14. Why can machine languages not be used to define statements in operational semantics?

For operational semantic we need a language that does not have any ambiguous meanings and is easy to interpret. If we use machine language for this purpose. It could be hard to understand and interpret the language and most importantly, the language becomes machine dependent and then the language should have to be rewritten for all machine that wants to use it.

19. Write an attribute grammar whose BNF basis is that of Example 3.6 in Section 3.4.5 but whose language rules are as follows: Data types cannot be mixed in expressions, but assignment statements need not have the same types on both sides of the assignment operator.

1. Syntax rule: <assign> → <var> = <expr>

   Semantic rule: <expr>.expected_type ← <exp>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]

   Semantic rule: <expr>.expected_type ←<var[2]>.actual_type

   Predicate: <var[2]>.actual_type == <var[3]>.actual_type
3. Syntax rule: <expr> → <var>

   Semantic rule: <expr>.actual_type ← <var>.actual_type

   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C

   Semantic rule: <var>.actual_type ← look-up (<var>.string)

1. What are the reasons why using BNF is advantageous over using an informal syntax description?

To create a readable, writable and unambiguous programming language we need a clear syntax rules. These rules should be clear and detailed so that it won't create an ambiguous. We need a powerful syntax description to define such a syntax for programming language. BNF is simple and a powerful tool to describe nearly all of the syntax of programming languages. BNF uses abstractions for syntactic structures which makes it easy to understand and make changes over it. For syntax analysis we need to create a parse tree to detect syntax errors. The Parser can be based directly on the BNF. BNF helps us to create this parse tree more easily with its clear description. With an informal syntax description it would be much harder to create a powerful abstraction and an unambiguous language.

5. Describe briefly the three approaches to building a lexical analyzer.

A lexical analyzer is essentially a pattern matcher. It takes input as a single string of characters and collects characters into logical groupings and assigns internal codes to the groupings according to their structure. There are three approaches to building a lexical analyzer:

1) Firstly we define a formal description of token patterns of the language using a descriptive language related to regular expressions. We decide what will be token's value for identifier, literals and values. Functions takes inputs as a string of characters and read the character of string in this part and decide what it is. After deciding its type, assign a token for its type. If it is an identifier then its token will be IDENT which is a special number and its lexeme will be its value in this case a string of characters. Returns its token and lexeme and waits for the next input. If it is an addition operator then it return ADD_OP token. With this part every string of characters of source code will have its token and lexemes.

2) In this part we describe a state transition diagram that describes token patterns of language. The state diagram is a directed graph. The nodes of the state diagram are labeled with state names. State diagram takes a character every time and keep counting till it comes across with a white space. If characters it takes a letter then set class as LETTER if it is a digit then set class as a DIGIT. It looks every string of character of source code and decide what it is. State diagram helps lexical analysis by grouping character of string into class and with that way it is more easy to give them right token.

3) In this part we design a state diagram same as second part but different is that we look for the special character for this case. For special character we need a hand construct table and decide what special character can be used in this programming language. Such as ( ) + - * / is some of special characters and should be dealt with a construct table. If a character that comes to lexical analyzer is not a letter or digit then it is forwarded to this part. For this special character we look at the table to see if it is allowed or not and if it is allowed then decide its type return its token.