Bytecode interpreter. v1.05

Changes.
v1.05 Change the generated code for call. Change the generated code for ret. Change the generated code for pushi. Change the generated code for popm. Change the generated code for popv. All changes shown in green.
v1.04 Change print opcodes to start at 144 to be the same as the compiler document. Change pushv psuedo-code to account for fpstack. Change the popv opcode to accout for the fpstack. Changes are shown in orange.
v1.03. Fix typo in prints which said there was a float at the top of the stack. Change is also shown in green.
v1.02. Change first line of the description of pokei, which had a confusing typo. Change definition of pokei which was wrong. Changes are shown in green.
v1.01 added a closing parentheses in the jmpc opcode description.
v1.0. Added the popa opcoded (77). Removed all colors from previous changes. Current changes are shown in blue.
v.015 fix two uses of stack in the description of the frame pointer stack pointer and the description of opcode 44. Replaced "stack stack" in two places with "stack". Changes are shown in orange.
v.014 actually change the bytecode for print to be 148 + type. Changes are shown in green.
v.013 changed names of pushi and pushf to those names from pushs. Expanded the print<type> and pushv<type> opcodes to cover the different types, e.g., printc, printf, etc. Added the *swp* bytecode and its semantics. Changes are shown in green.
v.012 add poke>type> and peek instructions. Put ret and call bytecodes back in.
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 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 pointer (fpsp): This points to the top of the frame 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
meaning: compare the top two elements on the runtime stack and make the new top of the runtime stack 1 if the elements are equal, and 0 otherwise.
```
rstack[sp-1] = rstack[sp-1] == rstack[sp]
sp--;
```

cmplt: 136, or 10001000
meaning: compare the top two elements on the runtime stack and make the new top of the runtime stack 1 if the next to the top element is less than the top element, and 0 otherwise.
```
rstack[sp-1] = rstack[sp-1] < rstack[sp]
sp--;
```

cmpgt: 140, or 10001100
meaning: compare the top two elements on the runtime stack and make the new top of the runtime stack 1 if the next to the top element is greater than the top element, and 0 otherwise.
```
rstack[sp-1] = rstack[sp-1] > rstack[sp]
sp--;
```

control flow bytecodes
jmp: 36, or 00100100
meaning: jump to the location at the top of the runtime stack.
```
pc = rstack[sp]
sp = sp-1;
```

jmpc: 40, or 00101000
meaning: jump to the location at the top of the runtime stack is the next to the top of the runtime stack contains the integer value 1 (true)
```
if (rstack[sp-1]) pc = rstack[sp]
sp = sp-2
```

call: 44, or 00101100
meaning: save the frame stack pointer for the current frame in the fpstack (frame pointer stack). Jump to the location of the function being called, whose address is on the top of the runtime stack.
```
fpstack[++fpsp] = sp – rstack[sp] - 1 // subtract off argument stack
                                   // entries
sp--;
pc = rstack[sp--] //set the PC to the address of the label to be
                  // jumped to
```

ret: 48, or 00110000
meaning: restore the runtime stack pointer of the function being returned to. Set the PC to the value at the top of the runtime stack, which is the address of the instruction following the call or callr statement.
```
sp = fpstack[fpsp--]
pc = rstack[sp--]
```

stack manipulation byte codes

pushc: 68, or 01000100
meaning: push a character literal onto the top of the runtime stack.
```
rstack[++sp] = mem[pc+1]
pc += 2;
```

pushs: 69 or 01000101
meaning: push a short literal onto the top of the runtime stack.
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;
```

pushi: 70 or 01000110
meaning: push an integer literal onto the top of the runtime stack.
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] = i
pc += 5;
```

