NAME: JUBRIL DAMILOLA

MATRIC NUMBER: 17/SCI01/044

COURSE CODE: CSC312

COURSE TITLE: Compiler Construction

ASSIGNMENT

With the aid of a diagram, describe how a C++ code can be converted to Machine Language code.

Solution

**Phases of a compiler**

Source Program

Lexical Analyzer

Symbol

Table

Error

Handler

Syntax Analyzer

Semantic Analyzer

Intermediate Code Generation

Code Optimization

Code Generation

Target Code

**Lexical Analysis**

This is the first phase of the compilation process and is handled by the lexical analyzer which is also called the Scanner. In this phase the input C++ source code is scanned and separated into lexical units called tokens. The lexical analyses reads the input code character-by-character.

The Symbol table is generated in this phase and populated with tokens generated. A symbol table is typically a data structure that holds a record for each identifier in the source code.

The output of this phase is Stream of Tokens

**Syntax Analysis**

This phase is handled by the syntax analyser. The stream of tokens generated in the lexical analysis phase is analyzed further to ensure that the input code follows the syntax of the particular language.

Syntax errors are detected in this phase.

The output of this phase includes abstract syntax trees

**Semantic Analysis**

Semantic analysis is handled by the Semantic Analyses and has to do with ensuring that the source code follows standard semantic rules.

Type Checking is taken care of in this phase. This ensures that the variables are assigned values according to their declaration.

So if a variable have been declared as integer and then assigned a float, the error is trapped by the Semantic Analyzer.

This phase also identifies chunks of code such as operands and operators of statements in the input code.

The output of this phase includes the Parse Tree

 **Intermediate Code Generation**

Intermediate code refers to a code that is somehow between the source code and the target code, an intermediate representation of the input source program. One attribute of an Intermediate Code is ease of translation to target program.

An example would be a C++ programs compiled into C++ Bytecodes (.class files) for the C Virtual Machine.

One form of intermediate code is the “Three-Address-Code” which resembles an assembly language.

The final target code is generated from the intermediate code.

 **Code Optimization**

In Code Optimization, the code is optimized to remove redundant codes and the optimize for efficient memory management as well as improve the speed of execution. The intermediate code ensures that a target code can be generated for any machine enabling portability across different platforms.

Output of this phase is the Optimized Code.

**Target Code Generation**

Here the target code is generated for the particular platform. Machine instruction are generated from the optimized intermediate code. Assignment of variables and registers is handled here.The output of this phase is the target code.

**Symbol Table**

A symbol table contains information about all the identifiers in the program along with important attributes such as type and scope. Identifiers can be found in the lexical analysis phase and added to the symbol table. During the two phases that follow, the compiler updates the identifier entry in the table. To include information about its type and scopes. When generating intermediate code, the type of the variable is used to determine which instructions to emit. During optimization, the live range of each variable may be placed in the table to aid in register allocation. The memory location determined in the code generation phase might also be kept in the symbol.

**Error Handling**

Another activity that occurs across several phases is called error handling. Most errors handling occurs in the first 3 phases of analysis stage. The scanner keeps an eye for stray tokens, the syntax analysis reports invalid combination of tokens, and the semantics analysis phase reports type errors and the likes. Sometimes these are fatal errors that stop the entire process, while at other times the compiler can recover and continue