Ans1) (a)

A **lexical analyzer**, also known as a lexer or scanner, is a program that performs lexical analysis, which is the process of converting a sequence of characters into a sequence of tokens. A token is a sequence of characters that has a specific meaning in the programming language. For example, the token "int" might represent the integer data type, and the token "+" might represent the addition operator.

The lexical analyzer interacts with the parser, which is the program that analyzes the syntax of a programming language. The parser takes the tokens produced by the lexical analyzer and uses them to construct a parse tree, which is a representation of the syntactic structure of the program.

The **roles** of the lexical analyzer include:

- Identifying tokens in the input program
- Assigning a meaning to each token
- Passing tokens to the parser
- Reporting errors if an invalid token is found

(b)

Some of the most popular tools are:

- **LEX** is a tool that was originally developed by Mike Lesk at Bell Labs. LEX is a regular expression-based lexical analyzer generator.
- **Flex** is a free and open-source lexical analyzer generator that is based on LEX. Flex is a more powerful tool than LEX, and it supports a wider range of features.

The **working** of a lexical analyzer generator is as follows:

1. The user provides a lexical specification file to the lexical analyzer generator. The lexical specification file contains a set of rules that define the tokens that are allowed in the language.
2. The lexical analyzer generator uses the lexical specification file to generate a C program that implements the lexical analyzer.
3. The C program is compiled and linked into the compiler.
4. When the compiler encounters an input program, it passes the input program to the lexical analyzer.
5. The lexical analyzer scans the input program and produces a sequence of tokens.
6. The parser then takes the sequence of tokens from the lexical analyzer and constructs a parse tree.

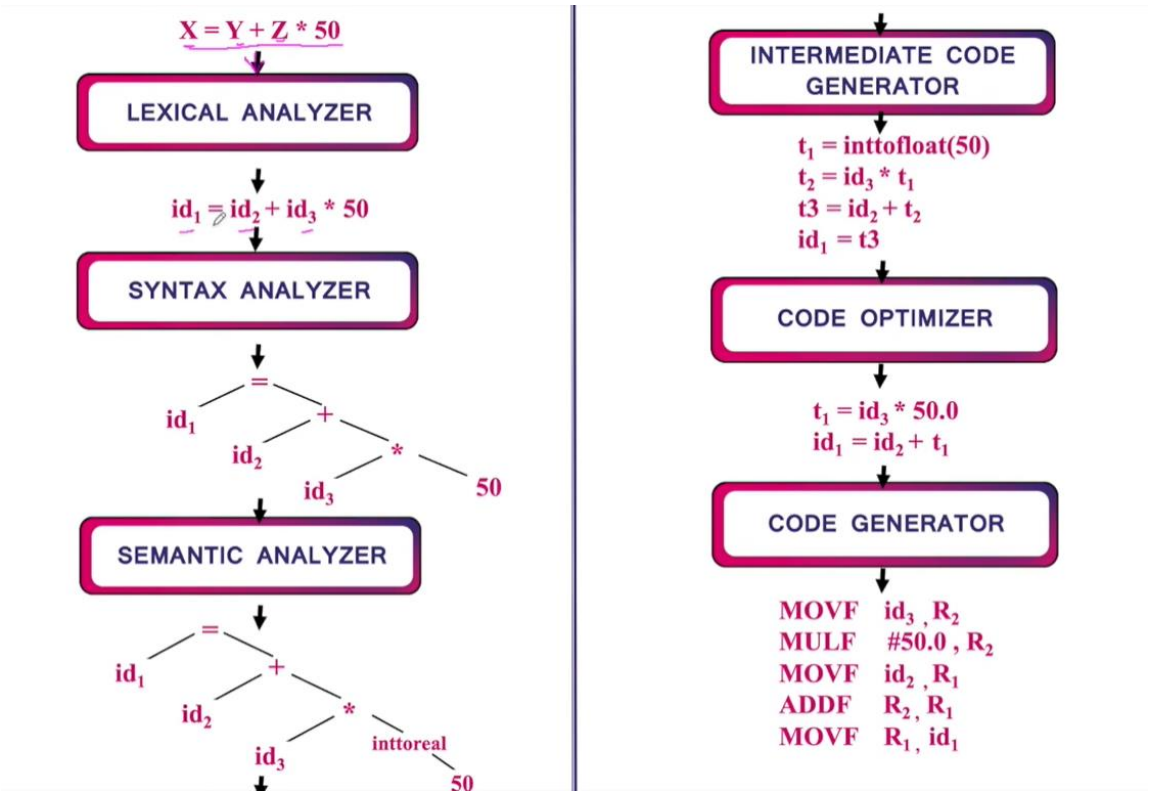
(c)

