluation.

I initially built the method call code to directly call the function (**jal** in MIPS) and only at the end went back and retrofitted in **vtbl** technique, requiring a MIPS **jair** after loading the right address into a register.

I used registers to the maximum extent possible, not realizing until well into the project that this actually is more complex than using just a stack frame! The 10 temporary registers (t0 through t9) were a boon, and my allocation strategy was just to rotate through them and never use any to save anything outside of a single statement. (I may have violated this in one place, but the test programs didn't show this flaw.)

I also built a peephole optimizer and was disappointed to see what little difference it made in the generated code ... but I also was able to do a fair amount of this type of optimization of noticing and avoiding redundant loads/stores/moves on the fly during the code generation. The final result is very little "dumb" code in the optimized versions of the instructions.

The code's weaknesses: I didn't have time to robustify it against more extreme programs (zillions of local variables, long arg lists, etc.) than occur in the MiniJava test suite. In particular, any method call with > 3 arguments will produce incorrect code, as there is no **a4** MIPS register. On the other hand, there are none of these restrictions that couldn't be fixed fairly easily and I can certainly avoid all this in any future compiler projects I am working on.

The semantic-checking pass didn't quite cover everything earlier, but I corrected this during the code generation step out of necessity.

I wrote the code in

```
parser.py
lexer.py
opt.py
```

and all the little scripts that drive the thing

```
ibtc (compiler "itty bitty toy compiler" driver)
rx (regression test run)
rxo (regression test run with optimization)

etc.
```

I got from David Beazley:

```
lex.py  
yacc.py
```

This worked great & met my expectations for the pleasure of doing the lex/yacc stuff in Python instead of using C and C++ with Lex/Flex/Yacc/Bison and others for decades previously.

As for the conversation we had regarding 14 vs. 7 shift/reduce conflicts: we were both right. I did have the precedence for the NOT operator missing, which when added reduced the number of conflicts from 14 to 7. But I fixed a problem with the NOT operator binding in the wrong place of a method call expression not in the grammar but in the building of the syntax tree. I wound up generating a different node with the NOT in the case it wound up where I didn't want it to be, and that worked fine.

I got from Guido Van Rossum:

```
mm.py (O(1) multimethods)
```

This is from his web note '5 minute multimethods' and in fact it took just about that long to get working. I should have looked at this sooner, but I left the semantic checker doing the 'dumb' if/elif checking version of the visitor pattern.

Books (beyond the textbooks) I found especially useful this quarter:

See MIPS Run, by Dominic Sweetman
Python Essential Reference, 3rd Edition, by David Beazley

Finally: I fixed the DOS line-ending problem in the scanner so `\r\n` files should work fine as input.

I hope this covers it. I very much enjoyed the course and I'll look forward to our meeting.