CD Assignment I

1. **Explain the various phases of the compiler with a simple example.**

The compilation process is a sequence of various phases.
Each phase takes input from the previous, and passes the output on to the next phase.

LEXICAL ANALYSIS

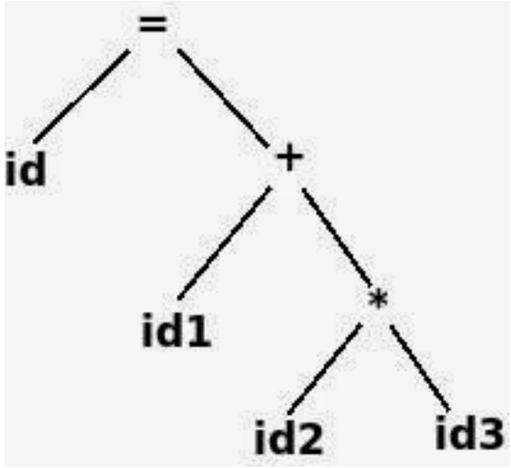
The first phase of compiler works as a text scanner. The lexical analyzer scans the source code as a stream of characters and converts it into meaningful lexemes of the form `<token-name, attribute-value>`.

SYNTAX ANALYSIS

It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). Token arrangements are checked against the source code grammar.

SEMANTIC ANALYSIS

Checks whether the parse tree constructed follows the rules of language. E.g. assignment of values is between compatible data types, and adding string to an integer. It keeps track of identifiers, their types and expressions. It produces an annotated syntax tree as an output.

For example, for input, `x = a + b * c.`	
Lexical Analyzer	`id = id1 + id2 * id3`
Syntax Analyzer	
Semantic Analyzer	No changes

Intermediate code generator	`temp1 = id2 * id3`
	`temp2 = id1 + temp1`
	`id = temp2`

Code Optimizer	`temp1 = id2 * id3` `id = id1 + temp1`
Code Generator	`MUL BX, CX` `ADD AX, CX`

INTERMEDIATE CODE GENERATION

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is generated in such a way that it makes it easier to be translated into the target machine code.

CODE OPTIMIZATION

Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources.

CODE GENERATION

This phase takes the optimized representation of the intermediate code and maps it to the target machine language (sequence of re-locatable machine code).

2. **Write a short note on Compilers and Interpreters. Also explain the various units of language processing system.**

A compiler scans the entire program and translates it as a whole into relocatable machine code. It takes large amount of time to analyze the source code but the overall execution time is comparatively faster. It generates intermediate object code which further requires linking, hence requires more memory. It generates the error message only after scanning the whole program.

An interpreter translates program one statement at a time. It takes less amount of time to analyze the source code but the overall execution time is slower. No intermediate object code is generated, hence are memory efficient. It continues translating the program until the first error is met, in which case it stops.

Figure: Compiler

Figure: Interpreter

A language processing system consists of 4 main units; the *preprocessor*, the *compiler*, the *assembler*, and the *linker/loader*.

The preprocessor collects the source program split into modules and files, expands macros into language statement.

The compiler takes modified source program and generates machine language code.

The assembler takes the machine language code and converts it into relocatable object code.

The linker links pieces of large programs that need to be compiled in pieces. It also resolves external memory references. The loader loads all the executables into memory.

3. **Explain Input Buffering. What is the drawback of using one buffer scheme and explain how it is overcome?**

We often have to look one or more characters beyond the next lexeme, before we can be sure that we have the right lexeme.

For example, the symbols '<' or '=' could be the beginning of '<<', '<=', ==', etc.

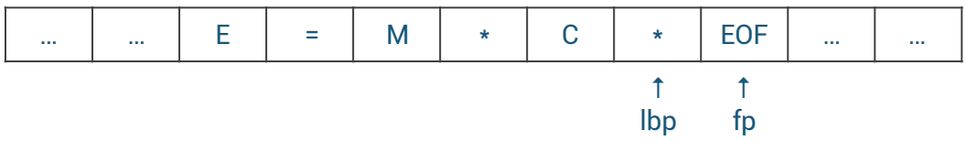
Hence we need two buffer schemes to handle the large lookahead safely.
The buffer pairs consist of a lexeme begin pointer (lbp) and a forward pointer (fp).

The lexeme begin pointer marks the beginning of the current lexeme, and the forward pointer scans ahead until a pattern match is found.

This concept is called Input Buffering.

When using a single buffer scheme, if the end of the current *file/block* is reached before the expression ends, the lexical analyzer might return an unexpected token.

For example, for input string E = M * C ** 2



When EOF is reached in the middle of the expression, the lexical analyzer emits '*' as a token; reloads the input buffer, emits the next '*' as another token.

Two '*'s are returned instead of '**', hence invalidating the expression.