- **Ambiguity:** Some tokens can be ambiguous, meaning that they can be interpreted in multiple ways. For example, the token "if" can be interpreted as either the keyword "if" or the identifier "if".
- **Incomplete tokens:** Sometimes, a token may be incomplete, meaning that it does not have enough characters to be identified. For example, the token "in" is incomplete if it is not followed by a valid identifier.
- **Invalid tokens:** Sometimes, a token may be invalid, meaning that it does not conform to the rules of the programming language. For example, the token "123abc" is invalid because it contains a digit and a letter.

Ans2) (b)

There are six main phases of a compiler:

1. **Lexical analysis:** This phase breaks down the source code into tokens, which are the basic building blocks of the programming language.
2. **Syntax analysis:** This phase checks the source code to make sure that it is syntactically correct.
3. **Semantic analysis:** This phase checks the source code to make sure that it is semantically correct.
4. **Intermediate code generation:** This phase generates an intermediate representation of the source code.
5. **Code optimization:** This phase improves the performance of the generated machine code.
6. **Code generation:** This phase generates the machine code that will be executed by the computer.



The symbol table allows the compiler to keep track of the variables and functions in the source code. This allows the compiler to perform semantic analysis, code optimization, and code generation correctly.

The symbol table is typically **implemented** as a hash table. A hash table is a data structure that maps keys to values. The keys in the symbol table are the identifiers in the source code. The values in the symbol table are the information about the identifiers, such as the type of the identifier, the scope of the identifier, and the value of the identifier.

The symbol table is **generated** during the lexical analysis phase of the compiler. The lexical analyzer breaks down the source code into tokens. Each token is associated with a key in the symbol table. The value of the key is the type of the token.

The symbol table is **managed** by the compiler during the semantic analysis, code optimization, and code generation phases of the compiler. The semantic analyzer uses the symbol table to check the meaning of the source code. The code optimizer uses the symbol table to optimize the generated machine code. The code generator uses the symbol table to generate the machine code.

Key operations that are performed on the symbol table are:

- **Insertion:** A new entry is added to the symbol table.
- **Lookup:** An entry is searched for in the symbol table.
- **Deletion:** An entry is removed from the symbol table.
- **Update:** The information about an entry in the symbol table is updated.

Ans3) (a)

Key	Top Down Parsing	Bottom Up Parsing
Strategy	Top-down approach starts evaluating the parse tree from the top and move downwards for parsing other nodes.	Bottom-up approach starts evaluating the parse tree from the lowest level of the tree and move upwards for parsing the node.
Attempt	Top-down parsing attempts to find the left most derivation for a given string.	Bottom-up parsing attempts to reduce the input string to first symbol of the grammar.
Derivation Type	Top-down parsing uses leftmost derivation.	Bottom-up parsing uses the rightmost derivation.
Objective	Top-down parsing searches for a production rule to be used to construct a string.	Bottom-up parsing searches for a production rule to be used to reduce a string to get a starting symbol of grammar.

Ans3) (b)

A shift-reduce parser is a bottom-up parser that uses a stack to store the tokens of the input program and a table to store the production rules of the grammar. The parser works by repeatedly shifting tokens from the input program onto the stack and then reducing the top two or more symbols on the stack using a production rule from the table. The process terminates when the stack contains only the start symbol of the grammar.

