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 *subject to the restrictions* noted in class.
   - **Computers** (including smart phones, tablets, etc.) **are not permitted**, except when used strictly as e-books or for viewing ones own notes.
   - **Network access** of any kind (cell, voice, text, data, ...) is **not permitted**.
   - 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.

2. (19 pts.) Consider the *JCoCo* virtual machine running the following JCoCo assembly language program. Depict the state of the operand stack after each instruction. State any output produced by the program. *Provide brief explanations to qualify for better partial credit.* Reminder: Use of computers is not permitted. Running the program using *coco* or similar is not allowed.

```
Function: main/0
Constants: 2, 3, 7, None
Locals: my, name, is
Globals: print
BEGIN
  LOAD_CONST   0              [2]
  STORE_FAST   0              []  my=2
  LOAD_FAST   0               [2]
  LOAD_FAST   0               [2, 2]
  LOAD_FAST   0               [2, 2, 2]
  BINARY_MULTIPLY             [4, 2]
  STORE_FAST   1              [2] name=4
  LOAD_FAST   1               [4, 2]
  LOAD_FAST   1               [4, 4, 2]
  LOAD_CONST   1              [3, 4, 4, 2]
  BINARY_MODULO               [1, 4, 2]
  LOAD_CONST   2              [7, 1, 4, 2]
  BINARY_SUBTRACT             [-6, 4, 2]
  STORE_FAST   2              [4, 2] is=-6
  LOAD_FAST   2               [-6, 4, 2]
  LOAD_FAST   0               [2, -6, 4, 2]
  LOAD_FAST   1               [4, 2, -6, 4, 2]
```

```
  BINARY_TRUE_DIVIDE          [0.5, -6, 4, 2]
  LOAD_FAST  2                [-6, 0.5, 7, 4, 2]
  BINARY_MULTIPLY             [-3.0, -6, 4, 2]
  LOAD_GLOBAL  0              [print, -3.0, -6, 4, 2]
  ROT_TWO                     [-3.0, print, -6, 4, 2]
  CALL_FUNCTION  1            [None, -6, 4, 2]
  POP_TOP                     [-6, 4, 2]
  LOAD_CONST  3               [None, -6, 4, 2]
  RETURN_VALUE
END
```

Ⓐ *The program prints "-3.0" as output. The column on the right above depicts the stack as a list with the top of stack leftmost.*

3. (15 pts.) Provide a complete JCoCo assembly language program that reads an integer n from *standard input* and prints the value of n^4 (the input number raised to the fourth power) on standard output. *Explain why your program is correct.*

   Ⓐ *The first three instructions push three built-in functions onto the stack, in reverse order of their later invocations: print, int, and input. The string "n: " as argument for the input function is pushed next, followed by calling that function. This results in the string read from stdin to appear on the top of the stack, replacing the earlier top two entries. The int function is called next with this string, giving an integer value on the top of the stack, replacing the previous int function object and the string argument. The exponentiation computation is performed next, after loading the constant 4, giving n^4 on the top of the stack. Finally, the print function is invoked (which now appears just below n^4 in the stack. [Grading note: There are several other correct answers. No points are lost if the input function is invoked with 0 arguments (no prompt; supported by Python but not by JCoCo). Only a small points deduction for missing the int function since that is a detail that is easy to miss.]*

```
      Function: main/0
      Constants: "n: ", 4
      Globals: input, int, print
      BEGIN
        LOAD_GLOBAL     2
        LOAD_GLOBAL     1
        LOAD_GLOBAL     0
        LOAD_CONST      0
        CALL_FUNCTION   1
        CALL_FUNCTION   1
        LOAD_CONST      1
        BINARY_POWER
        CALL_FUNCTION   1
        RETURN_VALUE
      END
```

4. (15 pts.) Provide a context-free grammar (CFG) for parsing lists in the format used by Homework HW01. The grammar should be designed to accept a single list (of arbitrary length) in that format, and nothing else. Be sure to use exactly the format specified in the homework. *Explain why your CFG is correct* (why it accepts all valid lists and also why it accepts nothing else).

   Ⓐ*[There are several correct answers.] In the following grammar, the nonterminal* `expr` *denotes expressions as defined by the basic calculator. So, in particular, it implies a rule* `expr` → `NUMBER`.

$$
\begin{aligned}
\texttt{list} &\rightarrow \texttt{( )} \\
\texttt{list} &\rightarrow \texttt{(expr ,)} \\
\texttt{list} &\rightarrow \texttt{(expr , expr exprsfx)} \\
\textit{exprsfx} &\rightarrow \epsilon \\
\textit{exprsfx} &\rightarrow \texttt{,} \\
\textit{exprsfx} &\rightarrow \texttt{, expr exprsfx}
\end{aligned}
$$

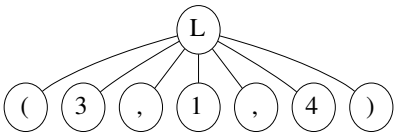
5. (15 pts.) Provide (1) **leftmost derivations**, (2) **parse trees**, and (3) **abstract syntax trees** for the following input (sentence) using the grammar of Question 4:

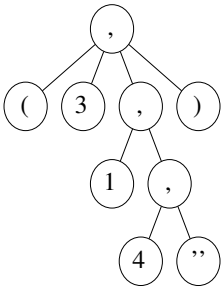
   `(3, 1, 4)`

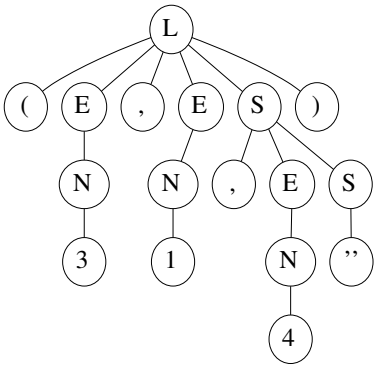
   Ⓐ*The following uses abbreviations* L, E, S, *and* N *for* `list`, *`expr`*, *`exprsfx`*, *and* *`NUMBER`*, *respectively.*

$$
\begin{aligned}
\underline{L} &\Rightarrow \texttt{(}\ \underline{E}\ \texttt{,}\ E\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ \underline{N}\ \texttt{,}\ E\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ \underline{E}\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ \underline{N}\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ 1\ \underline{S}\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ 1\ \texttt{,}\ \underline{E}\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ 1\ \texttt{,}\ \underline{N}\ S\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ 1\ \texttt{,}\ 4\ \underline{S}\ \texttt{)} \\
&\Rightarrow \texttt{(}\ 3\ \texttt{,}\ 1\ \texttt{,}\ 4\ \texttt{)}
\end{aligned}
$$

   The leftmost tree below is the parse tree corrsponding to the above derivation. There are multiple correct answers for the abstract syntax tree based on how a list is defined/interpreted by the language being parsed. Two are depicted here to the right of the parse tree. The first is based on a flat interpretation of lists while the second uses a sequence of Lisp-like cons operators (corresponding to commas in Python).


L
(
3
,
1
,
4
)


,
(
3
,
)
1
,
4
''


L
(
E
,
E
S
)
N
N
,
E
S
3
1
N
''
4