Name: _____

Solutions

1. (1 pt.)
   - **Read all material carefully.**
   - *If in doubt whether something is allowed, ask, don't assume.*
   - You may refer to your books, papers, and notes during this test.
   - E-books may be used.
   - Computers are permitted but discouraged.
   - **Electronic and network resources must only be used as a passive library.**
   - Write, and draw, carefully. Ambiguous or cryptic answers receive zero credit.
   - Use class and textbook conventions for notation, algorithmic options, etc.

   Write your name in the space provided above.
   Do not write anything else on this page.

2. (14 pts.) Consider the JCoCo assembly language program listed below.

   (a) (7 pts.) **Explain** what the program does as precisely as possible. (Recall recent classroom discussion of similar questions.) In particular, describe its output as a function of its input.

   (b) (7 pts.) Provide a **complete JCoCo assembly language program** that exhibits the same input-output behavior as this one but whose code is shorter by at least one instruction, or explain why no such shorter program is possible. If your shorter program reuses parts of this program then you may indicate so instead of rewriting those parts **but only if** the result is completely obvious and unambiguous. **Explain why your answer is correct.**

```
Function: main/0
Constants: ' ? '
Locals: x
Globals: print, input, split, len
BEGIN                           # operand stack states below, TOS is leftmost.
        LOAD_GLOBAL 1           # [input]
        LOAD_CONST 0            # [' ? ', input]
        CALL_FUNCTION 1         # ["he lo wrld"] # assuming I = "he lo wrld" on stdin
        DUP_TOP                 # ["he lo wrld", I]
        LOAD_ATTR 2             # ["he lo wrld".split, I]
        CALL_FUNCTION 0         # [["he", "lo", "wrld"], I] Let J = ["he", "lo", "wrld"]
        DUP_TOP                 # [J, J, I]
        SETUP_LOOP label2       # no change to op stack but block stack set up
```

```
        GET_ITER                # [iter(J), J, I]
label0: FOR_ITER label1         # ["he", iter(["lo", "wrld"]), J, I]
        LOAD_CONST 0            # [' ? ', "he", iter(["lo", "wrld"]), J, I]
        BUILD_LIST 2            # [["he", ' ? '], iter(["lo", "wrld"]), J, I]
        LOAD_GLOBAL 0           # [print, ["he", ' ? '], iter(["lo", "wrld"]), J, I]
        ROT_TWO                 # [["he", ' ? '], print, iter(["lo", "wrld"]), J, I]
        CALL_FUNCTION 1         # [None, iter(["lo", "wrld"]), J, I]
        POP_TOP                 # [iter(["lo", "wrld"]), J, I]
        JUMP_ABSOLUTE label0    # other iterations similar until iter is exhausted, then...
label1: POP_BLOCK               # block stack popped, op stack: [J, I]
label2: RETURN_VALUE            # J = ["he", "lo", "wrld"] returned from main
END
```

Ⓐ *The code has been commented to depict the state of the operand stack and a few other details that explain the effect of each instruction, assuming (for illustration) that the string* `he lo wrld` *followed by a newline is provided on standard input. (The comments abbreviate it to* `I` *in some cases.) The program prints the string* `' ?  '` *to standard output and waits for input on standard input. Assuming the sample input described above, the program then prints three lines to standard output:*

```
['he', ' ?  ']
['lo', ' ?  ']
['wrld', ' ?  ']
```

In general, the program prints the initial prompt string `' ?  '` *and waits for input terminated by a newline on standard input. It splits the read string into tokens separated by on white-space (e.g., tokens* `he`*,* `lo`*,* `wrld`*). The resulting list is called* `J` *in the comments. It then creates a two-element list for each token with the string* `' ? '` *as the second item of that list (and the token as the first item), and prints that list followed by a newline to standard output.*

The program may be shortened in several ways. One simple way is deleting the first `DUP_TOP` *instruction (with no other changes). The resulting program will operate in a manner very similar to above but the operand stack will not contain the rightmost* `"he lo wrld"` *string. The lack of this string at the bottom of the stack does not change the input-output behavior of the program because that string is never accessed by the original program and remains on the operand stack of that program just before it terminates (and so is useless).*

3. (15 pts.) Provide a **complete JCoCo assembly language program** that

   (a) Reads two newline-terminated string from *standard input* (one string per line).

