1. There are different ways to write the grammars correctly.
   a. `<binary-formula> ::= <unary-formula> \rightarrow <binary-formula> \mid <unary-formula>
      <unary-formula> ::= \sim <unary-formula> \mid ( <binary-formula> ) \mid <operand>
      <operand> ::= p \mid q`
   b. `<binary-expr> ::= <binary-expr> \rightarrow <unary-expr> \mid <unary-expr>
      <unary-expr> ::= * <unary-expr> \mid <postfix-expr>
      <postfix-expr> ::= <postfix-expr> ++ \mid <id>
      <id> ::= a \mid b`
   c. `<apl-expr> ::= + <apl-expr> \mid - <apl-expr> \mid * <apl-expr> \mid / <apl-expr> \mid
      \langle operand\rangle + <apl-expr> \mid \langle operand\rangle - <apl-expr> \mid
      \langle operand\rangle * <apl-expr> \mid \langle operand\rangle / <apl-expr> \mid \langle operand\rangle
      <operand> ::= a \mid b \mid c \mid ( <apl-expr> )`

(Long nonterminal names, like those used in the solutions here, are fine in general and can be helpful since they are more descriptive, but in an exam situation you might prefer to use short single-letter nonterminal names in order to be able to write answers more quickly.)

2. Both these grammars cure the ambiguity. The first uses the `end` ending an `if` statement to allow an `else` to be correctly associated with the proper `if` statement.
   The second grammar also cures the problem, by requiring the `then` part of the `if` statement to be surrounded by a `begin/end` pair.
   The best way to see this is to try derivations with grammar of some string which has an `if` statement containing a nested `if` statement with an `else` part, and convince yourself there is only one derivation possible for it in each of these grammars.

   Note that these grammars solve the dangling `else` problem by requiring that extra keywords (`end` or `begin` and `end`) be added to some or all `if` statements.

3. a. `<start> ::= ( define f ( <mid> )
      <mid> ::= <formal> <mid> <actual> \mid ) \langle body\rangle ) ( f
      <formal> ::= <id>
      <actual> ::= <expr>
      <body> ::= ...
      <id> ::= ...
      <expr> ::= ...

The idea behind the grammar is that it matches the first formal parameter in the function’s definition with the last actual parameter in the function’s call, the second formal parameter in the function’s definition with the second-to-last actual parameter in the call, etc. You can see that it works by deriving a program in which the function `f` has no parameters and the call has no arguments, a program in which the function `f` has one parameter and the call has one argument, and a program in which the function `f` has several parameters and the call has several arguments.
b. This approach wouldn’t work correctly for more than one function call. As a matter of fact, it is not possible to do this sort of match with a context free grammar (CFG), since a CFG cannot match arbitrarily many symbols. For example, the language \( L = \{ a^n b^n c^n \mid n \geq 0 \} \) is not a context–free language. And this is the formal language that essentially describes matching exactly two function calls with one function definition.

So although grammars are used to describe programming–language syntax when writing a compiler, some syntactic requirements must be enforced through some means other than a grammar. In fact, a later phase of a compiler checks properties like this which can’t be captured in a CFG.

4. a. First create the following DFA, and use it and the procedure or construction given in class to generate the grammar. \( A \) is the start symbol.

![DFA Diagram]

\[
\begin{align*}
A & \rightarrow aB \mid bC \\
B & \rightarrow aA \mid bD \mid b \\
C & \rightarrow bA \mid aD \mid a \\
D & \rightarrow aC \mid bB \\
\end{align*}
\]

b. First create the following DFA, and use it and the procedure given in class to generate the grammar. \( S' \) is the start symbol.

![DFA Diagram]

\[
\begin{align*}
S' & \rightarrow S \mid \epsilon \\
S & \rightarrow aT \mid a \\
S & \rightarrow bS \mid b \\
T & \rightarrow aT \mid a \\
T & \rightarrow bU \\
U & \rightarrow aU \mid bU \\
S' & \rightarrow S \mid \epsilon \\
S & \rightarrow aT \mid a \mid bS \mid b \\
T & \rightarrow aT \mid a \mid bU \\
U & \rightarrow aU \mid bU \\
\end{align*}
\]

Note that the productions for the nonterminal \( U \) are dead productions, meaning they can’t produce a string of terminals (corresponding to the dead state in the DFA). This is the grammar which would result from automatically applying the procedure given in class; an equivalent grammar which would generate the same language would consist of just the first four rules above and not the last two (i.e., simplifying the grammar by omitting its dead productions).

c. First create the following DFA, and use it and the procedure given in class to generate the grammar. \( S' \) is the start symbol.

![DFA Diagram]

\[
\begin{align*}
S' & \rightarrow S \mid \epsilon \\
S & \rightarrow aT \mid a \mid bS \mid b \\
T & \rightarrow aU \mid bS \mid b \\
U & \rightarrow aV \mid bS \mid b \\
V & \rightarrow aV \mid bV \\
\end{align*}
\]

Again, the productions from nonterminal \( V \) are dead productions (so are the productions that generate a \( V \)).