1. Give the context-free grammar (CFG) for each of the following languages:

   (a) The set of all strings over the alphabet \{2, -, +\} representing valid arithmetic expressions where each integer in the expression is a single digit and the expression evaluates to some value \(\geq 0\).

   Example Strings in the Language:
   
   \[
   2+2 \quad 2-2+2 \quad -2-2+2+2
   \]

   Strings not in the Language:
   
   \[
   -2 \quad -2+22 \quad 2++2-2
   \]

   \textbf{Answer:}

   \[
   S \rightarrow 2E \mid -2E + 2E \mid 2E - 2E \\
   E \rightarrow -2E + 2E \mid +2E \mid +2E - 2E \mid \epsilon
   \]
(b) The set of all strings over the alphabet \{string, ,%s,arg,\,\} representing valid arguments to the c printf() function. For the purposes of this problem, treat \textit{string} and \textit{arg} as tokens of your language where \textit{string} represents an arbitrary sequence of legal characters in a c string and \textit{arg} represents any arbitrary char* used as the location for a %s. \textbf{Note} that ‘,’ and ‘ ’ are in the alphabet.

Example Strings in the Language (surrounded by “):

“"
"string %s%s string", arg, arg"

"%s", arg"

Strings not in the Language (surrounded by “):

"arg"string"
"arg string"

"string %s, arg"

\textbf{Answer:} First, let’s denote ‘ ’ as a white space.

\[
S \rightarrow "B \\
B \rightarrow " | S, %SSrBA \\
A \rightarrow W, WargW \\
W \rightarrow i | W, i | \epsilon \\
S_r \rightarrow \epsilon | S, stringS_r | C \\
C \rightarrow i | , | CC | \epsilon 
\]
(c) The set of all strings over the alphabet \{0, 1\} in the language \( L : \{0^i1^j0^k \mid j \leq i + k\} \).

Example Strings in the Language:

\[
\begin{array}{ccc}
00000 & 000111100 & \varepsilon \\
\end{array}
\]

Strings not in the Language:

\[
\begin{array}{ccc}
1 & 000111101 \\
\end{array}
\]

Answer:

\[
S \rightarrow TU \\
T \rightarrow ITJ \mid \varepsilon \\
U \rightarrow JUK \mid \varepsilon \\
J \rightarrow 1 \mid \varepsilon \\
I \rightarrow I0 \mid 0 \\
K \rightarrow 0K \mid 0
\]
(d) The set of all strings over the alphabet \{[\}, {\}, ,\} which are sets. We define a set to be a collection of zero or more comma-separated arrays enclosed in an open brace and a close brace. Similarly, we define an array to be a collection of zero or more comma-separated sets enclosed in an open bracket and a close bracket. Note that "," is in the alphabet. Example Arrays:

\[
\{\}, \{\}\]
\[
\[
\]
\[
\{\}, \{\}\]

Example Sets:

\[
\{\}\]
\[
\{}, \{\}, []
\]

Example Strings in the Language:

\[
\{\}, \{\}, \{\}, \{\}, []
\]

Strings not in the Language:

\[
\[
\], \{\}, []
\]

Answer:

\[
S \rightarrow \{T\} | \{}
\]
\[
T \rightarrow U, T | U
\]
\[
U \rightarrow [V] | []
\]
\[
V \rightarrow S, V | S
\]
2. (a) Left factor the following grammar:

\[
S \rightarrow I \mid I - J \mid I + K \\
I \rightarrow (J - K) \mid (J) \\
J \rightarrow K1 \mid K2 \\
K \rightarrow K3 \mid \epsilon
\]

Answer:

\[
S \rightarrow IV \\
V \rightarrow \epsilon \mid -J \mid +K \\
I \rightarrow (JW \\
W \rightarrow -K) \mid ) \\
J \rightarrow KZ \\
Z \rightarrow 1 \mid 2 \\
K \rightarrow K3 \mid \epsilon
\]
(b) Eliminate left recursion from the following grammar:

\[
S \rightarrow STS \mid ST \mid T \\
T \rightarrow Ta \mid Tb \mid U \\
U \rightarrow T \mid c
\]

**Answer:**

\[
S \rightarrow TS' \\
S' \rightarrow TSS' \mid TS' \mid \epsilon \\
T \rightarrow cT' \\
T' \rightarrow aT' \mid bT' \mid \epsilon
\]
3. Consider the following CFG, where the set of terminals is \( \Sigma = \{a, b, \#, \%, !\} \):

\[
\begin{align*}
S & \rightarrow \%aT \mid U! \\
T & \rightarrow aS \mid baT \mid \epsilon \\
U & \rightarrow \#aTU \mid \epsilon
\end{align*}
\]

(a) Construct the FIRST sets for each of the nonterminals.

**Answer:**
- \( S \): \{\%, \#, \!\}
- \( T \): \{\(a, b, \epsilon\}\}
- \( U \): \{\#, \epsilon\}
(b) Construct the FOLLOW sets for each of the nonterminals.

**Answer:**

- **S:** \{#, !, $\}
- **T:** \{#, !, $\}
- **U:** \{!\}
(c) Construct the LL(1) parsing table for the grammar.

