Ans No. 1

Lexemes: A program can be thought of as an array of lexemes. A lexeme is the smallest unit of meaning in a programming language. For example if there is a statement ‘int val = 100’ the lexemes will be ‘int’, ‘val’, ‘=’, ‘100’

Token: Tokens are basically categorized lexemes. Every lexeme can be categorized into keywords, identifiers, operators or an expression. For example, if there is a statement ‘int val = 100’, here tokens are ‘int’(reserved word), ‘val’ (identifier), ‘=’(operator), ‘100’ (a number)

Reserved Words: Reserved words in a programming language can not be used as variable names of function name (identifier). They have special meanings for example ‘if’, ‘break’, ‘for’, ‘while’ are reserved words and they can not be used as an identifier.

Metalanguage: Metalanguage is a language that is used to define the syntax and rules of another language. For example, the Backus-Naur Form (BNF) is a metalanguage that is used to define the syntax of programming languages.

Context-Free Grammar: In formal language theory, a context-free grammar is a formal grammar whose production rules can be applied to a nonterminal symbol regardless of its context. For example, if a language defines it will only have alphabets in their identifier name regardless of their position then it will be context free grammar.

Derivation: A derivation is a sequence of production rule applications that transform a nonterminal symbol into a string of terminal symbols.

Production Rule: A production rule is a rule that describes how a nonterminal symbol can be rewritten as a sequence of terminal and/or nonterminal symbols. For example,

    <expr> → <expr> + <term> | <term>

Ans No. 2

EBNF (Extended Backus-Naur Form) is an extension of BNF (Backus-Naur Form). EBNF overcomes the main disadvantages of BNF, which are that repetition has to be expressed by a recursive definition and that options and alternatives require auxiliary definitions, by incorporating a notation to specify repetition and alternation.

In BNF, we can define a grammar:
<term> → <term> * <factor> | <term> / <factor> | <term> + <factor> | <term> - <factor>

Which can be replaced in EBNF with:
<term> → <term> (* | / | + | -) <factor>

Ans No. 3

The given grammar is:

<*S*>⟶ <*A*>*a*<*B*>*b*
<*A*>⟶<*A*>*b* | *b*
<*B*>⟶*b*

1. For the string "babb"
   <S> → <A> a <B> b
   → b a <B> b [<A> → b]
   → b a b b [<B> → b]

   So string "babb" is derivable from the given grammar

2. For the string "bbbabb"
   <S> → <A> a <B> b
   → <A> b a <B> b [<A> →<A> b]
   → <A> b b a <B> b [<A> →<A> b]
   → b b b a <B> b [<A> → b]
   → b b b a b b [<B> → b]

   So string "bbbabb" is derivable from the given grammar

3. For the string "bbaaaaabc"
   <S> → <A> a <B> b
   → <A> b a <B> b [<A> →<A> b]
   → b b a <B> b [<A> → b]
   → b b a <B> b

   We can't get 'a' or 'c' anyway from the given grammar, hence "bbaaaaabc" is not derivable

4. For the string "aaaaaa"
   <S> → <A> a <B> b

   We can't get 'a' anyway from the given grammar, hence "aaaaaa" is not derivable

Ans No. 4

Here's a grammar for the language that includes all strings containing n copies of alphabet a, followed by n+1 copies of the alphabet b:

