CS 577 Homework 1 – Counting Bytecode Frequences – due 2pm, Wednesday, Apr. 17, 2002

Your assignment is to write a program `freq` that records and reports how many times each different bytecode instruction appears in one or more JVM `.class` files.

In more detail, your program should take one or more class file names as command line arguments. For each named class file, consider every method. For each method, consider every instruction. For each instruction, determine the instruction's opcode, and increment a counter corresponding to that opcode. When all files have been processed, output the opcodes that have non-zero counts. For each opcode you put out, print the opcode mnemonic and the corresponding count. Opcodes must be printed in decreasing frequency order. If two or more opcodes appear with the same frequency, they must be printed in increasing order of numeric opcode.

For example, the output for the class file `example.class`, available on the course web page, should be as follows:

```
invokevirtual 8
iconst_0 3
aload 3
aload_2 3
invokespecial 3
arraylength 3
iload 2
iload_3 2
aload_0 2
aaload 2
astore 2
istore_3 2
dup 2
return 2
new 2
ldc 1
aload_1 1
istore 1
astore_1 1
astore_2 1
iadd 1
idiv 1
iinc 1
if_icmplt 1
goto 1
getstatic 1
ifnull 1
```

Note that the `wide` modifier should *not* be treated as a separate instruction; just ignore it for counting purposes.

For extra credit, allow a `.jar` archive file to be specified as an input, and process *all* the class files contained in it.

Incidentally, the point of assigning this problem is just to get you messing around with class files and byte codes, but it does have some practical significance. Java class files can get quite large and therefore expensive to store and slow to transmit, so it is sometimes useful to compress them. One approach to compression is to use frequency information: frequently used opcodes should be encoded in fewer bits than uncommon opcodes. If you're interested in this topic, check out these papers:

Pugh, *Compressing Java Class Files*, PLDI99. `http://www.cs.umd.edu/~pugh/pack.pdf`

Evans and Fraser, *Bytecode Compression via Profiled Grammar Rewriting*, PLDI 2001. `http://research.microsoft.com/~cwfraser/papers/pldi2001.pdf`

Clausen, et al., *Java Bytecode Compression for Low-end Embedded Systems*, TOPLAS 22(3), May 2000. `http://www.daimi.au.dk/~ups/papers/toplas00.pdf`

How to write the program.

There are many legitimate ways to write the program. I'll describe three here.

1. Do it by hand.

The JVM Specification precisely describes the layout of `.class` files. You can write a program (in any language whatever) that reads the `.class` file as a byte stream, locates all the bytecode sequences within it, and counts opcode frequencies from them.

This approach is largely straightforward, and you'll learn a lot about class files, but you'll find it quite tedious. If you pursue this approach, remember that not every byte in a bytecode sequence is an opcode; many opcodes are followed by one or more bytes representing arguments.

2. Use an existing toolkit.

There are a number of freely-available toolkits that support reading, manipulating, and dumping of class files. These toolkits (typically written in Java, naturally) provide APIs for parsing class files into an internal data structure, and then accessing the individual methods, instructions, etc. out of that structure. They also typically contain tables mapping opcodes to their mnemonics, which is very handy for the output stage of your program!

This is probably the approach I'd recommend, if you're comfortable with Java or want to become so, since it exposes you to the internals of class files but does a lot of the dirty work for you. The main challenge with it is finding your way around the rather ill-documented tool APIs. Of the available tools, I would suggest using *Byte Code Engineering Library (BCEL)*. This tool is available from the course web page, together with a file `example.java` that illustrates its use. You can also download it directly from `http://jakarta.apache.org/builds/jakarta-bcel/release/v5.0rc1`. (Note: If you choose to read the manual, you'll find it somewhat out of date. In particular, as the BCEL project recently moved from private hands into the Apache project, all package prefixes `de.fub.bytecode` should be changed to `org.apache.bcel`. Also, the author seems to have confused the words "faculty" and "factorial.")

Another possible toolkit for the job is `soot` (`http://www.sable.mcgill.ca/soot/`),

in particulary the `coffi` package. Unfortunately, not all the functionality you need for this project is publicly exported, so you'll need to do some source hacking (or cut-and-paste from the source). A third possible toolkit is `BITS` (`http://www.cs.ucsb.edu/~ckrintz/tools/BIT.tar.gz`) which I haven't tried out.

3. Use Unix.

A quick and dirty approach is to use the standard Sun utility `javap -c` to dump the class files into a textual format, and then use your favorite Unix tools (`awk` and `sort` come to mind) to process text files. This is undoubtedly the easiest approach, if you know these tools. (Of course you can also write an ad-hoc program – in any language – to process the text files, but it's probably almost as easy to process class files directly.) The only tedious part will be getting the sorting order right within groups of the same frequency. The main disadvantage is that you won't learn much about class files.

How to submit your homework.

Submit the homework *on paper* at the beginning of class on the due date. You should submit:

- Your source files.
- A README and/or `makefile` describing how to compile, build, and run your program, unless this is blindingly obvious.
- Evidence that your program works, in the form of two program runs: (a) on the standard `example.class` file; (b) on two or more additional `.class` files of your choice, processed during a single run.