**Answer:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>#</th>
<th>%</th>
<th>!</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>S → U!</td>
<td>S → %aT</td>
<td>S → U!</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → aS</td>
<td>T → baT</td>
<td>T → ε</td>
<td>T → ε</td>
<td>T → ε</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td>U → #aTU</td>
<td></td>
<td>U → ε</td>
<td></td>
</tr>
</tbody>
</table>
(d) Show the sequence of stack, input and action configurations that occur during an LL(1) parse of the string “#abaa%aba!”. At the beginning of the parse, the stack should contain a single $S$.

**Answer:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>#abaa%aba!$</td>
<td>output $S \rightarrow U!$</td>
</tr>
<tr>
<td>$U$</td>
<td>#abaa%aba!$</td>
<td>output $U \rightarrow #aTU$</td>
</tr>
<tr>
<td>$#aTU!$</td>
<td>#abaa%aba!$</td>
<td>match #</td>
</tr>
<tr>
<td>$aTU!$</td>
<td>abaa%aba!$</td>
<td>match a</td>
</tr>
<tr>
<td>$TU!$</td>
<td>baa%aba!$</td>
<td>output $T \rightarrow baT$</td>
</tr>
<tr>
<td>$baTU!$</td>
<td>baa%aba!$</td>
<td>match b</td>
</tr>
<tr>
<td>$aTU!$</td>
<td>aa%aba!$</td>
<td>match a</td>
</tr>
<tr>
<td>$SU!$</td>
<td>%aba!$</td>
<td>output $S \rightarrow %aT$</td>
</tr>
<tr>
<td>$%aTU!$</td>
<td>%aba!$</td>
<td>match %</td>
</tr>
<tr>
<td>$aTU!$</td>
<td>aba!$</td>
<td>match a</td>
</tr>
<tr>
<td>$TU!$</td>
<td>ba!$</td>
<td>output $T \rightarrow baT$</td>
</tr>
<tr>
<td>$baTU!$</td>
<td>ba!$</td>
<td>match b</td>
</tr>
<tr>
<td>$aTU!$</td>
<td>a!$</td>
<td>match a</td>
</tr>
<tr>
<td>$TU!$</td>
<td>!$</td>
<td>output $T \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$U!$</td>
<td>!$</td>
<td>output $U \rightarrow \epsilon$</td>
</tr>
<tr>
<td>!$</td>
<td>!$</td>
<td>match !</td>
</tr>
<tr>
<td>$$</td>
<td>$$</td>
<td>accept</td>
</tr>
</tbody>
</table>
4. What advantage does left recursion have over right recursion in shift-reduce parsing?

**Hint:** Consider left and right recursive grammars for the language $a^*$. What happens if your input has a million $a$’s?

**Answer:** Consider what happens with the right recursive grammar for $a^*$: $S \rightarrow aS \mid \epsilon$. The resulting shift-reduce machine will shift the entire input onto the stack, before performing the first reduction. It will accept all strings in the language, but it requires an unbounded stack size. Now consider $S \rightarrow Sa \mid \epsilon$. This machine will initially reduce by $S \rightarrow \epsilon$ and then alternate between shifting an “$a$” on to the stack and reducing by $S \rightarrow Sa$. The stack space used is thus constant. In general, using left recursion in grammars for shift-reduce parsers helps limit the size of the stack.
5. Consider the following grammar $G$ over the alphabet $\Sigma = \{a, b, c\}$:

\[
\begin{align*}
S' &\rightarrow S \\
S &\rightarrow Aa \\
S &\rightarrow Bb \\
A &\rightarrow Ac \\
A &\rightarrow \epsilon \\
B &\rightarrow Bc \\
B &\rightarrow \epsilon 
\end{align*}
\]

You want to implement $G$ using an SLR(1) parser (note that we have already added the $S' \rightarrow S$ production for you).

(a) Construct the first state of the LR(0) machine, compute the FOLLOW sets of $A$ and $B$, and point out the conflicts that prevent the grammar from being SLR(1).

**Answer:** Here is the first state of the LR(0) machine:

\[
\begin{align*}
S' &\rightarrow .S \\
S &\rightarrow .Aa \\
S &\rightarrow .Bb \\
A &\rightarrow .Ac \\
A &\rightarrow .\epsilon \\
B &\rightarrow .Bc \\
B &\rightarrow .\epsilon 
\end{align*}
\]

We have that $\text{FOLLOW}(A) = \{a, c\}$ and $\text{FOLLOW}(B) = \{b, c\}$. We have a reduce-reduce conflict between production 5 ($A \rightarrow \epsilon$) and production 7 ($B \rightarrow \epsilon$), so the grammar is not SLR(1).
(b) Show modifications to production 4 \((A \rightarrow Ac)\) and production 6 \((B \rightarrow Bc)\) that make the grammar SLR(1) while having the same language as the original grammar \(G\). Explain the intuition behind this result.

**Answer:** We change productions 4 and 6 to be right recursive, as follows:

\[
A \rightarrow cA \\
B \rightarrow cB
\]

As a result, \(c\) is no longer in the follow sets of either \(A\) nor \(B\). Since the follow sets are now disjoint, the reduce-reduce conflict in the SLR(1) table is removed. Intuitively, using right recursion causes the parser to defer the decision of whether to reduce a string of \(c\)’s to \(A\)’s or \(B\)’s. When we reach the end of the input, the final character gives us enough information to determine which reduction to perform.