This homework is about implementing a **compiler for the calculator language** that we have been studying, including the div and mod extensions (but not the list extensions). The input is source code in the calculator language and the output is the corresponding JCoCo assembly language program.

Please refer to the previous homework for important details on **submission**; **discussion forum**; **elicitation of additional details**; **resources**; **clean, portable Python code**; and **standard input, output, and error streams**; they apply here too. In particular, it is important to ask for clarifications for any unclear details (using the discussion forum).

The **input** consists of the calculator language of calc.py as discussed in class, extended to support the div and mod operators from the previous homework (but not the list operations). The **output** consists of (only) a well-formatted (using the textbook's conventions) JCoCo assembly language program such that, when that program is executed (say, using the *coco* command), it performs the actions specified by the input calculator program. In particular, the output of such an execution (of the output of this homework's compiler) should be exactly equal to the output of the program from the previous homework on the same input.

Unlike the program of the previous homework, this homework's program (the compiler) does not have a uniquely specified correct output for a given input, since there are several assembly language programs that are equivalent (in the above sense) to a given calculator language program.

The following sample input and output illustrates some of these details. (The formatting of the outputs should be improved if possible but the presented form is acceptable.)

Sample input 1:

```
xyzzy = -300000 + 5 * 7 - 3
xyzzy
```

Sample Output 1:

```
Function: main/0
Constants: 300000, 0, 5, 7, 3, None
Locals: xyzzy
Globals: print
BEGIN
  LOAD_CONST  0
  LOAD_CONST  1
  ROT_TWO
  BINARY_SUBTRACT
  LOAD_CONST  2
  LOAD_CONST  3
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  4
  BINARY_SUBTRACT
  STORE_FAST  0
  LOAD_FAST  0
  LOAD_FAST  0
  LOAD_GLOBAL  0
  ROT_TWO
  CALL_FUNCTION  1
  POP_TOP
  LOAD_CONST  5
  RETURN_VALUE
END
```

Sample input 2:

```
tri = 1 + 2 + 3 + 4 + 5
pin = 1 * 2 * 3 * 4 * 5
ssq = 1*1 + 2*2 + 3*3 + 4*4 + 5*5
scb = 1*1*1 + 2*2*2 + 3*3*3 + 4*4*4 + 5*5*5
tri
pin
ssq
scb
```

Sample Output 2:

```
Function: main/0
Constants: 1, 2, 3, 4, 5, None
Locals: tri, pin, ssq, scb
Globals: print
BEGIN
  LOAD_CONST  0
  LOAD_CONST  1
  BINARY_ADD
  LOAD_CONST  2
  BINARY_ADD
  LOAD_CONST  3
  BINARY_ADD
  LOAD_CONST  4
  BINARY_ADD
  STORE_FAST  0
  LOAD_FAST  0
  LOAD_CONST  0
  LOAD_CONST  1
  BINARY_MULTIPLY
  LOAD_CONST  2
  BINARY_MULTIPLY
  LOAD_CONST  3
  BINARY_MULTIPLY
  LOAD_CONST  4
  BINARY_MULTIPLY
  STORE_FAST  1
  LOAD_FAST  1
  LOAD_CONST  0
  LOAD_CONST  0
  BINARY_MULTIPLY
  LOAD_CONST  1
  LOAD_CONST  1
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  2
  LOAD_CONST  2
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  3
  LOAD_CONST  3
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  4
  LOAD_CONST  4
  BINARY_MULTIPLY
  BINARY_ADD
  STORE_FAST  2
  LOAD_FAST  2
  LOAD_CONST  0
  LOAD_CONST  0
  BINARY_MULTIPLY
  LOAD_CONST  0
  BINARY_MULTIPLY
  LOAD_CONST  1
  LOAD_CONST  1
  BINARY_MULTIPLY
  LOAD_CONST  1
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  2
  LOAD_CONST  2
  BINARY_MULTIPLY
  LOAD_CONST  2
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  3
  LOAD_CONST  3
  BINARY_MULTIPLY
  LOAD_CONST  3
  BINARY_MULTIPLY
  BINARY_ADD
  LOAD_CONST  4
  LOAD_CONST  4
  BINARY_MULTIPLY
  LOAD_CONST  4
  BINARY_MULTIPLY
  BINARY_ADD
  STORE_FAST  3
  LOAD_FAST  3
  LOAD_FAST  0
  LOAD_GLOBAL  0
  ROT_TWO
  CALL_FUNCTION  1
  POP_TOP
  LOAD_FAST  1
  LOAD_GLOBAL  0
  ROT_TWO
  CALL_FUNCTION  1
  POP_TOP
  LOAD_FAST  2
  LOAD_GLOBAL  0
  ROT_TWO
  CALL_FUNCTION  1
  POP_TOP
  LOAD_FAST  3
  LOAD_GLOBAL  0
  ROT_TWO
  CALL_FUNCTION  1
  POP_TOP
  LOAD_CONST  5
  RETURN_VALUE
END
```