Q.1 What is a Language Processor?

- A Language Processor is a **software which bridges a specification or execution gap.**
- Program to input to a LP is referred as a Source Program and output as Target Program.
- A language translator bridges an execution gap to the machine language of a computer system.
- Language processors are essential tools that translate human-readable code into machine-executable instructions.
- Language processors are tools that convert human-readable code into a form that computers can execute.

Q.2 Explain Language Processing activities in brief.

- There are mainly two types of language processing activity which bridges the semantic gap between source language and target language:

1. **Program generation activities**
2. **Program execution activity.**

- A program generation activity aims at automatic generation of a program.
- The source language is a specification language of an application domain and the target language is typically a procedure oriented PL.
- A program execution activity organizes the execution of a program written in a PL on a computer system.
- Its source language could be a procedure oriented language or a problem oriented language.

Program Generation activities

- This activity generates a program from its specification program. Program generation activity bridges the specification gap.
- A program generator is a software system program that accepts the specifications of the program to be generated and generates the program in the target program in a target programming language.
- The program generator introduces a new domain between the application and the programming language domain called the program generator domain.

Program execution activities

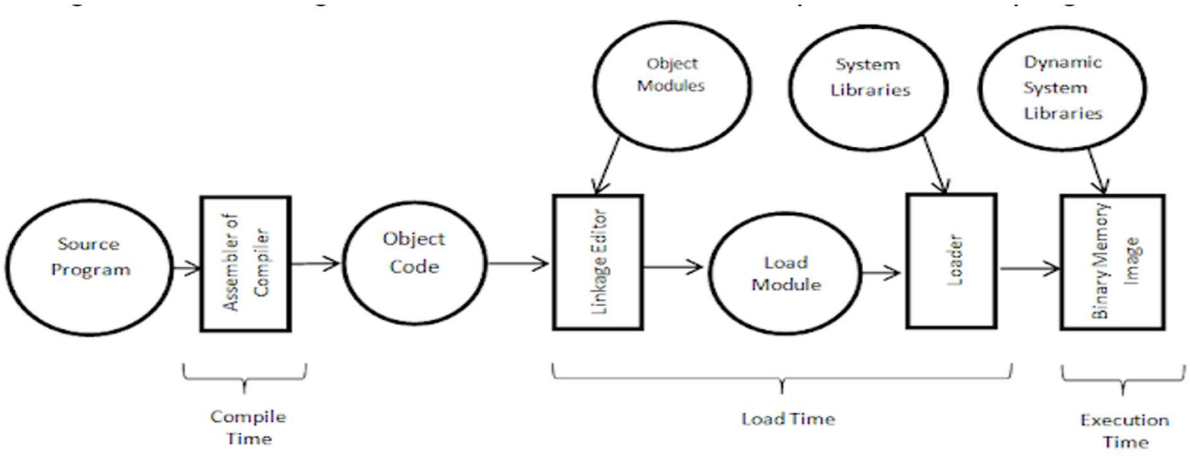
- This activity aims at bridging the execution gap by organizing the execution of a program written in a programming language on a computer system.
- A program may be executed through ***Translation*** and ***Interpretation.***

Name: Sojitra Hemanshu H

Working:

- This activity involves the actual execution of the program generated by the program generator. The program execution activity bridges the implementation gap.
- The program execution activity involves loading the program into the memory of the computer system, interpreting the program instructions, and executing them on the computer system.
- During program execution, the computer system reads the program instructions from memory and executes them one by one, performing the necessary computations and producing the desired results.
- The program execution activity involves various components of the computer system, including the processor, memory, input/output devices, and other system resources required to execute the program.
- In summary, the program generation activity generates a program from its specification, while the program execution activity involves the actual execution of the program on the computer system.

Q.3 Explain Life Cycle of a Source Program.



- **Edit time:** It is the phase where editing of the program code takes place and is also known as design time. At this stage, the code is in its raw form and may not be in a consistent state.
- **Compile time:** At the compile time stage, the source code after editing is passed to a translator that translates it into machine code. One such translator is a compiler. This stage checks the program for inconsistencies and errors and produces an executable file.
- **Distribution time:** It is the stage that sends or distributes the program from the entity creating it to an entity invoking it. Mostly executable files are distributed.
- **Installation time:** Typically, a program goes through the installation process, which makes it ready for execution within the system. The installation can also optionally generate calls to other stages of a program's life cycle.

