CS 6410: Compilers

Fall 2023

Part 4 – Code Generation
Acknowledgment: Project assignment modified from an assignment developed by Hal Perkins, faculty member of the University of Washington, Allen School of Computer Science and Engineering

Posted on Wednesday, October 4, 2023

Instructor: Tamara Bonaci
Khoury College of Computer Science
Northeastern University – Seattle

Please submit your project by 11:59pm on Sunday, December 3, 2023. You should submit your project by pushing it to your CCS GitHub repository, and providing a suitable tag. See the end of this writeup for details.

1 Assignment Overview

The purpose of this part of the project is twofold:

- Add code generation to the compiler so that it can produce x86-64 assembly code, and
- Add the runtime support needed to execute compiled programs.

We suggest that you use the simple code generation strategy outlined in class to be sure you get running code on time, but you are free to do something different (i.e., better) if you have time. Whatever strategy you use, remember that simple, correct, and working is better than clever, complex, and not done.

We also strongly suggest thorough testing after you implement each part of the code generator. Debugging of code generators can be difficult, and you will make your life easier if you find bugs early, before your generator is too complex. So, in this part of the project you really want to use a test-driven development approach (it has also been effective for some groups in the past) - you potentially want to write tests for particular language features prior to writing the code generation that implements them.

Modify your `MiniJava main` program so that when it is executed using the command

```
1    java MiniJava filename.java
```

with no options, it will read the `MiniJava` program from the named input file, parse it and perform semantics checks, then print on standard output a x86-64 gcc-compatible assembly-language translation of the input program.

If you wish to use a different assembler or target machine for your compiler, please come and talk to me first, so that we can be sure it will be possible to run, and test your compiled code.

If translation is successful, the compiler should terminate with `System.exit(0)`. If any errors are detected in the input program, including static semantics or type-checking errors, the compiler should terminate using `System.exit(1)`. If errors are detected, the compiler does not need to produce any assembly language code.

The `java` command shown above will also need a `-cp` argument or `CLASSPATH` variable as before to locate the compiled .class files and libraries. See the scanner assignment if you need a refresher on the details

Your `MiniJava` compiler should still be able to print out scanner tokens if the `-S` option is used; the `-P` and `-A` options should continue to print the AST; and `-T` should still cause the compiler to print symbol tables with information gathered during the static semantics phase. There is no requirement for how your compiler should behave if more than one of `-A, -P, -S` or `-T` is specified at the same time, or whether your compiler should generate code if one of these options are provided. That is up to you.

As before, if you are using a different implementation language or additional libraries, please be sure that your compiler continues to work as similarly as possible to the specification above, and you must add to your `README` file any new or additional information we need to build, run, and test your compiler.

2 Implementation Strategy

Code generation incorporates many more-or-less independent tasks. One of the first things to do is to figure out what to implement first, what to put off, and how to test your code as you go. The following sections outline one reasonable way to break the job down into smaller parts. We suggest that you tackle the job in roughly this order so you can get something compiled and running quickly, and add to it incrementally until you're done. Your experience implementing the first parts of the code generator also should give you insights that will ease implementation of the rest.

2.1 Integer Expressions & `System.out.println`

Get a main program containing `System.out.println(17)` to run. Then add code generation for basic arithmetic expressions including only integer constants, `+`, `-`, `*` and parentheses. You will also need the basic prologue and return code for the `MiniJava main` method, which uses the x86-64 C language conventions.

2.2 Object Creation and Method Calls

Next, try implementing objects with methods, but without instance variables, method parameters, or local variables. This includes:

- Operator new (i.e., allocate an object with a method table pointer, but no fields)
- Generation of method tables for simple classes that don't extend other classes
- Methods with no parameters or local variables.

Once you've gotten this far, you should be able to run programs that create objects and call their methods. These methods can contain `System.out.println` statements to verify that objects are created, and that evaluation, and printing of arithmetic expressions work in this context.

2.3 Variables, Parameters, & Assignment

Next try adding:

- Integer parameters and variables in methods, including assigning stack frame locations for variables.
- Parameters and variables in expressions
- Assignment statements involving parameters and local variables.

Suggestion: Some of the complexity dealing with methods is handling registers during method calls. It can help to develop and test this incrementally - first a single, simple function argument, then multiple arguments, then arguments that require evaluation of nested method calls.

2.4 Control Flow

This includes:

- `While` loops
- `If` statements
- Boolean expressions, but only in the context of controlling conditional statements and loops.

2.5 Classes and Instance Variables

Add the remaining code for classes that don't extend other classes, including calculating object sizes and assigning offsets to instance variables, and using instance variables in expressions and as the target of assignments. At this point, you should be able to compile and execute substantial programs.

2.6 Extended Classes

The main issue here is generating the right object layouts and method tables for extended classes, including handling method overriding properly. Once you've done that, dynamic dispatching of method calls should work, and you will have almost all of `MiniJava` working.

2.7 Arrays

We suggest that you leave this until late in the project, since you can get most everything else working without arrays.

2.8 The Rest

Whatever is left, including any extensions you've added to the project, and items like storable Boolean values, which are not essential to the rest of the project.

3 C Bootstrap

The easiest way to run the compiled code is to call it from a trivial C program. That ensures that the stack is properly set up when the compiled code begins execution, and provides a convenient place to put other functions that provide an interface between the compiled code and the external world.

We have provided a small bootstrap program, `boot.c`, in the src/runtime directory of the starter code, and we suggest you start with this. Feel free to embellish this code as you wish. In particular, you may find that it is sometimes easier to have your compiler generate code that calls a C runtime function to do something instead of generating the full sequence of instructions directly in the assembly code. You can add such functions to the .c file. Be sure to add any updates to the `src/runtime/boot.c` file. We will use the file found there to run your compiled code.

4 Executing x86-64 Code with gcc

Your compiler should produce output containing x86-64 assembly language code suitable as input to the GNU assembler. You can compile and execute your generated code and the bootstrap program using gcc, and you can use gdb to debug it at the x86-64 instruction level.

There is a sample assembler file `demo.s` in src/runtime that demonstrates the linkage between `boot.c` and assembler code. This demo file does not contain a full `MiniJava` program, and the code produced by your compiler will be different, but it should give you a decent idea of how the setup is designed to work. Use this and `boot.c` as input to gcc to generate an executable demo program. You can also use gcc to generate additional examples of x86-64 assembly code. If foo.c contains C code, gcc -S foo.c will compile it and create a file foo.s with the corresponding x86-64 code.

The output produced by your compiler should compile and run on 64-bit linux-based systems. You should test your compiler by processing several MiniJava programs. By the time you're done, you should be able to compile any of the MiniJava example programs distributed with the starter code. Since every legal MiniJava program is also a legal full Java program, you can compare the behavior of programs compiled and executed by your MiniJava compiler with the results produced when the same program is compiled and executed using javac/java.

5 What to Turn In

As with previous parts of the project, you should include a brief file, called **CODEGEN-NOTES** this time, describing anything unusual about your project, including notes about extensions, clever code generation strategies, or other interesting things in this phase of the compiler. You should give a brief description of how much is working and any major surprises (either good or bad) you encountered along the way. In particular, if this phase of the project required going back and making changes to previously implemented parts, give a brief description of what was done and why it was needed. This file should only discuss this phase of the project.

As before, you will submit this part of the project by pushing code to your GitHub repository. Once you are satisfied that everything is working properly, create a `codegen-final` tag, and push that to the repository.