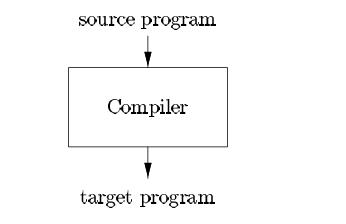
What is compiler? Explain phases of a compiler.  
A compiler is a program that can read a program in one lan guage | the source language | and translate it into an equivalent program in another language | the target language;



Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process.

It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes.

Lexical analyzer represents these lexemes in the form of tokens which are the basic units of the programming language.

It removes whitespace and comments from the source code and produces a stream of tokens for further processing.

The tokens are then passed on to the next phase for further processing.

Syntax Analysis

Syntax analysis is the second phase of compilation process.

It takes tokens as input and generates a parse tree as output.

This phase, often called parsing, analyzes the structure of the source code according to the grammar of the programming language.

It verifies whether the sequence of tokens conforms to the syntactic rules specified by the language's grammar. If the code is syntactically correct, it constructs a parse tree or abstract syntax tree representing the structure of the code.

Semantic Analysis

Semantic analysis is the third phase of compilation process.

It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions.

Semantic analysis checks the meaning of the source code beyond its syntax. It performs tasks such as type checking to ensure that variables are used consistently and according to the language rules, and scope resolution to determine the scope of identifiers.

Semantic analysis also detects certain semantic errors that cannot be captured by syntax alone.

The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

In this phase, the compiler translates the source code into an intermediate representation that is independent of the target machine architecture.

Intermediate code is generated between the high-level language and the machine language.

The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Common intermediate representations include three-address code, abstract syntax trees, and intermediate languages like LLVM IR.

Code Optimization

Code optimization is an optional phase.

It is used to improve the intermediate code so that the output of the program could run faster and take less space.

It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

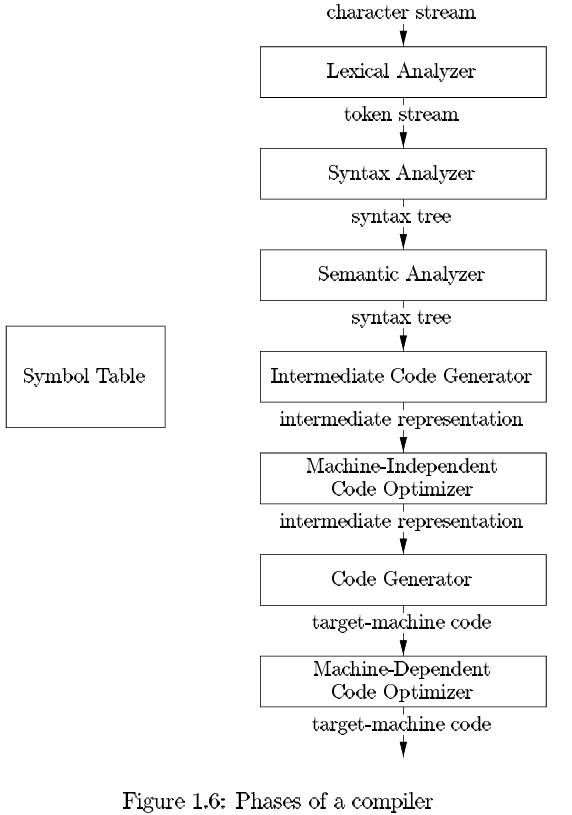
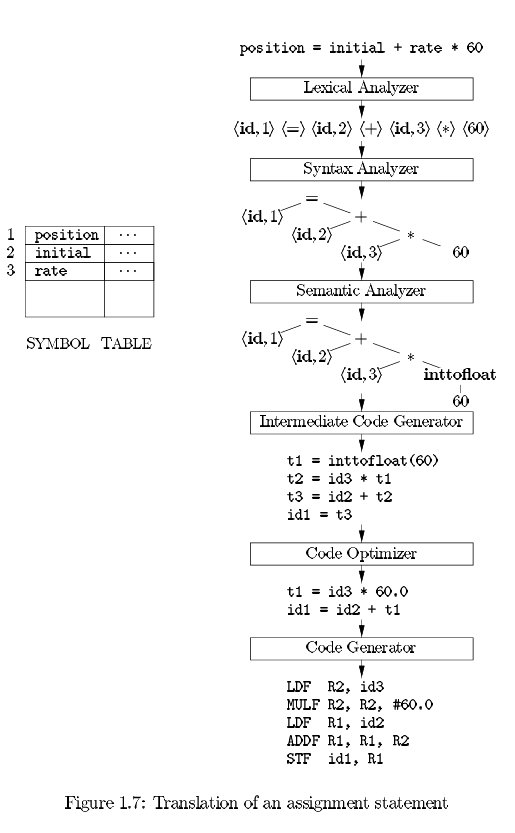
Optimization aims to minimize execution time, memory usage, and other resources while preserving the semantics of the program.

