**What's the major difference between an interpreter and a compiler?**

**Interpreter:**

An interpreter transforms or interprets a high-level programming code into code that can be understood by the machine (machine code) or into an intermediate language that can be easily executed as well. The interpreter reads each statement of code and then converts or executes it directly.

**Compiler**:

A compiler is a software program that transforms high level source code that is written by a developer in a high-level programming language into a low-level object code (binary code) in machine language, which can be understood by the processor. The process of converting high- level programming into machine language is known as compilation.

**Difference between Compiler and Interpreter:**

| | |
|---|---|
| Interpreter translates just one statement of the program at a time into machine code. | Compiler scans the entire program and translates the whole of it into machine code at once. |
| An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower. | A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster. |
| An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory. | A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed. |
| Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy. | A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler. |

Interpreters are used by programming languages like Ruby and Python for example.

Compliers are used by programming languages like C and C++ for example.

**Describe the rules for constructing regular expression?**

The set of words accepted by a finite automaton, F, forms a language, denoted L(F). The transition diagram of the fa specifies, in precise detail, that language. It is not, however, a specification that humans find intuitive. For any fa, we can also describe its language using a notation called a regular expression.

The language consisting of the single word fee can be described by an re written as fee. Writing two characters next to each other implies that they are expected to appear in that order. The expression fee is an re for the language.

The language consisting of the two words fee or while can be written as fee or while. To avoid possible misinterpretation of or, we write this using the symbol | to mean or. Thus, we write the re as fee | while.

The language consisting of fee or fie can be written as fee | fie. A second re is possible, f(e|i)e. The re f(e|i)e might reflect the structure of the fa more closely than fee | fie.

To make this concrete, consider some examples from programming languages. Punctuation marks, such as colons, semicolons, commas, and various brackets, can be represented by their character representations. Thus, we might find the following res in the lexical specification for a typical programming language

: ; ? => ( ) { } [ ]

Similarly, keywords have simple res.

if while this integer instanceof

**Consider the following statement**

**Result=(a+b)/(c-d)-5+(c*d)**

**Differentiate between source language and Target language**

**Source language:**

A program in the language consists of a block with optional declarations and statements.

Token basic represents basic types.

$$
\begin{array}{rcl}
program & \rightarrow & block \\
block & \rightarrow & \{\ decls\ stmts\ \} \\
decls & \rightarrow & decls\ decl \mid \epsilon \\
decl & \rightarrow & type\ \mathbf{id}\ ; \\
type & \rightarrow & type\ [\ \mathbf{num}\ ] \mid \mathbf{basic} \\
stmts & \rightarrow & stmts\ stmt \mid \epsilon
\end{array}
$$

**Target language:**

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine.

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.

**Write a detailed short note on following**

**Syntax analysis :**

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

This tree shows the order in which the operations in the assignment

position = initial + rate * 60

are to be performed. The tree has an interior node labeled with **{id, 3}** as its left child and the integer **60** as its right child. The node **{id, 3}** represents the identifier rate. The node labeled makes it explicit that we must first multiply the value of **rate** by **60**. The node labeled + indicates that we must add the result of this multiplication to the value of **initial**.

The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.
we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars.

**Code optimization**

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code **(1.3)**, using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform (1.3) into the shorter sequence

t1 = id3 * 60.0

id1 = id2 + t1

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called “optimizing compilers” a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.