Name: Sojitra Hemanshu H

- **Link time:** System libraries are linked by using the lookup of the name and the interface of the library needed during compile time or throughout the installation time, or invoked with the start or even during the execution process.
- **Load time:** This stage actively takes the executable image from its stored repositories and places them into active memory to initiate the execution. Load time activities are influenced by the underlying operating system.
- **Run time:** This is the final stage of the life cycle in which the programmed behavior of the source program is demonstrated.

Q.4 Write a short note on Text Editors and its types.

- A text editor is considered a primary interface for all types of workers as they compose, organise, study and manipulate computer based information.
- The term document includes objects such as computer program, text, equations, table, diagram and almost anything that can appear on a printed page.
- Text editors range from simple tools for basic text editing to sophisticated environments for code development.

The editing process is essential for refining and improving text, whether it's for writing, programming, or document preparation.

1. Select the path of the target document to be viewed and manipulated.
2. Determine how to format this view on-line and how to display it.
3. Specify and execute operations that modify the target document.
4. Update the view appropriately.

1. **Line-by-line Editor**

- Line-by-line editors are a type of text editor where the user interacts with the text one line at a time.
- These editors are typically used in command-line environments and are often preferred for their simplicity and efficiency in certain scenarios, such as quick edits on remote servers.
- Users can edit, delete, or insert lines of text one at a time.
- Often used for quick edits or scripting tasks.
- Generally less intuitive for complex editing compared to modern text editors.
- **Examples**: **ed**(for Unix-based systems), **sed**(Stream Editor)

2. File-Oriented Editor (WYSIWYG)

- File-oriented text editors allow users to interact with the entire file as a whole, providing a more comprehensive and visual interface for text editing.
- These editors are generally more user-friendly and suitable for more complex editing tasks, including coding, writing, and document preparation.
- Users can see and edit multiple lines or the entire file at once.
- Often include advanced features such as syntax highlighting, search and replace, and multi-file editing.
- Provide a more intuitive interface for complex tasks.
- **Examples: Notepad++, Visual Studio Code (VS Code), Emacs, Vim**

Q.5 What are phases and passes? Compare and contrast.

Phases

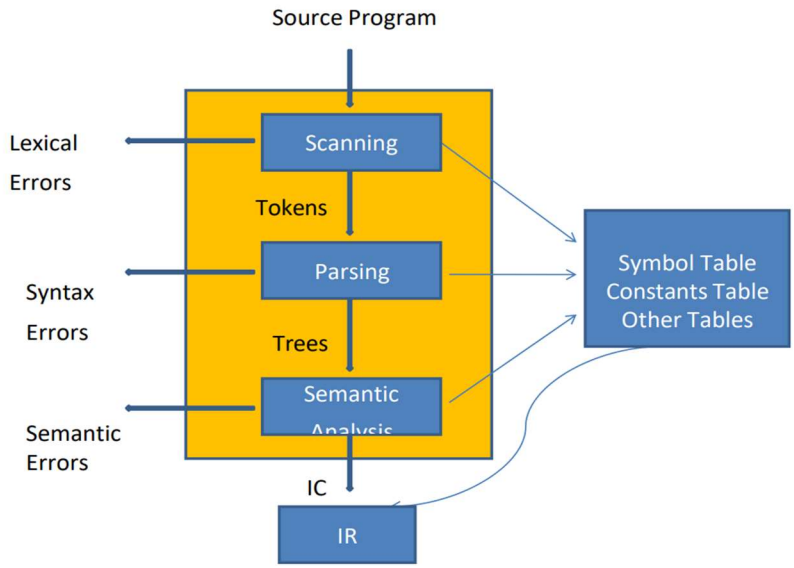
- **Broader Scope:** Phases typically refer to the major stages or steps involved in the entire compilation or execution process.
- **Sequential Execution:** Phases are usually executed sequentially, with the output of one phase serving as the input for the next.
- **Examples:**
    - **Lexical Analysis:** Breaking the input text into tokens.
    - **Syntax Analysis:** Parsing the tokens into a parse tree.
    - **Semantic Analysis:** Analyzing the meaning of the parsed code.
    - **Code Generation:** Generating machine code.
    - **Optimization:** Applying optimizations to improve the generated code.

Phases

- **Broader Scope:** Phases typically refer to the major stages or steps involved in the entire compilation or execution process.
- **Sequential Execution:** Phases are usually executed sequentially, with the output of one phase serving as the input for the next.
- **Examples:**
    - **Lexical Analysis:** Breaking the input text into tokens.