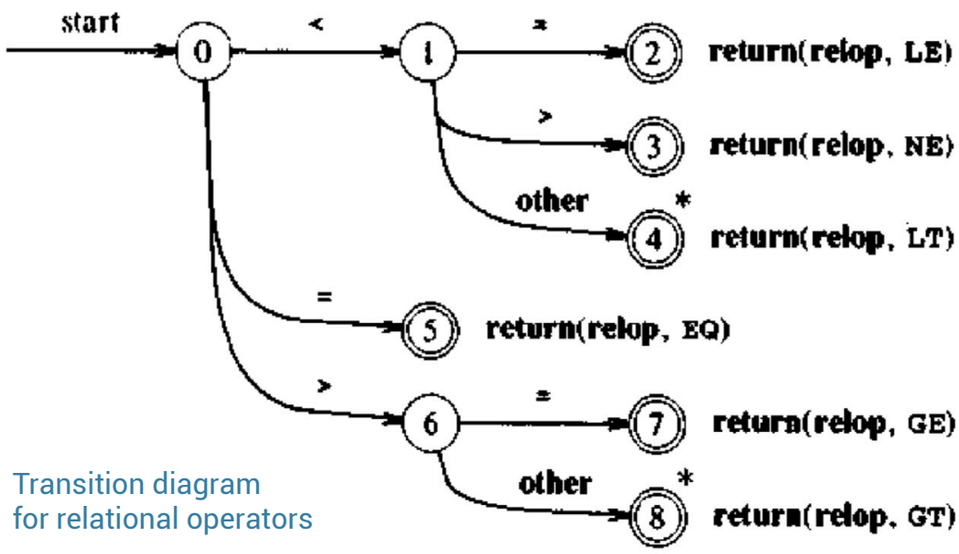
This drawback is overcome by using two buffer schemes.

When we use two buffers, when the input ends prematurely, the other buffer is loaded with the next *file's* contents. Hence, the forward pointer moves to the next file to process the continuing input. The lexical analyzer hence emits '**' as the token.

4. **Explain how transition diagram is used to identify relational operator during Lexical Analysis. Write the code for same.**

Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

We can use a transition diagram to keep track of information about the characters that are seen as the forward pointer scans the input. We do so by moving position to position in the diagrams as the characters are read.



Transition diagram
for relational operators

```
token getRelop () {

 // ...

 while (YES) {

  switch (state) {

   case 0:
    c = getNextChar();
    if (c == '<') state = 1;
    else if (c == '=') state = 5;
    else if (c == '>') state = 6;
    else fail();
    break;

   case 1:
    c = getNextChar();
    if (c == '=') state = 2;
    else if (c == '>') state = 3;
    else state = 4;
    break;

   case 2:
    return newToken(relop, 'LE');

   case 3:
    return newToken(relop, 'NE');

   case 4:
    retract();
    return newToken(relop, 'LT');

   case 5:
    return newToken(relop, 'EQ');

   case 6:
    c = getNextChar();
    if (c == '=') state = 7;
    else state = 8;
    break;

   case 7:
    return newToken(relop, 'GE');

   case 8:
    retract();
    return newToken(relop, 'GT');

   default:
    state = 0;
    break;

  }

 }

 // ...

}
```

5.

A. **Explain the different methods of handling reserved words that look like identifiers.**

There are two ways, you can handle reserved words that look like identifiers:

— **Put the reserved words in a symbol table** initially, with a field indicating that they're not ordinary identifiers, and tells what tokens do they represent.

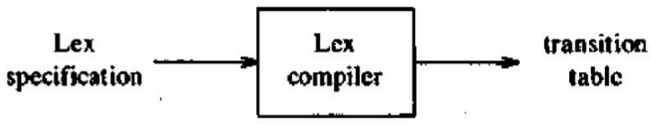
When we find an identifier, a call to *installID()* places it in the symbol table if it's not already there, and returns a pointer to the symbol table entry for the lexeme found. Any identifier not in the table is recognized as an *'id'*.

— **Create separate transition diagrams** for each keyword, consisting of states representing the situation the keyword is to be detected.

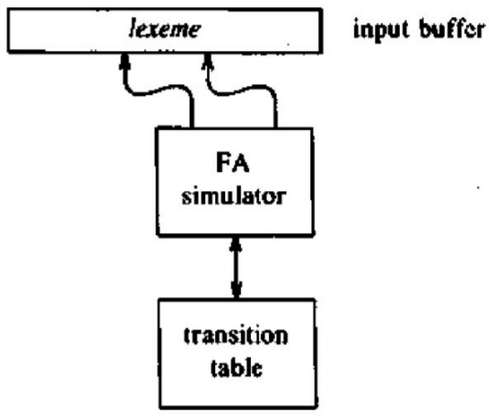
The diagram consists of states representing each letter in the keyword, and check if the identifier has ended, with proper suffix detection. For e.g. 'then' should not be matched in 'thenextstep'.

B. **Write a short note on the structure of Generated Analyzer.**

A generated analyzer is a software tool that automatically constructs a lexical analyzer from a program in the lex language.



(a) Lex compiler.



(b) Schematic lexical analyzer.

The lex compiler generates a transition table for a finite automaton from the regular expressions in the lex specification.

The lexical analyzer consists of finite automaton simulator that uses the transition table to look for the regular expression patterns in the input buffer.