1. Determine if each of the following claims is true (T) or false (F).

_F_ A language consists of a set of strings, its grammar structure, and a set of operations.
(Note: a language is nothing more than a set of strings. The structure of the parse tree is a
property of the grammar not the language (e.g. you can create two grammars that parse
1+2+3 in a left-associative or right-associative manner, but they would still accept the same
language.)

_F_ Tokens can be described using regular grammars (RGs) but not context free grammars
(CFGs).

_F_ NFA is more powerful than DFA since it allows epsilon and nondeterministic moves.

_T_ L1={[^i] | i>1} CANNOT be described using a regular expression (RE).

_F_ Since C language is a context free language, we use CFG to parse C programs.

_T_ A grammar is considered ambiguous if, for a sentence in the language generated by
the grammar, we can find two or more parse trees.

_T_ Given a sentence to parse, top-down parsing strategies such as LL(k) find its left-most
derivation starting from the start symbol.

_T_ LR(k) can process both left-recursive and right-recursive grammars.

_T_ LL(2) parse table is bigger than LL(1) parse table.

_T_ For LR-parsing, a handle always appears on the top of the syntax stack.

_T_ SLR(1) parsing is less powerful than LR(1).

_T_ The parse scheme used in YACC tool is LALR parsing.

_T_ For LALR parsing, when an error is detected, the content stored in the syntax stack
may become non-viable.
(Note: since LALR may cause the error detection to be delayed by one more reductions, the
top of the syntax stack may no longer contain a non-terminal that is directly related to the
error but a non-terminal much farther up the parse tree. This may render the stack non-viable
for further parsing the program.)

_F_ Type checking removes all type errors in the program.
(Note: this only applies to so called strongly typed systems. Languages like C/C++ are not
strongly typed. Implicit conversion of values from boolean to int to float can occur during
runtime as we saw in the if-then-else example, that modifies the original type of that value.
This may result in uncaught type errors such as trying to do an addition of a boolean to a float,
that clearly was not the intention of the programmer.)
1. \[ \frac{O \vdash e_0 : T}{T \leq T_0} \quad \frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \] is a correct let-rule for declaration with initialization.

2. Give **brief** answers to the following questions.

   a. What is left factoring? Why is it necessary in a predictive parser but not a recursive descent parser?

   Left factoring is a technique to remove common left factors in grammars. For the grammar \( A \rightarrow \alpha \beta | \alpha \gamma \), a left-factored grammar would be \( A \rightarrow \alpha A' \), \( A' \rightarrow \beta | \gamma \). This is necessary for a predictive parser since a common factor prevents it from making a decision between two expansions based on lookahead. It is not needed in a recursive descent parser because it tries expansions exhaustively without needing to make a decision beforehand.

   b. Describe two ways that you could extend the power of an SLR(1) parser so that it can parse more grammars.

   You could extend the lookahead to SLR(\( k \)). You could increase the precision of the lookahead by making LALR(1) or LR(1).

   c. Write a grammar for the pattern \( a^i b^j c^i \) (\( i, j \geq 1 \)).

   \[
   S \rightarrow a \ S \ c \ | \ B
   \]

   \[
   B \rightarrow b \ | \ b \ B
   \]

   d. In an LR(1) state graph, for the following conflicting states:

   \[
   \text{S1} = \{ \ldots [A \rightarrow \bullet a B , c/d] , \ [C \rightarrow \bullet c , c/d] , \ [E \rightarrow \bullet a , a] \} \}
   \]

   The items that conflict are \([A \rightarrow \bullet a B , c/d] \) and \([E \rightarrow \bullet a , a]\)

   Type of conflict is shift-reduce

   \[
   \text{S2} = \{ \ldots [A \rightarrow \bullet a B , c/d] , \ [C \rightarrow \bullet c , c/d] , \ [F \rightarrow \bullet c , c/d] \} \}
   \]

   The items that conflict are \([C \rightarrow \bullet c , c/d] \) and \([F \rightarrow \bullet c , c/d]\)

   Type of conflict is reduce-reduce

   e. Briefly describe LL(0) and LR(0) grammars. That is what does lookahead 0 mean for each of these grammars?
LL(0) is a top-down parser that produces a leftmost derivation of a string using a lookahead of 0 symbols. Lookahead of 0 in this context means the parser looks 0 symbols ahead to decide on the production rule for a non-terminal. So in effect, there can only be one production rule per non-terminal in LL(0). LR(0) is a bottom-up parser that produces a rightmost derivation of a string using a lookahead of 0 symbols. Lookahead of 0 in this context means the parser looks 0 symbols ahead to decide whether to perform a reduction. So in effect, if there is a reduction item in a state, there can be no other items in that state. These restrictions limit both LL(0) and LR(0) severely and hence neither are used in actual compilers.

3. Construct a DFA for the following NFA:

3. Construct a DFA for the following NFA:

![](image1.png)

DFA for the above NFA:

![](image2.png)

4. Give the First and Follow sets of the following grammar and then explain why or why not the grammar is LL(1). Explain in words without building the parse table. All capital letters are non-terminals and small letters are terminals.

\[
S \rightarrow A \ B \ C \\
A \rightarrow a \ | \ B \ S \ B \\
B \rightarrow b \ | \ C \\
C \rightarrow c \ | \ \varepsilon
\]

Fill in the first and follow set table for non-terminal symbols.

<table>
<thead>
<tr>
<th>First Set</th>
<th>Follow Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>N/A</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
No. According to the criteria on the slides, for $A \rightarrow a \mid B S B$, First (a) $\land$ First (BSB) = \{a\}. Also for $B \rightarrow b \mid C$, First (b) $\land$ Follow (B) = \{b\}. Also for $C \rightarrow c \mid \varepsilon$, First (c) $\land$ Follow (C) = \{c\}. All these sets need to be empty for the grammar to be LL(1). Refer to the below table:
5. Considering the following augmented grammar,

\begin{align*}
0: & \quad S' \rightarrow S \\
1: & \quad S \rightarrow S \ A \\
2: & \quad S \rightarrow A \\
3: & \quad A \rightarrow a \\
4: & \quad A \rightarrow ( S )
\end{align*}

(1) Complete the construction of LR(0) set of items and the state graph.

(2) Build the parse table; and

(3) Decide if the grammar is SLR(1) or not.
Yes, the grammar is SLR(1). Refer to the below state machine and table:
Given the following inference rules:

\[
\begin{align*}
\text{\textit{i is an integer}} & \quad \frac{\text{i : int}}{} \\
\text{\textit{e}_0 : \text{int}} & \quad \frac{\text{\textit{e}_1 : \text{int}}}{\text{\textit{e}_0 + \text{\textit{e}_1} : \text{int}}}
\end{align*}
\]

and

Prove the following:

\[
\begin{align*}
\text{3 is an integer} & \quad \frac{\text{3 : int}}{} \\
\text{4 is an integer} & \quad \frac{\text{4 : int}}{} \\
\text{5 is an integer} & \quad \frac{\text{5 : int}}{} \\
\text{3 + 4 + 5 : int}
\end{align*}
\]

Note that the mechanical application of a rule in each step. There should be no ‘jumps of logic’ in the proof:

\[
\begin{align*}
\text{3 is an integer} & \quad \frac{\text{3 : int}}{} \\
\text{4 is an integer} & \quad \frac{\text{4 : int}}{} \\
\text{5 is an integer} & \quad \frac{\text{5 : int}}{} \\
\text{3 + 4 : int} & \quad \frac{\text{3 + 4 : int}}{} \\
\text{3 + 4 + 5 : int}
\end{align*}
\]