1. Are the following grammars LL(1)? Justify your answer using FIRST and FOLLOW sets. (Thanks to Karen Lemone, at WPI, for these problems.)

(a) \[ A : \text{d} A, \quad B : g \]

| \text{d} B |
| \text{f} |

Answer: The FIRST and FOLLOW sets for this grammar are:

\[
\text{FIRST}(d A) = \{`d`\} \quad \text{FIRST}(d B) = \{`d`\} \quad \text{FIRST}(f) = \{`f`\} \quad \text{FIRST}(g) = \{`g`\}
\]

\[
\text{FOLLOW}(A) = \{\} \quad \text{FOLLOW}(B) = \{\}
\]

Considering productions \( A \rightarrow \text{d} A \) and \( A \rightarrow \text{d} \), we see that \( `d` \) is in \( \text{FIRST}(d A) \) and \( \text{FIRST}(d B) \). Thus, the grammar is not LL(1).

(b) \[ S : X \text{d} \quad X : C \quad C : \epsilon \quad \text{B : d} \]

| B a |

Answer: The FIRST and FOLLOW sets for this grammar are:

\[
\text{FIRST}(X \text{d}) = \{`d`\} \quad \text{FIRST}(C) = \{\epsilon\} \quad \text{FIRST}(B \text{a}) = \{`d`\}
\]

\[
\text{FIRST}(\epsilon) = \{\epsilon\} \quad \text{FIRST}(d) = \{`d`\}
\]

\[
\text{FOLLOW}(S) = \{\} \quad \text{FOLLOW}(X) = \{`d`\} \quad \text{FOLLOW}(C) = \{`d`\} \quad \text{FOLLOW}(B) = \{`a`\}
\]

Considering productions \( X \rightarrow B \text{a} \) and \( X \rightarrow C \), we see that \( `d` \) is in \( \text{FIRST}(B \text{a}) \) and that \( \epsilon \) is in \( \text{FIRST}(C) \) while \( `d` \) is in \( \text{FOLLOW}(C) \). Thus, the grammar is not LL(1).

2. For the grammar from 1(a):

(a) Rewrite the grammar so that it is LL(1) by introducing the non-terminal \( AB : A \mid B \).

\[ A : \text{d} AB \quad B : g \quad AB : A \]

| \text{f} |
| \quad B |

(b) Compute the FIRST sets. (Why don’t we need the FOLLOW sets for this grammar?)

\[
\text{FIRST}(d AB) = \{`d`\} \quad \text{FIRST}(f) = \{`f`\}
\]

\[
\text{FIRST}(g) = \{`g`\}
\]

\[
\text{FIRST}(A) = \{`d`, `f`\} \quad \text{FIRST}(B) = \{`g`\}
\]
(c) Draw the LL(1) parsing table for the grammar. (Recall that cell \((A, c)\) contains the production \(A \rightarrow \alpha_1 \ldots \alpha_n\) that should be used to parse an \(A\) when the next token is \(c\).)

<table>
<thead>
<tr>
<th>A</th>
<th>d</th>
<th></th>
<th></th>
<th></th>
<th>f</th>
<th></th>
<th></th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>d</td>
<td>AB</td>
<td></td>
<td>A</td>
<td>f</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>AB</td>
<td>A</td>
<td>AB</td>
<td></td>
<td>AB</td>
<td>B</td>
<td>AB</td>
<td></td>
</tr>
</tbody>
</table>

(d) Write the recursive descent parser (recognizer) for the grammar. Assume you are given \(\text{next}()\) and \(\text{scan}()\) functions.

```python
A():
    if next() == 'd': scan('g')
    if next() in ['d', 'f']:
        scan('d'); AB(); A()
    elif next() == 'f':
        scan('f'); B()
    else:
        error()

B():
    A()

AB():
    if next() == 'd': scan('g')
    if next() in ['d', 'f']:
        scan('d'); AB(); A()
    elif next() == 'f':
        scan('f'); B()
    else:
        error()
```

3. Consider the following grammar for numerical expressions with division, addition, and unary minus:

```plaintext
expr : NUM
| expr '／' expr
| expr '＋' expr
| '－' expr

rest : ϵ
| '＋' expr
| '／' expr1
```

(a) Rewrite the grammar so that it is LL(1), so that '／' has higher precedence than '＋', and so that '－' has highest precedence. (Note that '+' and '/' will be parsed in a right-associative way. We can fix '+' and '/' to be left-associative in the semantic actions.)

```plaintext
expr : expr1 rest
    expr1 : expr2 rest1
    expr2 : '－' expr2
    | NUM

rest : ϵ
    rest1 : ϵ
    | '＋' expr
    | '／' expr1
```

(b) Compute the FIRST and FOLLOW sets for your re-written LL(1) grammar.

\[
\text{FIRST}(\text{expr1 rest}) = \text{FIRST}(\text{expr2 rest1}) = \{ '－', \text{NUM} \}
\]
\[
\text{FIRST}(\text{expr2 expr2}) = \{ '－' \}
\]
\[
\text{FIRST}(\text{epsilon}) = \{ \epsilon \}
\]
\[
\text{FIRST}(\text{NUM}) = \{ \text{NUM} \}
\]
\[
\text{FIRST}(\text{expr1}) = \text{FIRST}(\text{rest1}) = \{ '＋', \epsilon \}
\]
\[
\text{FIRST}(\text{expr}) = \text{FIRST}(\text{rest}) = \{ \epsilon \}
\]
\[
\text{FOLLOW}(\text{expr2}) = \{ '／', '＋', \epsilon \}
\]
\[
\text{FOLLOW}(\text{expr1}) = \text{FOLLOW}(\text{rest1}) = \{ '＋', \epsilon \}
\]
\[
\text{FOLLOW}(\text{expr}) = \text{FOLLOW}(\text{rest}) = \{ \epsilon \}
\]
(c) Draw the LL(1) parsing table for your re-written grammar.

<table>
<thead>
<tr>
<th></th>
<th>NUM</th>
<th>'+'</th>
<th>'/'</th>
<th>'-'</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
</tr>
<tr>
<td>rest</td>
<td>'+' expr</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
<td>expr1 rest</td>
</tr>
<tr>
<td>expr1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
</tr>
<tr>
<td>rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
</tr>
<tr>
<td>expr2</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
<td>expr2 rest1</td>
</tr>
</tbody>
</table>

4. (Challenge Question) Find a context-free language (CFL) for which there exists no LL(k) grammar, for any k.
   

5. (Challenge Question) Find a CFL for which there is an LL(k) grammar, for some k > 1, but no LL(1) grammar.
   