pushf: 71 or 01000111
meaning: push a float literal onto the top of the runtime stack.
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;
```

pushvc: 72, or 01001000
meaning: push a character variable’s value (where the variable location is at the top of the runtime stack) onto the runtime stack.
```
rstack[sp] = rstack[fpstack[fpsp]+rstack[sp]+1]
```

pushvs: 73, or 01001001
meaning: push a short variable’s value (where the variable location is at the top of the runtime stack) onto the runtime stack.
```
rstack[sp] = rstack[fpstack[fpsp]+rstack[sp]+1]
```

pushvi: 74, or 01001010
meaning: push an integer variable’s value (where the variable location is at the top of the runtime stack) onto the runtime stack.
```
rstack[sp] = rstack[fpstack[fpsp]+rstack[sp]+1]
```

pushvf: 75, or 01001011
meaning: push a floating point variable’s value (where the variable location is at the top of the runtime stack) onto the runtime stack.
```
rstack[sp] = rstack[fpstack[fpsp]+rstack[sp]+1]
```

popm: 76, or 01001100

meaning: pop multiple entries off of the runtime stack, discarding their values. The number of entries to pop is at the top of the runtime stack.
```
sp -= rstack[sp]+1
```

popv: 80, or 01010000
meaning: pop a value off of the runtime stack into a variable. The variable's location is given by the top of the stack, the value popped is the next element into the stack.
```
rstack[fpstack[fpsp]+rstack[sp]+1] = rstack[sp-1]
sp -= 2
```

popa: 77, or 01001101
meaning: pop all of the top entries to frame stack point from the runtime stack but keep val top entries. The number of entries to keep is at the top of the runtime stack.
```
rstack[fpstack[fpsp] + 1] = rstack[sp – rstack[sp]]
rstack[fpstack[fpsp] + 2] = rstack[sp – rstack[sp]+1]
. . .
rstack[fpstack[fpsp] + rstack[sp]] = rstack[sp-1]
sp = fpstack[fpsp]+rstack[sp]
```

peekc: 84 or 01011000
meaning: take the character value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and put it into the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

peeks: 85 or 01011001
meaning: take the short value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and put it into the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

peeki: 86 or 01011010
meaning: take the integer value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and put it into the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

peekf: 87 or 01011011
meaning: take the float value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and put it into the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] + rstack[sp-1]+1] = rstack[fpstack[fpsp]
+rstack[sp]+1]
```

pokec: 88 or 01100000

meaning: change the character value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and to the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] +rstack[sp]+1] = rstack[fpstack[fpsp] +
rstack[sp-1]+1]
```

pokes: 89 or 01100001
meaning: change the short value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and to the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] +rstack[sp]+1] = rstack[fpstack[fpsp] +
rstack[sp-1]+1]
```

pokei: 90 or 01100010
meaning: change the integer value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and to the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] +rstack[sp]+1] = rstack[fpstack[fpsp] +
rstack[sp-1]+1]
```

pokef: 91 or 01100011
meaning: change the float value at the offset (given by the value of the top of the stack element) from the start of the current runtime stack frame and to the variable whose address is given by the next to the top element of the runtime stack.
```
rstack[fpstack[fpsp] +rstack[sp]+1] = rstack[fpstack[fpsp] +
rstack[sp-1]+1]
```

swp: 94 or 01100100
meaning: swap the top of the stack with the next to the top of the stack element.
```
tmp = rstack[sp-1]
rstack[sp-1] = rstack[sp]
rstack[sp] = tmp
```

arithmetic byte codes
meaning: perform the indicated operation on the top two stack elements, pushing the result onto the stack
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
meaning: the print opcodes print the value or variable at the top of the stack. halt terminates the interpreter.
printc, 144, or 10010000. . Print the character at the top of the stack
```
System.out.println(rstack[sp--]);
```

prints, 145, or 10010001. Print the short at the top of the stack
```
System.out.println(rstack[sp--]);
```

printi, 146, or 10010010. Print the integer at the top of the stack
```
System.out.println(rstack[sp--]);
```

printf, 147, or 10010011. Print the float at the top of the stack.
```
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.