- **Syntax Analysis:** Parsing the tokens into a parse tree.
- **Semantic Analysis:** Analyzing the meaning of the parsed code.
- **Code Generation:** Generating machine code.
- **Optimization:** Applying optimizations to improve the generated code.

Q.6 Draw diagram and explain the analysis phase of a Language Processor.



- This phase can be further divided into several sub-phases:

1. **Lexical Analysis:**

- **Purpose:** Convert the sequence of characters in the source code into tokens.
- **Activities:** Scanning and tokenizing the source code.
- **Output:** Tokens (basic syntactic units like keywords, identifiers, literals, operators).
- **Example:** Converting int a = 5; into tokens [int, a, =, 5, ;].
- Lexical analyzer represents these lexemes in the form of tokens as: Lexical analysis builds a descriptor, called a token.
    - **<token-name, attribute-value>**
- We represent token as entry *"**Code #no**"* where *"Code"* can be Id or Op for identifier or operator respectively and *"no"* indicates the entry for the identifier or operator in symbol or operator table.

2. Syntax Analysis (Parsing):

- **Purpose:** Determine the grammatical structure of the source code.

- **Activities:** Constructing a parse tree from the tokens.
- **Output:** Parse tree or abstract syntax tree (AST).
- **Example:** Verifying that int a = 5; follows the rules of the language's grammar.
- Syntax analysis processes the string of token to determine its grammatical structure builds an intermediate code that represents the structure.
- The tree structure is used to represent the intermediate code.

3. Semantic Analysis:

- **Purpose:** Ensure the code makes logical sense.
- **Activities:** Type checking, scope resolution, and verifying that operations are semantically valid.
- **Output:** Annotated syntax tree.
- **Example:** Checking that variables are declared before use and that operations are performed on compatible types.
- While processing a ***declaration statement***, it adds information concerning the type, length and dimensionality of a symbol to the symbol table.
- While processing an ***imperative statement***, it determines the sequence of actions that would have to be performed for implementing the meaning of the statement and represents them in the intermediate code.

4. Intermediate Code(IC) Generation:

- **Purpose:** Convert the high-level source code into an intermediate representation (IR).
- **Activities:** Generating a platform-independent intermediate code.
- **Output:** Intermediate code.
- **Example:** Transforming int a = 5; into an intermediate form like three-address code.
- IR contains intermediate code and table.

Q.7 Briefly explain IR with an example.

- IR **serve as a bridge** between the source code and the target machine code, providing an abstraction that enables various optimizations and transformations.
- Intermediate codes can be represented in a variety of ways and they have their own benefits.
- **High Level IR**

- High-level intermediate code representation is **very close to the source language itself**. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

- **Low Level IR**
  - This one is **close to the target machine**, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-independent optimization.

Example

```
int x = 10;

int y = 20;

int z = x + y;
```

A possible TAC representation of this code could be:

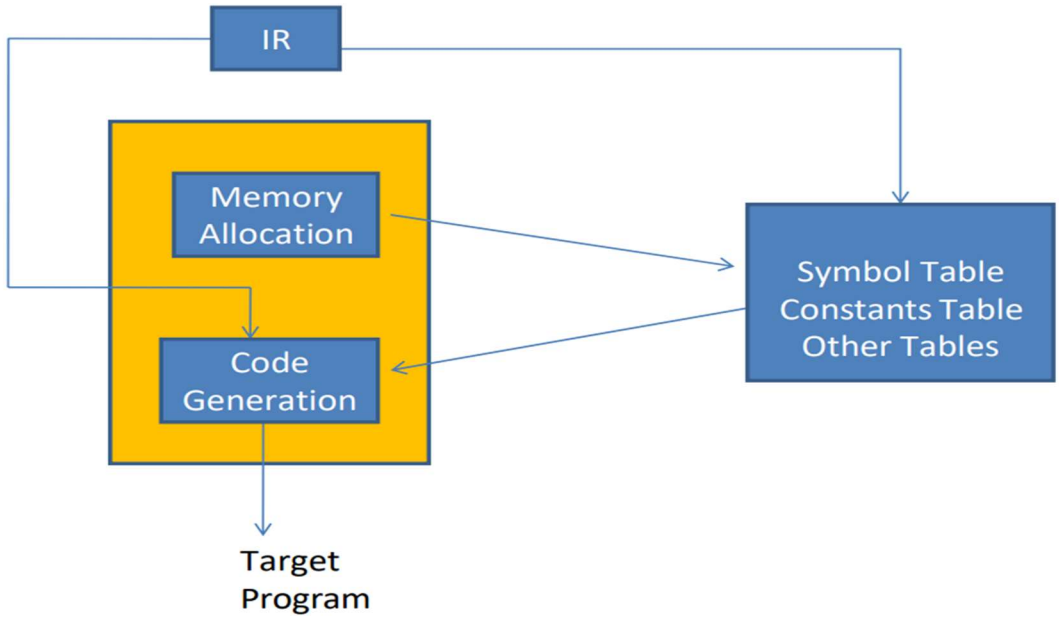
```
t1 = 10

t2 = 20

t3 = t1 + t2

z = t3
```