<str> → a <str> b | abb

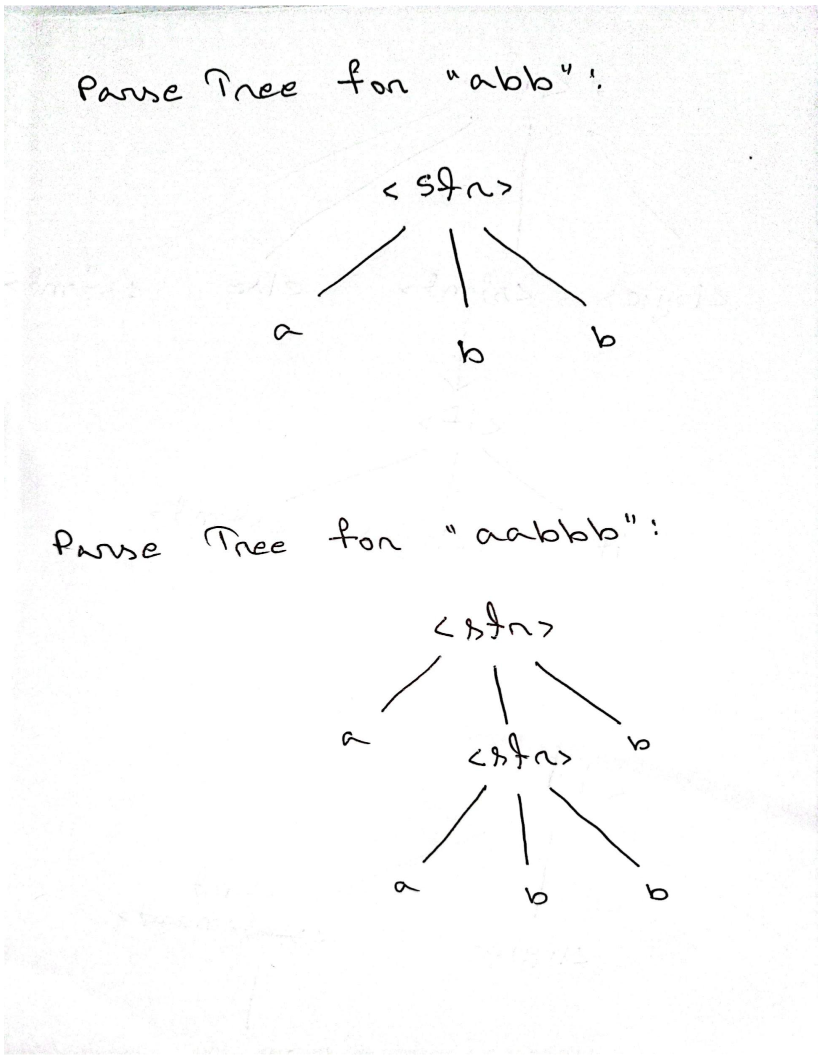
1. For string “abb”,

   <str> → abb

2. For String “aabbb”

   <str> → a <str> b

   → aabbb [<str> → abb]



Ans No. 5

Using EBNF, we can define the grammar as follows:

<id_name> → <letter> {(<letter> | <number>)}
<letter> → (‘A’ | ‘B’ | ‘C’ | …… | ‘Z’ | ‘a’ | ‘b’ | ‘c |’ ……. | ‘z’)
<number> → (0 | 1 | …. | 9)