Ans6) (b)

Syntax-directed definition (SDD) is a technique used in compiler design to associate attributes with syntactic constructs in a programming language. SDDs are used to specify the meaning of a program by describing how the attributes of the program's syntactic constructs are related to each other.

There are two types of SDDs: S-attributed SDDs and L-attributed SDDs.

- S-attributed SDDs: In S-attributed SDDs, the attributes of a production rule are only defined in terms of the attributes of the child nodes of the production rule. This means that the attributes of a production rule can be determined without knowing the attributes of any other production rules in the grammar.
- L-attributed SDDs: In L-attributed SDDs, the attributes of a production rule can be defined in terms of the attributes of the child nodes of the production rule and the attributes of other production rules in the grammar. This means that the attributes of a production rule may depend on the attributes of other production rules in the grammar.

The **evaluation of attributes** in SDD is done using a process called attribute evaluation. Attribute evaluation is a top-down process that starts with the root of the parse tree and works its way down to the leaves.

At each node in the parse tree, the attributes of the node are evaluated using the SDD. The attributes of the child nodes of the node are used to evaluate the attributes of the node.

The process of attribute evaluation continues until all of the attributes in the parse tree have been evaluated.

Steps involved in attribute evaluation:

1. **Initialization:** The attributes of all nodes in the parse tree are initialized to their default values.
2. **Evaluation:** The attributes of each node in the parse tree are evaluated using the SDD.
3. **Error detection:** If any of the attributes in the parse tree cannot be evaluated, an error is reported.
4. **Termination:** The attribute evaluation process terminates when all of the attributes in the parse tree have been evaluated.

(c)

Run-time storage organization is the process of allocating and managing memory during the execution of a program.

Some common data structures include:

Stack: Stack-based run-time storage organization is a method of allocating and managing memory during the execution of a program. In stack-based storage, memory is allocated on a stack, which is a last-in, first-out (LIFO) data structure. When a function is called, a new activation record is pushed onto the stack. The activation record stores the local variables, function parameters, and return address for the function. When the function returns, the activation record is popped off the stack and the memory is deallocated.

Heap: Heap-based run-time storage organization is a method of allocating and managing memory during the execution of a program. In heap-based storage, memory is allocated dynamically. Heap-based storage is used to store dynamically allocated objects, such as strings and arrays. These objects are created using the `new` keyword and they are deallocated using the `delete` keyword. When an object is created using `new`, the run-time system allocates memory for the object on the heap and returns a pointer to the object. When an object is deleted using `delete`, the run-time system deallocates the memory for the object and frees it up for other use.

Ans7) (a)

Intermediate code representation is required for a number of reasons, including:

- **Efficiency:** Intermediate code is often more efficient than generating machine code directly from the source code. This is because intermediate code can be optimized by the compiler, which can improve performance and reduce the size of the generated code.
- **Portability:** Intermediate code is often more portable than machine code. This is because intermediate code is not tied to a specific processor or operating system. This means that the same compiler can be used to generate code for multiple platforms.
- **Reusability:** Intermediate code can be reused for different purposes. For example, it can be used to generate different target languages or to perform code analysis.

A **low-level representation** is a representation of data or instructions that is close to the physical hardware of the computer.

Some low-level representations, are:

- **Machine code:** Machine code is the lowest level of representation. It is a set of binary instructions that are directly executed by the computer's processor.
- **Assembly language:** Assembly language is a human-readable form of machine code. It is a low-level language that uses mnemonics to represent machine instructions.
- **Object code:** Object code is a compiled form of a high-level language. It is a low-level representation that can be directly loaded into the computer's memory and executed.

