UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Prof. R. Fateman

Fall, 2005

CS 164 Assignment 6: MJ Code Generation and Competition

Due: Thursday, Nov 24, 2005, 11:59PM

Overall objectives

This assignment requires you to revise your previous project, or start from the translator we supplied to you. Or somewhat less likely start again from the interpreter we supplied to you. You must produce machine language code that will execute MiniJavaprograms. These programs should be executable (that is, compute results, and write output), and you should, for example, be able to run `Factorial.java` or your sort program.

To simplify this task we have supplied you with a virtual machine simulator and an assembler for it in the file `simple-machine.lisp`. It is open source. Feel free to look at it. We have also set up a number of helpful programs which will take care of some of the routine tasks. You are not allowed to modify the machine language of the machine you have in hand in terms of the assignment, but you are welcome to modify it for your debugging purposes. In fact, we think you won't have to do any modifications here, either. We suggest moderation in changing the machine, since we are convinced it is actually not necessary to do so.

The result of the processing should be tantamount to a binary encoding of a machine-language program. You should also be able to display convincing assembly-language code for any program that passes the type-checker, but is otherwise arbitrary MiniJava.

A serious implementation of MiniJavain assembler should likely have two kinds of run-time checks: conformance to array bounds and insuring that you don't try to follow a nil pointer from a variable of a class that hasn't been initialized.

Warning

Writing the code generator should be *less* time-consuming than the typechecker, at least judging by code size.

Some of the code in `simple-compile.fasl` has been re-used from the translator. Recall that the code for assignment 5 (typechecker) was about 650 lines, if you include the environment setup. This code is about 330 lines long. We are supplying you with `short-compile.lisp` which is a nearly-bare file, showing only the principal interfaces.

This project will again require close attention to detail, and you should run as many test cases as you think are necessary to convince us that your program is correct. You are not expected to produce code from an AST that fails to pass the typechecker.

Experience with this class suggests that you may find glitches in our compiler just as for the typechecker and interpreter and possibly in the VM. It would not surprise us if your program is better than ours.

How to approach this task

Starting from the AST, just as you did for typechecking, you can separately code the compilation of each of the kinds of computational expressions you have encountered in typechecking. That is, a central `comp` function can check for the "atomic" cases (e.g. `this` generates an instruction to put this object on the stack.) Then you enter a dispatch to produce sections of code for each MiniJavaconstruct. For example, to compile an `IntegerLiteral 3` you can call generate an instruction to put 3 on the run-time stack.

You will have to generate code for each construction `If, Block, While` etc.

What about Declarations?

One ordinarily doesn't produce any code in compiling type declarations: these are used just to help keep track of information for the rest of the code generation. There is some code generated for function and variable declarations, including the compilation of the initial value expressions for variables, but MJ doesn't have that. The generation of code will proceed from the compilation of function bodies. This code is stored in some other piece of memory, to be called when necessary. Much of the sequence of operations will look just like the typechecking, especially since you must shuffle around information about environments. In effect when you need to get access to a variable, you will look up where it would live at run-time in an environment, and then generate the appropriate reference instructions. You will have to be keenly aware of the difference between lval and rval access to names or values, and how to get and set values in different environments.

Does my code have to be efficient?

NO. It should be correct, and it should be fairly obviously correct, at least when looked at in small pieces.

HOWEVER, we encourage you to look at optimization possibilities and supply another version of your compiler that either produces more compact code, or faster-running code, or perhaps shorter AND faster. The VM instructions may take variable amounts of time for simulations. Details later.

You will receive extra credit for achievements in optimization as well as presenting problematical MJ programs which break other teams' optimizers. If you find bugs in our code, please tell us.

Built-in Functions

If we had many built-in functions, we would have to systematically store these in some "precompiled" form to load up with our compiled program. Since MiniJava is so meager maybe we can just hack up a call to execute `Println` as an assembler op-code.

In order for optimization to make sense we need to have some input method. You will provided such a method for integer input.

Where do I get information on the virtual machine?

There is more information on-line in the source files which completely define the assembler and the VM, will be discussed in section this week as well as in the lecture notes for several upcoming lectures numbers 19 / 20.

What do I turn in?

Expect to turn in one or more files `comp.lisp`, transcripts and tests described in README. If you have compiler optimizations, turn those in too, with further discussion. Keep your eye on the class newsgroup for suggestions.