LR Parsing Solutions

- We begin with 0 on the state stack.
- We use the top of the state stack and the first letter of input to determine the next action.
- If it is a shift, move the top of the input to the symbol stack and push the new state to the state stack.
- If it is a reduce, remove a number of items equal to the length of the right hand side of the rule we reduce by, then shift the nonterminal.
- If the entry does not exist, reject.
- If we shift ⊣, accept. A final reduction for $S'$ is shown here for the sake of completeness; however, a SLR(1) parsing program may just accept when ⊣ is accepted.

<table>
<thead>
<tr>
<th>State Stack</th>
<th>Symbol Stack</th>
<th>Remaining Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\varepsilon$</td>
<td>⊢pqab⊣</td>
<td>shift ⊣, 1</td>
</tr>
<tr>
<td>0 1</td>
<td>⊣</td>
<td>pqab⊣</td>
<td>shift p, 5</td>
</tr>
<tr>
<td>0 1 5</td>
<td>⊢p</td>
<td>qab⊣</td>
<td>reduce X $\rightarrow$</td>
</tr>
<tr>
<td>0 1 5 9</td>
<td>⊢pX</td>
<td>qab⊣</td>
<td>reduce X $\rightarrow$ pX</td>
</tr>
<tr>
<td>0 1 3</td>
<td>⊢X</td>
<td>qab⊣</td>
<td>shift q, 7</td>
</tr>
<tr>
<td>0 1 3 7</td>
<td>⊢Xq</td>
<td>ab⊣</td>
<td>reduce Y $\rightarrow$ q</td>
</tr>
<tr>
<td>0 1 3 2</td>
<td>⊢XY</td>
<td>ab⊣</td>
<td>reduce S $\rightarrow$ XY</td>
</tr>
<tr>
<td>0 1 8</td>
<td>⊢S</td>
<td>ab⊣</td>
<td>shift a, 4</td>
</tr>
<tr>
<td>0 1 8 4</td>
<td>⊢Sa</td>
<td>b⊣</td>
<td>shift b, 10</td>
</tr>
<tr>
<td>0 1 8 4 10</td>
<td>⊢Sab</td>
<td>⊣</td>
<td>reduce S $\rightarrow$ Sab</td>
</tr>
<tr>
<td>0 1 8 6</td>
<td>⊢S</td>
<td>⊣</td>
<td>shift ⊣, 6</td>
</tr>
<tr>
<td>0 1 8 6 10</td>
<td>⊢S⊣</td>
<td>$\varepsilon$</td>
<td>accept</td>
</tr>
</tbody>
</table>

The reversed rightmost derivation can be obtained by reading the rules used in reductions from top to bottom, and then adding the $S'$ rule at the end. Here is the reversed derivation in CFG-R format:

```
X
X p X
Y q
S X Y
S S a b
S' ⊢ S ⊣
```
Furthermore, we can obtain a rightmost derivation by reading the symbol stack concatenated with the remaining input from bottom to top:

\[
\begin{align*}
S' & \Rightarrow \vdash S \vdash \\
& \Rightarrow \vdash S \ a \ b \ \vdash \\
& \Rightarrow \vdash X \ Y \ a \ b \ \vdash \\
& \Rightarrow \vdash X \ q \ a \ b \ \vdash \\
& \Rightarrow \vdash p \ X \ q \ a \ b \ \vdash \\
& \Rightarrow \vdash p \ q \ a \ b \ \vdash 
\end{align*}
\]

We can obtain a parse tree from this derivation by looking at what rule was used to expand each non-terminal symbol in the derivation.

![Parse tree](image)

1. **Error Detection**

1.1 **WLP4**

1. Symbol `ˆ` is not a valid token in WLP4.

2. The variable ‘y’ is declared twice. This is a semantic error.

3. The variable ‘idx’ is not declared at all. This is a semantic error.

4. In WLP4, all variable declarations must proceed all statements. This is a syntax error, since the WLP4 grammar forces this structure.

5. We don’t have character literals (in single quotes) in WLP4. This is a syntax error.

6. The compiler will not detect error in this program, however, this program contains an infinite loop. This is a runtime error.

7. Function f is declared before function g, therefore f has no knowledge about g’s existence. Note that this prevents the use of mutual recursion in WLP4. This is a semantic error.

1.2 **C**

1. It may at first appear that there should be a semantic error when assigning the output of a function that returns a float to a variable of type int, however this is perfectly legal in C. In fact, the type of constants ‘3.0’ and ‘4.4’ are actually double, so when you return the result of ‘a * 3.0’ or pass ‘4.4’ as a parameter to ‘triple’, there is an implicit narrowing conversion to type float! The error actually
occurs in two places: when trying to assign the address of ‘b’ into ‘y’, and when trying to dereference ‘y’. This is because the type of ‘y’ is actually int, not int*, as it may first appear. This is a semantic error.

2. The function ‘getRandom()’ is not declared. This is a semantic error.