All the intermediate code representations are:

- **Abstract syntax tree (AST):** An AST is a tree-like representation of the source code. Each node in the tree represents a different construct in the source code, such as a variable, a function, or an expression. ASTs are used by compilers to analyse and transform code.
- **Directed acyclic graph (DAG):** A DAG is a graph that has no cycles. DAGs are used by compilers to represent the control flow of a program.
- **Postfix notation:** Postfix notation is a notation for representing expressions in which the operators follow the operands. Postfix notation is often used by compilers to represent expressions.
- **One address code:** One address code is a representation of instructions that uses one address for each instruction. The address specifies the operand of the instruction. One address code is a simple representation of instructions, but it can be inefficient for some types of code.
- **Two address code:** Two address code is a representation of instructions that uses two addresses for each instruction. The first address specifies the operator, and the second address specifies the first operand. Two address code is more efficient than one address code for some types of code, but it is more difficult to understand and generate.
- **Three address code:** Three address code is a representation of instructions that uses three addresses for each instruction. The first address specifies the operator, the second address specifies the first operand, and the third address specifies the second operand. Three address code is the most efficient representation of instructions, but it is the most difficult to understand and generate.

Ans8) (a)

A **code generator** is a program that converts a high-level language program into a low-level language program, such as machine code or assembly language. The code generator takes the high-level language program as input and produces the low-level language program as output.

The **code generation algorithm** is the set of steps that the code generator follows to convert the high-level language program into the low-level language program. The code generation algorithm typically includes the following steps:

1. Parsing: The code generator parses the high-level language program to create an abstract syntax tree (AST). The AST is a tree-like representation of the high-level language program.
2. Analysis: The code generator analyzes the AST to determine the meaning of the high-level language program. This includes determining the types of variables, the control flow of the program, and the operations that need to be performed.
3. Generation: The code generator generates the low-level language program from the AST. This includes generating instructions for the target machine, assigning registers to variables, and inserting comments.

(b)

Some of the key issues that need to be addressed include:

- The target machine: The code generator must be designed to generate code for a specific target machine. This includes taking into account the instruction set architecture, the memory model, and the register file.
- The high-level language: The code generator must be designed to generate code for a specific high-level language. This includes taking into account the syntax, the semantics, and the data types of the language.
- The code generation algorithm: The code generation algorithm must be designed to generate efficient and portable code. This includes taking into account the specific features of the target machine and the high-level language.
- The code generation framework: The code generator must be designed to be easy to extend and maintain. This includes using a modular design and using a well-defined interface for the code generation algorithm.

(c)

Register allocation is a compiler optimization technique that assigns variables to registers in the target machine. Registers are fast memory locations that can be used to store data and instructions. By assigning variables to registers, the compiler can improve the performance of the generated code.

There are two main approaches to register allocation: global register allocation and local register allocation.

- **Global register allocation** attempts to find a global solution that assigns all variables to registers. This is the most challenging approach, but it can lead to the best performance.
- **Local register allocation** attempts to find a local solution that assigns variables to registers within a basic block. This is a less challenging approach, but it can still lead to significant performance improvements.

(d)

Yacc (Yet Another Compiler Compiler) is a parser generator that is used to create parsers for programming languages.

The YACC specification is a text file that describes the grammar of the programming language that the parser is to generate.

When YACC is run on the YACC specification, it generates a C program that implements the parser. The parser is a recursive descent parser that parses the input stream according to the grammar that is specified in the YACC specification.

The parser that is generated by YACC is a top-down parser. This means that the parser starts at the top of the input stream and works its way down, parsing the input one token at a time. The parser uses a stack to keep track of the current state of the parse.

(e)

To set precedence and associativity in YACC, you use the `%left` and `%right` directives. The `%left` directive specifies that the operator is left-associative, and the `%right` directive specifies that the operator is right-associative.

For example, the following YACC grammar specifies that the `+` and `-` operators are left-associative, and the `*` and `/` operators are right-associative:

```
%%
expression: expression '+' expression
        | expression '-' expression
        | expression '*' expression
        | expression '/' expression
        | number
        | '(' expression ')'
%%
%left '+' '-'
%right '*' '/'
```