Program #4 - Assemble this!

Write a Java program to assemble IJVM assembly code into object code

- Due: **Fri Mar 14, 2014**
- Worth: **10 points**

Good luck!

1. Description

Please write a two-pass assembler for the IJVM instruction set. The input to your program is an assembly source file (our homemade brand). The output is an object file.
Read on!

✪ Textbook references

Well, this is a good starting point… the IJVM instruction set.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Page 266 has a nice, tiny IJVM assembly code and object code example. And Chapter 7.3 includes Tanenbaum's pseudo-code for a two-pass assembler. We'll talk about the merits of his approach in class.

✦ ASM file format

The input to your assembler is an assembly source (ASM) file. Our format closely follows the example on Page 266. We need to add/clarify a couple things:

- Let's make the pound sign (#) our comment character. This simplifies our scanner. Anything after the "#" is comment and can be disregarded by your assembler.
- We need one pseudo-instruction, `.method`. This pseudo-instruction will signify the start of a method. It's like .global in Intel assembly. So, each method will start with a .method pseudo-instruction and a label. Like this:

```
.method foo
foo:
```

- Let's restrict our ASM files to one method per file.

☆ Object file format

The output of your program is an object file. We'll make our object file as simple and readable as possible. I propose three parts.

```
<object_file> := <magic_number> <symbols_section> <code_section>
```

1. Magic number

All CSC 220 object files begin with the magic number "`DC`" as the first line. A magic number is a constant that identifies this as a CSC 220 object file. I chose this because `DC` is 220 in hex.

```
<magic_number> := DC
```

2. Symbols section

The Symbols section defines the symbols in this method, one line at a time.

```
<symbols_section> :=
      .symbols
      {<symbol_defn>}
      .end
```

There are three types of symbol definitions: labels, variables, and externals. Each appears one line at a time. The format of each is:

```
<symbol_defn> := <label_defn> | <var_defn> | <extern_defn>
<label_defn> := LAB <label_name> <label_address>
<var_defn> := VAR <variable_name> <variable_num>
```

```
<extern_defn> := EXT,<method_name> <methond_num>
```

In the definitions above, a number or address should be an integer.

3. Code section

The code section lists the object code for the method, one line at a time.

```
<code_section> :=
      .code
      {<object_code>}
      .end
```

Each object code line looks like this:

```
<object_code> := <opcode> <operands>
<opcode> := 1 byte opcode
<operands> := { <operand> }
<operand> := 1 or 2 byte operand
```

Our object code will be in a readable hex format. We'll print the characters for hex digits rather than the hex values themselves, so that we can more easily read the file. I'll talk about this in class and provide a couple methods for you to do this.

♖ Object-oriented design

We'll discuss this in class. What objects are present in this program?

♕ Solo vs. Team project

I invite you to work in pairs on Program #4. If you do this, I ask that only one person works on any given class. For this reason, you'll want to split the two-pass assembler into separate classes.

If you do this program on your own, then you can write an empty symbols section. Just write your assembler and output an empty symbols section, followed by a code section. This will make my disassembler output be a little rough (no names), but that's OK.

♚ Etc

More Program #4 details:

- **Examples** - I'll have examples of ASM and object files available on the k: drive.
- **File names** - Let's use .txt as a suffix for all our files, so that we can notepad them

up. For an ASM file call `X.txt`, let's call the object file `X_obj.txt`.

- **Disassembler** - I will provide a disassembler. This program will do the opposite of your assembler. Its input will be an object file, from which it will create an assembly source file. You can use this tool to test the validity of your program's output.
- **Object-oriented design** - Spaghetti code is not acceptable. Identify the objects in your design and code them up. We'll discuss these objects in lecture.

When we bump into more details, we'll discuss them in lecture.

2. Grading

Please create a **program4** folder in your k: drive space. I'll look for these files:

- Your **README.txt** file where you describe the state of your program. Tell me what examples run and
- Your **Java code**. In your README, please tell me where your code is and what IDE you used (NetBeans, Eclipse, etc) so that I can run it.
- Your results, the **object code** created by your program. In your program4 folder, please create an examples folder. Place the the object code created by your program there.

Your code MUST be beautiful. Ugly code will receive an ugly grade. You can find "Prof Bill's Java Coding Guidelines" on our Program page on the website.

I figure that we'll have issues along the way, so I have a separate page of notes, here: Note35 - Program #4 Notes.

A cautionary note - plagiarism

It's GREAT to get help from me: via email or in person. It's OK to talk to your peers as well. But you know you've crossed the line and cheated when:

- You copy-paste code from someone else
- You don't understand all your code
- You change variables from someone else's existing code

Have fun!
Thanks, Bill