   (b) Writes a single integer n followed by a newline to *standard output*, where n is the product of the lengths (in characters) of the two input strings.

   Explain why your program is correct.

   Ⓐ *[There are several correct answers.] The code appears below, with material after a* `#` *on each line representing a comment for a human (not part of the program). Within that comment, the state of the operand stack is depicted before the second* `#`*, as a list*

with the top-of-stack leftmost. The general plan of action of the program is to loads the print function in preparation for the final printing. Then it reads the first line from stdin and compute its length (leaving it on the stack). This sequence of instructions is repeated to read the second line and compute the length, also leaving it on the stack. The two lengths on the stack are next multiplied to give the integer to be printed. The comments assume that the input lines are `foo` *and* `barbaz` *for illustration.*

```
Function: main/0
Constants: ""
Globals: print, input, len
BEGIN
  LOAD_GLOBAL      0  # [print] #  (0) Pushes print fn for use at the end (1).
  LOAD_GLOBAL      1  # [input, print] # (2) Pushes input function.
  LOAD_CONST       0  # ["", input, print] # (3) Pushes arg for input function.
  CALL_FUNCTION    1  # ["foo", print] # (4) Invokes input("") to read string from stdin.
  LOAD_GLOBAL      2  # [len, "foo", print] # (5) Pushes the len function.
  ROT_TWO             # ["foo", len, print] # (6) Puts len's arg is above it.
  CALL_FUNCTION    1  # [3, print] # (7) Invokes len("foo")
  LOAD_GLOBAL      1  # [input, 3, print] # Similar to (2).
  LOAD_CONST       0  # ["", input, 3, print] # Similar to (3).
  CALL_FUNCTION    1  # ["barbaz", 3, print] # Similar to (4).
  LOAD_GLOBAL      2  # [len, "barbaz", input, 3, print] # Similar to (5).
  ROT_TWO             # ["barbaz", len, 3, print] # Similar to (6).
  CALL_FUNCTION    1  # [4, 3, print] # Similar to (7).
  BINARY_MULTIPLY     # [12, print] # args to multiply are TOS and TOS1.
  CALL_FUNCTION    1  # [None] # Print function loaded at (0) called with above as arg.
  RETURN_VALUE        # [] # Above None returned from main.
END
```

4. (6 pts.) For each of the following *Standard ML* expressions, provide the response when that expression is evaluated by the `sml` REPL (read-eval-print loop). Assume that the expressions are evaluated in the order listed. In your response, *draw a* ***box*** *around the* ***type*** *and* `oval` *around the* ***value****.* (If there is an error then clearly explain the error.)

   (a) (2 pts.) `42.42 / 2.0;` Ⓐ `val it =` *21.0* `:` `real`

   (b) (2 pts.) `42 / 2;` Ⓐ *Error because operator* `/` *requires* `real` *operands but the ones in the expression are* `int`*. [However, for grading, we will accept 21, int or 21.0, real as answers for this exam (only!) since this topic was only lightly covered.]*

   (c) (2 pts.) `42 / 2.0;` Ⓐ *Error because operator* `/` *requires* `real` *operands but the numerator in the expression is* `int`*.*

   Ⓐ

5. (8 pts.) For each of the following SML expressions: (1) **State** its type and (2) **explain** how that type is inferred (using the recent classroom discussion type inference as a model).

   (a) (4 pts.)
   `fun f101(x) = x + 101;` Ⓐ *The type is (a function)* `int -> int`*. Inference:*

101 is an int, so x must also be an int in order for x + 101 to be valid. So the domain of the function is int. The result of x + 101 is also an int and so the codomain of the function is also an int.

(b) (4 pts.)

```
fun f301 (i, j) =
    if i < j then
        2 * i + j + 1
    else
        j * f301(i - 1, j + 1);
```

Ⓐ *The type is (a function)* `int * int -> int`*. [For this exam we will also accept* `int, int -> int`*.] Inference: In the true arm of the if expression, the function returns the result of an arithetic expression with 2 and 1, which are ints, and i and j, which must therefore also be ints. So the domain of the function (whose arguments are i and j) is* `int * int`*. Similarly, in that case, the result is also an int, so the codomain of the function is* `int` *(in all cases, because it cannot change based on the predicate of the if expression).*