1. Grammar for RE

```
RE → RE '∪' concat
   | concat
concat → concat closure
   | closure
closure → item*
   | item+    
   | item
item → (RE)
   | alphabet
   | char
alphabet → [ contents ]
   | [ ^ contents ]
contents → alph contents
   | alph
alph → char ... char
   | char
```

2. Is the following grammar LL(1)? Fix it if it isn’t.

```
A → Ba
B → dab
   | Cb
C → cB
   | Ac
```

No, this grammar is **not** LL(1). This contains a circular instance of left recursion:

```
A → Ba → Cba → Acba
```

Making this LL(1) requires a fair amount of left recursion elimination (both direct and indirect) and a little bit of left factoring. We finally end up with a grammar that looks like this:

```
A → cBbaA’ | dabaA’
A’ → cbaA’ | ε
B → dabB’ | cBbB’
B’ → acbB’ | ε
C → cBC’ | dabacC’
C’ → bacC’ | ε
```
3. Elevator Grammar

\[
\begin{align*}
\text{LiftList} & \rightarrow \text{Lift LiftList} \mid \varepsilon \\
\text{Lift} & \rightarrow \uparrow \text{Lift}' \\
\text{Lift}' & \rightarrow \text{Lift} \downarrow \mid \downarrow
\end{align*}
\]

This grammar is LL(1). We can tell because there is no left recursion and there is no backtracking necessary.

4. What kinds of parsing conflicts are possible?

<table>
<thead>
<tr>
<th>Conflict</th>
<th>Action 1</th>
<th>Action 2</th>
<th>Possible?</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>shift</td>
<td>no</td>
<td>We might think we could get one of these by having two rules that could both accept the next lookahead symbol. However, this will just become the same state in our parser tables. They will either a) diverge – not a problem, the deciding lookahead symbol will decide the correct production rule. b) be identical – this will become obvious when we reduce and be a reduce-reduce error c) be identical to the end of one of the rules – this will cause us to try to reduce one and shift the other (i.e., a shift-reduce error)</td>
<td></td>
</tr>
<tr>
<td>reduce</td>
<td>yes</td>
<td></td>
<td>This can be caused when we have a production that is a subset of a second. We don’t know whether to keep adding (shift) or to match the shorter rule (reduce).</td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>no</td>
<td></td>
<td>In order to accept, we have reduced down to the Goal symbol and we have eof as a lookahead. There is no way that would have a rule that shifted on an eof.</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>no</td>
<td></td>
<td>This wouldn’t make sense – if we can shift than we haven’t encountered an error.</td>
<td></td>
</tr>
<tr>
<td>reduce</td>
<td>reduce</td>
<td>yes</td>
<td>This comes about when we have two rules that match. This comes up most frequently when we have single symbol productions and we duplicate them.</td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>no</td>
<td></td>
<td>In order to accept, we have reduced down to the Goal symbol and we have eof as a lookahead. There is no way that would have a rule that would generate a reduce for this state.</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>no</td>
<td></td>
<td>This wouldn’t make sense – if we can reduce than we haven’t encountered an error.</td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>accept</td>
<td>no</td>
<td>We only have a single Goal symbol, so there is no way that we could have multiple ways to accept.</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>----</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>no</td>
<td></td>
<td>This wouldn’t make sense – if we can accept than we haven’t encountered an error.</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>error</td>
<td>no</td>
<td>This would just be… an error</td>
<td></td>
</tr>
</tbody>
</table>