Why symbol table are required in compiler? In which part of memory are they stored? Discuss the main functions of symbol table. Name the data structure used for symbol table organization. Explain how scope information is represented by symbol table organization using hash table.

A symbol table is a data structure used by compilers to store information about the identifiers used in a program. This information includes the identifier's name, type, scope, and value. The symbol table is used by the compiler to perform a variety of tasks, such as:

- **Name resolution:** The compiler uses the symbol table to resolve identifiers to their corresponding objects. This is necessary to ensure that the compiler can correctly interpret the program.
- **Type checking:** The compiler uses the symbol table to check the types of expressions to ensure that they are compatible. This is necessary to prevent errors at runtime.
- **Scope checking:** The compiler uses the symbol table to verify that identifiers are used in the correct scope. This is necessary to prevent errors at runtime.
- **Optimization:** The compiler can use the symbol table to perform optimizations, such as common subexpression elimination and dead code elimination.

Symbol tables are typically stored in the compiler's **heap memory**. This is because the size of the symbol table can vary depending on the size of the program.

The **main functions** of a symbol table are:

- To store information about identifiers used in a program.
- To provide a way to look up identifiers by name.
- To provide a way to determine the type, scope, and value of an identifier.

The data structure used for symbol table organization is typically a **hash table**. A hash table is a data structure that maps keys to values. The key is the identifier name, and the value is the identifier's information.

Scope information is represented by symbol table organization using hash tables by dividing the symbol table into a hierarchy of scopes. Each scope has a unique name, and the identifiers declared in a scope are only visible within that scope.

For example, the following code shows a simple symbol table with two scopes:

Code snippet
```
int main() {
  int a = 10; // Identifier `a` is declared in the global scope.
  {
    int b = 20; // Identifier `b` is declared in the local scope.
  }
}
```

The symbol table for this code would look like this:

Code snippet
```
Global scope:
  a: int
Local scope:
  b: int
```

What is peephole optimization? Write short note on code generator. What is the role of compiler construction tools? Name some of the compiler construction tools

Peephole optimization is a compiler optimization technique that involves looking at small sequences of instructions and replacing them with equivalent sequences that are more efficient. Peephole optimization is a local optimization technique, meaning that it only considers small sequences of instructions. This makes it a relatively simple optimization technique to implement, but it can also be very effective in improving the performance of code.

Here are some examples of peephole optimizations:

- **Redundant instructions:** Peephole optimization can remove redundant instructions. For example, if the same value is loaded into a register twice, the second load can be removed.
- **Combining instructions:** Peephole optimization can combine two instructions into a single instruction. For example, the addition and assignment instructions `a = b + c` can be combined into a single instruction `a += c`.
- **Constant folding:** Peephole optimization can fold constants. For example, the multiplication instruction `a = b * c` can be folded into a constant if `b` and `c` are both constants.

Code generator is a part of a compiler that translates the high-level language code into low-level code, such as assembly language or machine code. The code generator is responsible for generating code that is efficient and easy to understand by the target machine.

The code generator typically works by first translating the high-level language code into an intermediate representation, such as a tree or a graph. The intermediate representation is then used to generate the low-level code.

Compiler construction tools are tools that help developers to build compilers. These tools can be used to automate many of the tasks involved in compiler construction, such as lexical analysis, syntax analysis, semantic analysis, and code generation.

Some of the most popular compiler construction tools are:

- **Bison:** Bison is another parser generator that is similar to ANTLR.
- **Flex:** Flex is a lexical analyzer generator that can be used to generate lexical analyzers for a variety of programming languages.
- **YACC:** YACC is a parser generator that is similar to Bison.