Due Date: 29 February, 2016

Problem Description:

Your task is write a two pass assembler/interpreter for a hypothetical stack machine operating on 32 bit integers with the following instructions:

PUSH *v* -- push *v* (an integer constant) on the stack

RVALUE *l* -- push the contents of variable *l*

LVALUE *l* -- push the address of the variable *l*

POP -- throw away the top value on the stack

:= -- the rvalue on top of the stack is place in the lvalue below it and both are popped

COPY -- push a copy of the top value on the stack

ADD -- pop the top two values off the stack, add them, and push the result

SUB -- pop the top two values off the stack, subtract them, and push the result

MPY -- pop the top two values off the stack, multiply them, and push the result

DIV -- pop the top two values off the stack, divide them, and push the result

MOD -- pop the top two values off the stack, compute the modulus, and push the result

OR -- pop the top two values off the stack, compute the logical OR, and push the result

AND -- pop the top two values off the stack, compute the logical AND, and push the result

LABEL *n* -- serves as the target of jumps to *n*; has no other effect

GOTO *n* -- the next instruction is taken from statement with label *n*

GOFALSE *n* -- pop the top value; jump if it is zero

GOTRUE *n* -- pop the top value; jump if it is nonzero  
 PRINT -- pop the top value off the stack and display it as a base 10 integer

HALT -- stop execution

The numeric opcodes are as follows:

HALT 0

PUSH 1

RVALUE 2

LVALUE 3

POP 4

:= 5

COPY 6

ADD 7

SUB 8

MPY 9

DIV 10

MOD 11

OR 12

AND 13

LABEL 14

GOTO 15

GOFALSE 16

GOTRUE 17  
 PRINT 18

All instructions for this machine are 32 bits (4 bytes) long, with the following format: Bits 32-21 are ignored, bits 20-16 hold the opcode, and bits 15-0 hold the operand. (If there is no operand, those bits are filled with zeroes, but otherwise ignored.)

We use the Harvard memory model, with two separate 256KB (65,536 32-bit words) for instructions and data.

In a two-pass assembler, the first pass serves to build the symbol table. To do this, as the assembler progresses through the data and code sections, it increments a location counter which is initialized to 0 at the beginning of the two sections.

To illustrate, consider the following assembly code which is equivalent to the following pseudocode:  
 **IF flag THEN answer := alpha + 2 \* gamma / (C3P0 - R2D2) ENDIF**

**PRINT answer**

**Section .data**

**flag dw 1**

**answer dw 0**

**alpha dw 30**

**gamma dw 18**

**C3PO dw 5**

**R2D2 dw 2**

**Section .code**

**RVALUE flag**

**GOFALSE L0**

**LVALUE answer**

**RVALUE alpha**

**PUSH 2**

**RVALUE gamma**

**MPY**

**RVALUE C3P0**

**RVALUE R2D2**

**SUB**

**DIV**

**ADD**

**:=**

**LABEL L0  
 PRINT answer**

**HALT**

The symbol table would conceptually look something like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Lexeme | Type | Address | | **flag** | **Int** | **0** | | **answer** | **Int** | **4** | | **alpha** | **Int** | **8** | | **gamma** | **Int** | **12** | | **C3P0** | **Int** | **16** | | **R2D2** | **Int** | **20** | | **L0** | **Code** | **52** | |

The second pass then initializes the data in the data section and translates the code section into the appropriate byte codes.

“Meaningful” assembler error messages should be printed; i.e., “Invalid opcode on line xx”, “undefined label”, *etc*.

You may use C, Java, or Python to write your program.

If there are no assembly errors, then the program will interpret the byte codes, giving whatever output is appropriate

2 | 0