Q.8 Draw diagram and explain the synthesis phase of a Language Processor.



Name: Sojitra Hemanshu H

- The synthesis phase, also known as the back-end of a compiler, is responsible for translating the intermediate representation of a program into machine code that can be executed by the target processor.

1. **Code Optimization**

- Improve the intermediate code to enhance performance and reduce resource usage.
- **Activities:** Removing redundant code, optimizing loops, and enhancing resource utilization.
- **Output:** Optimized intermediate code.
- **Example:** Eliminating dead code and simplifying expressions to reduce the number of instructions.

2. Code Generation

- **Purpose:** Translate the optimized intermediate code into machine code.
- **Activities:**
    - **Instruction Selection:** Mapping intermediate instructions to the target machine instructions.
    - **Register Allocation:** Assigning variables and temporary values to machine registers.
    - **Instruction Scheduling:** Reordering instructions to minimize pipeline stalls and improve execution speed.
- **Output:** Machine code instructions.
- **Example:** Converting an intermediate representation of an addition operation to the appropriate ADD machine instruction.

3. Creation of Data Structures

- **Purpose:** Set up runtime structures required during program execution.
- **Activities:**
    - **Symbol Table Management:** Storing information about variables, functions, scopes, and types.
    - **Activation Records:** Managing function calls and local variables using stack frames.
    - **Control Flow Graphs (CFG):** Representing the flow of control within the program.
- **Output:** Symbol tables, activation records, and CFGs.
- **Example:** Creating an activation record for a function call to manage local variables and return address.

Q.9 Write a Short note on the Symbol Table.

- Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.
- A language processor uses the symbol table to maintain the information about attributes of symbols used in a source program.
- It performs the following four kinds of operations on the symbol table:

1. **Add a symbol and its attributes:** Make a new entry in the symbol table.
2. **Locate a symbol's entry:** Find a symbol's entry in the symbol table.
3. **Delete a symbol's entry:** Remove the symbol's information from the table.
4. **Access a symbol's entry:** Access the entry and set, modify or copy its attribute information.

- The symbol table consists of a set of entries organized in memory.
- A symbol table may serve the following purposes depending upon the language in hand:

1. To **store the names of all entities in a structured form** at one place.
2. To **verify if a variable has been declared**.
3. To **implement type checking**, by verifying assignments and expressions in the source code are semantically correct.
4. To determine the scope of a name (scope resolution).

Q.10 What are the different data structures used for symbol table? List them and

explain it briefly.

1. Hash Table

- **Advantages: Fast lookup, insertion, and deletion operations, especially for large symbol tables.**
- **Disadvantages: Can have collisions, which can degrade performance if not handled properly.**
- **Implementation: Often implemented using hash functions to map identifiers to unique hash values, which are then used as indices into an array.**

2. Binary Search Tree (BST)

- **Advantages: Efficient search, insertion, and deletion operations for ordered data.**
- **Disadvantages: Can be unbalanced, leading to slow performance in worst-case scenarios.**

- **Implementation: A tree-based data structure where each node has a key (the identifier) and two children (left and right). Nodes are arranged in a way that the key of the left child is less than the key of the parent, and the key of the right child is greater than the key of the parent.**

3. Balanced Binary Search Tree (BBST)

- **Advantages: Guarantees logarithmic search, insertion, and deletion performance, even in worst-case scenarios.**
- **Disadvantages: More complex to implement than a simple BST.**
- **Examples: Red-black trees, AVL trees, B-trees.**

4. Linked List

- **Advantages: Simple to implement, flexible, and allows for efficient insertion and deletion at the beginning or end of the list.**
- **Disadvantages: Slow sequential search, especially for large lists.**