Bytecode interpreter. v0.12

Changes.
v.012 add poke>type> and peek instructions. Put ret and call bytecodes back in. Changes are shown in blue.
v0.11 - remove the ret and call bytecodes. They are redundant because of compiler code generation changes in v0.12 of the compiler.

The bytecode interpreter will read a bytecode file produced by your compiler, and execute it. The interpreter has 6 major data structures:

A runtime stack (rstack): All operations are executed against values in the runtime stack. All variables exist on the runtime stack. To make your life easier, each datum takes exactly one position in the runtime stacks. Since data are of different lengths, this is accomplished by having the stack be represented by a vector, where each element of the vector points to an object holding the actual data item. The object should have a field to hold the value of the data item, and a field to hold the type of the data items. Data items are of type char, short, int or float.

The stack is initially empty.

A runtime stack stack pointer (sp). This points to the top of the runtime stack. Its value should initially be -1.

A stack of frame pointers (fpstack): This is a stack of indices into the start of the frame for a procedure call. It points to the first element in the functions stack frame.

A frame pointer stack stack pointer (fpsp): This points to the top of the runtime stack. Its initial value should be -1.

A program counter (pc). The program counter contains the address of the next instruction to be executed. Its value is initially zero.

Program memory (mem). Program memory holds the program that is read into the interpreter. It is an array of bytes.

The program consists of bytecodes and data that follows certain bytecodes. The first byte of the program contains the initial bytecode in the **main** routine. In general, execution proceeds by executing the byte code pointed to by the PC, updating the PC based on the instruction executing, and continuing until the program executes. The **halt** instruction will terminate the execution of the program.

The instructions to be executed.

comparison bytecodes
cmpe: 132, or 10000100
```
rstack[sp-1] = rstack[sp-1] == rstack[sp]
sp--;
```

cmplt: 136, or 10001000

```
rstack[sp-1] = rstack[sp-1] < rstack[sp]
sp--;
```

cmpgt: 140, or 10001100
```
rstack[sp-1] = rstack[sp-1] > rstack[sp]
sp--;
```

control flow bytecodes
jmp: 36, or 00100100
```
pc = rstack[sp]
sp = sp-1;
```

jmpc: 40, or 00101000
```
if (rstack[sp-1] pc = rstack[sp]
sp = sp-2
```

call: 44, or 00101100
```
fpstack[++fpsp] = sp - rstack[sp]; // subtract off argument stack
                                   // entries
sp--;
pc = rstack[sp--]
```

ret: 48, or 00110000
```
sp = fpstack[fpsp--]
pc = rstack[sp]
```

stack manipulation byte codes
pushc: 68, or 01000100
```
rstack[++sp] = mem[pc+1]
pc += 2;
```

pushs: 69 or 01000101
convert to a short `s = mem[pc+1, mem[pc+2]` (see https://stackoverflow.com/questions/13469681/how-to-convert-4-bytes-array-to-float-in-java)
```
rstack[++sp] = s
pc += 3;
```

pushs: 70 or 01000110
convert to an int `i = mem[pc+1, mem[pc+2], mem[pc+3], mem[pc+4]` (see https://stackoverflow.com/questions/13469681/how-to-convert-4-bytes-array-to-float-in-java)
```
rstack[++sp] = f
pc += 5;
```

pushs: 71 or 01000111
convert to a float `f = mem[pc+1, mem[pc+2], mem[pc+3], mem[pc+4]` (see https://stackoverflow.com/questions/13469681/how-to-convert-4-bytes-array-to-float-in-java)
```
rstack[++sp] = f
pc += 5;
```

pushv: 72, or 01001000
```
rstack[sp] = rstack[rstack[sp]
```

popm: 76, or 01001100
```
sp -= rstack[sp]
```

popv: 80, or 01010000
```
rstack[rstack[sp]] = rstack[sp-1]
sp -= 2
```

peek: 84 or 01011000
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

pokec: 88 or 01100000
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

pokes: 89 or 01100001
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

pokei: 90 or 01100010
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

pokef: 91 or 01100011
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

arithmetic byte codes
add: 100, or 01100100
```
rstack[sp-1] = rstack[sp-1] + rstack[sp]
sp--;
```

sub: 104, or 01101000
```
rstack[sp-1] = rstack[sp-1] - rstack[sp]
sp--;
```

mul: 108, or 01101100
```
rstack[sp-1] = rstack[sp-1] * rstack[sp]
sp--;
```

div: 112, or 01110000
```
rstack[sp-1] = rstack[sp-1] / rstack[sp]
sp--;
```

special op codes

print<type>, 132, or 10000100

```
System.out.println(rstack[sp--]);
```

halt, 0, or 00000000

Terminate the program. Print pc, sp, rstack, fpsp, fpstack. Print empty if a stack is empty.