

Exercise sheet 10 for 23 April 2003

2003-04-09

Do exercises 10.1, 10.2, and 10.3. Hand in the solutions.

Exercise 10.1 Compile the micro-C examples `imp/ex8.c`, `imp/ex13.c`, and `imp/ex20.c` using the `imp/jvmcomp.sml` compiler, and disassemble the resulting class files using `javap -c`.

Compare with the JVM code generated by a Java compiler (such as `javac`) for similar Java-programs.

In most cases you can probably guess what the JVM instructions do. But if you want the precise description of `istore_1`, say, then open <http://java.sun.com/docs/books/vmspec/>, click on *View HTML*, click on Chapter 6 *The Java Virtual Machine Instruction Set*, click on the letter *I* at the top of the page, and then scroll to `istore_<n>`.

Exercise 10.2 The JVM has an increment instruction

```
iinc idx b
```

which increments the local variable with index `idx` by the amount `b`, where `b` must be an integer in the interval $[-128 \dots 127]$. In the SML-JVM Toolkit, this instruction is represented by `Jiinc{var = idx, const = b}`.

Exercises 6.2 and 7.3 introduced the abstract syntax `PostInc acc` and `PostDec acc` for the postincrement `++` and postdecrement `--` operators. These correspond to `b = 1` and `b = ~1` respectively.

Modify function `cExpr` in the `imp/jvmcomp.sml` compiler to implement postincrement and postdecrement for variables `i` (not for array accesses `a[e]`). The `cExpr` function needs to recognize abstract syntax of the form `PostInc(AccVar i)` and `PostDec(AccVar i)`. Remember that in both cases the *old* value of the variable `i` must be left on the evaluation stack, so in addition to the `iinc` an `iload` is needed.

Write, compile, and run a micro-C program to print the numbers from 0 to 20. Disassemble it.

Check whether this micro-C program runs faster than `imp/ex8.c`:

```
void main() {
  int i;
  i = 20000000;
  while (i) {
    i--;
  }
}
```

In a simple-minded JVM-implementation, `i--` will be much faster than `i=i-1`. However, an advanced JVM JIT (just-in-time compiler) such as Sun's Java Hotspot may itself recognize the pattern `i=i-1` and optimize it, in which case there is no need to optimize it in the micro-C to JVM compiler. Partly for this reason, most Java-to-JVM compilers perform only few optimizations.

Exercise 10.3 Add also preincrement and predecrement to the abstract syntax:

```
datatype expr =
  ...
  | PreInc of access (* Java ++i or ++a[e] *)
  | PreDec of access (* Java --i or --a[e] *)
```

Implement preincrement and predecrement for variables `i` (not for array accesses `a[e]`) as in the previous exercise. Remember that in both cases the *new* value of the variable `i` must be left on the evaluation stack.

Exercise 10.4 More cumbersome: Modify the function `cExpr` in the `imp/jvmcomp.sml` compiler to recognize assignments of the form `i=i+b` and implement them using `iinc` when `b` is small enough.

An assignment of the form `x = x + b` can be recognized as abstract syntax of the form `Assign(AccVar x1, Prim2("+", Access(AccVar x2), Cst(CstI b)))` where `x1` and `x2` are the same. The equality of `x1` and `x2` must be tested separately; that cannot be done using SML pattern matching.