Ans No. 6

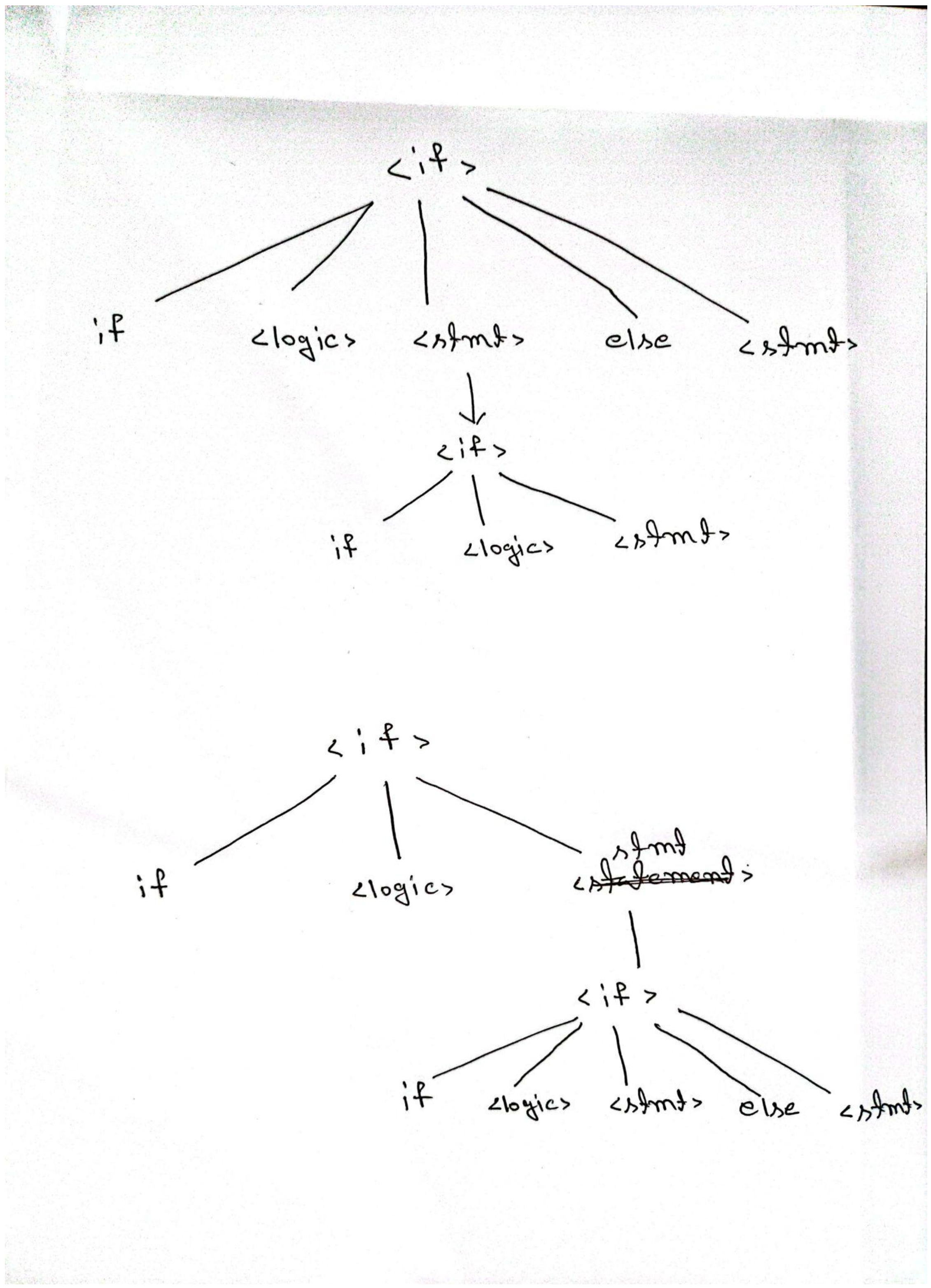
Ambiguity in grammar refers to a situation where a grammar can produce more than one parse tree for a single statement. In other words, the same statement can have multiple interpretations based on the grammar rules applied.

For example considering a grammar defined as:
<if> → if (<logic>) <stmt> | if (<logic>) <stmt> else <stmt>

Now if we have a program

```
if(is_done == true)
if(d == 0)
   q= 0;
else
   q = n / d;
```

We can get two parse tree from the given grammar which will be:



So we can see that there can be two parse trees from the given grammar which is considered as ambiguous grammar.

Ans No. 7

In C programming, three common flow control statements are if-else statements, for loops, and while loops. Here's how the relevant grammars for these statements are written:

1. If-else statement:

```
<if_statement>      →     if ( <logic> ) <statement> [ else <statement> ]
                    |  if ( <logic> ) ‘{‘ <statement_list> ‘}’ [ else ‘{‘ <statement_list> ‘}’ ]
<statement_list>    →     <statement>{<statement>}
```

2. While loop:

```
<while_statement>   →     while ( <logic> ) <statement>
                          | while ( <logic> ) ‘{‘ <statement_list> ‘}’
<statement_list>    →     <statement>{<statement>}
```

3. Do-while loop:

```
<do_while_loop>     →     do ‘{‘ <statement_list> ‘}’ while ( <logic> ) ‘;’
<statement_list>    →     <statement>{<statement>}
```