Use only one side of the paper and start each problem on a new page!!
Please show all work to receive ANY credit!!!
1. (10 points) Context-Sensitive Grammars and Derivations

In a CFG, the left-hand-side of a production rule must be a non-terminal. However, in a context-sensitive grammar (CSG), the left-hand-side of each production rule can be a sequence of one or more non-terminals (at least one) and terminals (zero or more) as shown below:

\[
\begin{align*}
S & \rightarrow x \ S \ B \ C \\
C \ B & \rightarrow B \ C \\
y \ C & \rightarrow y \ z \\
S & \rightarrow x \ y \ C \\
y \ B & \rightarrow y \ y \\
z \ C & \rightarrow z \ z
\end{align*}
\]

Note that \(x\), \(y\), and \(z\) are terminals, and \(S\), \(A\), \(B\), and \(C\) are non-terminals. In a CSG, during a single step of the derivation, two symbols (say \(y \ B\)) can be replaced by two symbols (say \(y \ y\)) for the production rule \(y \ B \rightarrow y \ y\), which is unlike a CFG where one symbol (always a non-terminal) can be replaced by zero or more non-terminals or terminals. This is shown in the step of the derivation below:

\[
\alpha \ y \ B \ \beta \Rightarrow \alpha \ y \ y \ \beta
\]

where \(\alpha\) and \(\beta\) are strings of terminals and non-terminals. Using this grammar, develop a derivation for the string: \(xxxyyyzzz\).
2. (10 points) Regular Expressions, Languages, and CFGs

For parts a and b., you may only use the operators: +, *, |, ?, (), and concatenation to construct regular expressions.

(a) (5 points) Write a regular expression for the language of all strings of \{0, 1\} with either an even number of zeros (at least two) or an odd number of ones (at least one).

(b) (5 points) Describe using one or two prose sentences, the language that is represented by the regular expression:

\[(s? (he)) \mid (h ((i(s|m)) \mid (er(s?))))\]
3. (15 points) CFGs, Left Recursion, and $\epsilon$ Moves

As the first step in the design of a top-down parser, reformulate the grammar given below into an equivalent CFG that is suitable for top-down parsing by first removing any left recursion that exists, and then removing all $\epsilon$ moves.

$$
A_1 \rightarrow x \ A_2 \\
A_1 \rightarrow A_3 \ y \\
A_2 \rightarrow A_3 \ x \\
A_2 \rightarrow A_1 \ y \\
A_3 \rightarrow A_1 \ x \\
A_3 \rightarrow A_2 \ y \\
A_3 \rightarrow z
$$

In the grammar, $x$, $y$, and $z$ are terminals, and $A_1$, $A_2$, and $A_3$ are non-terminals. Make sure that you clearly indicate your results separately, i.e., box the CFG without left recursion with $\epsilon$ moves, and box the result after $\epsilon$ moves have been eliminated.
4. (15 points) FIRST and FOLLOW for CFGs

Consider the grammar given below, with $S$ as the start symbol, $X$, $Y$, and $Z$, as non-terminals, $a$, $b$, $c$, $d$, $f$, and $g$ as terminals, and $\epsilon$ as the empty symbol.

$$
S \rightarrow X \ Y \ Z \\
X \rightarrow a \ | \ Z \ b \ | \ \epsilon \\
Y \rightarrow c \ | \ d \ X \ Y \ | \ \epsilon \\
Z \rightarrow f \ | \ g
$$

(a) (10 points) Compute FIRST and FOLLOW for each non-terminal in the grammar.
(b) **(5 points)** In calculating FIRST and FOLLOW, we have been discussing LL(1) grammars. The number ‘1’ indicates the lookahead, and in this case, means that we use the current input symbol as the lookahead in conjunction with the top element of the parsing stack to determine the next parsing action (via the parsing table). If we wanted to develop an LL(2) grammar, we would use both the current input symbol and the next input symbol as lookaheads. In this case, when calculating first, instead of determining a set of single terminals, we would have to find possibly pairs of terminals. Thus, define FIRST2(A) for the non-terminal A to be the first, zero, one, or two terminals that A turns into during a derivation. For example, given the grammar above,

$$\text{FIRST2}(X) = \{ a, fb, gb, \epsilon \}$$

where fb and gb are in FIRST2(X) as the result of the rule $X \rightarrow Z b$. Using the calculation techniques for FIRST and this example, determine: FIRST2(Y) and FIRST2(S) for the same grammar below:

S $\rightarrow$ X Y Z  
X $\rightarrow$ a | Z b | $\epsilon$
Y $\rightarrow$ c | d X Y | $\epsilon$
Z $\rightarrow$ f | g
5. **(20 points) Design of a Context Free Grammar**

In the ML functional programming language, a list is a sequence of one or more list elements (which may also be themselves lists), that are enclosed in square brackets with elements separated by commas. For example:

```ml
val L = [1, 2, 3];       -- an integer list with three elements
val N = ["one", "two"];
val P = [1.5];           -- a real list with one element
val Q = [[4,5,6], [7,8,9]];
```

Lists are accessed in ML by applying the function head (hd) to return the first list element and tail (tl) to return a list containing all elements except the first one, e.g.,

```ml
hd(L);                   -- returns the integer 1
tl(L);                   -- returns the list [2, 3]
hd(P);                   -- returns the real 1.5
tl(P);                   -- returns the null list []
hd(tl(L));               -- returns the integer 2 -- note that combinations are possible
tl(tl(tl(L)));           -- returns the list [3] -- applied in a nested fashion
```

Assume that tokens have been defined for: INT, REAL, STR, IDENT, HD, TL, VAL.

(a) **(20 points) Design a CFG for ML programs that contain one or more val list declarations followed by one or more statements to access lists and/or apply functions.** To get you started, the first few grammar rules are given below, where val declarations refers to the val statements above, and statement_list refers to the application of hd, tl, etc., to declared list variables.

```ml
ml_program      -->  val_declarations statement_list
val_declarations -->  val_declarations list_declaration | list_declarations
list_declaration -->  ???
statement_list  -->  ???
```
Additional space for part 5a.
(b) (5 points) Demonstrate that your CFG works by deriving the input:

```scala
val L = [[1], [2]];
tl(L);
hd(tl(L));
```