Code Generation

Code generation is the final stage of the compilation process.

It takes the optimized intermediate code as input and maps it to the target machine language.

Code generator translates the intermediate code into the machine code of the specified computer.

Explain different following tools for which compiler technology is used to create:

**a. Structure editors**

Structure editors are tools used for creating and manipulating the structure of programs or documents, often in graphical form.

Compiler technology can be used to parse the structure of the program or document, providing syntax highlighting, code completion, and error checking features.

These editors use lexical and syntactic analysis techniques similar to those used in compilers to understand the structure of the code or document and provide intelligent editing capabilities.

Examples of structure editors include:

Eclipse’s Java Development Tools (JDT)

Visual Studio's C# Development Tools

**b. Pretty printers**

Pretty printers are tools used to format source code in a visually pleasing and consistent manner.

Compiler technology can be used to parse the source code and then regenerate it in a standardized format according to predefined formatting rules.

This process involves lexical and syntactic analysis to understand the structure of the code and generate properly formatted output.

Pretty printers can be used to improve code readability and maintainability by enforcing consistent coding styles.

Examples of pretty printers include:

Eclipse’s Java Pretty Printer

Visual Studio's C# Pretty Printer

c. Static Checker

Static checkers are tools used to analyze source code for potential errors or violations of coding standards without actually executing the code.

Compiler technology is used to perform static analysis of the code, examining its structure and behavior to identify potential issues such as type mismatches, uninitialized variables, or unreachable code.

Static checkers can help improve code quality and reliability by detecting issues early in the development process.

Examples of static checkers include:

Eclipse’s Java Compiler’s built-in static checker

Visual Studio's C# Compiler’s built-in static checker

d. Interpreters

Interpreters are tools used to execute source code directly, typically line by line, without the need for a separate compilation step. Compiler technology can be used to implement interpreters by first parsing the source code and then executing it directly, often using an intermediate representation such as bytecode or an abstract syntax tree. Interpreters are commonly used in scripting languages and development environments where rapid prototyping and interactive debugging are important.

Examples of interpreters include:

Python’s built-in interpreter

Ruby’s built-in interpreter

e. Silicon Compilers

Silicon compilers are tools used to generate hardware descriptions or configurations for designing integrated circuits (ICs) or hardware systems. Compiler technology is used to translate high-level hardware description languages (HDLs) such as Verilog or VHDL into low-level representations such as gate-level netlists or register transfer level (RTL) descriptions. Silicon compilers optimize the design for factors such as performance, power consumption, and area utilization, and can also perform tasks such as technology mapping and placement and routing to generate the final hardware configuration.

Examples of silicon compilers include:

ARM’s Compiler for ARM processors

Intel’s Compiler for Intel processors

Explain different compiler construction tools.

1. Parser generators

2. Scanner generators

3. Syntax-directed translation engines

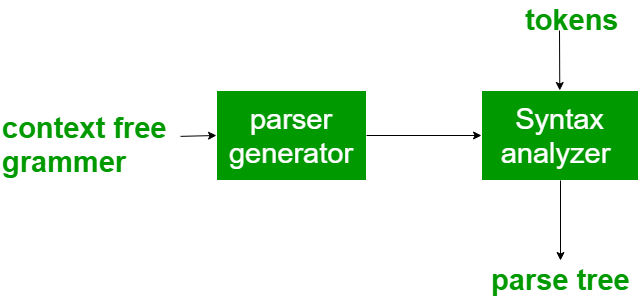
4. Code-generator

5. Data- flow analysis engines

6. Compiler-construction toolkits

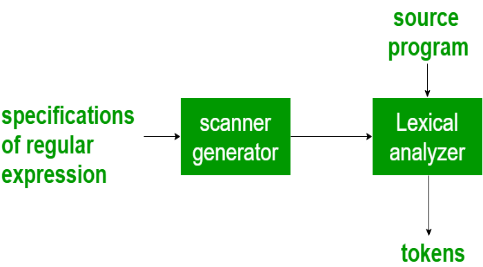
1. **Parser Generators:**

Parser generators automate the construction of parsers based on formal grammars. They take a formal grammar of the programming language as input and generate source code for a parser that can recognize syntactic structures in the input source code. Examples include Yacc (Yet Another Compiler Compiler) and Bison. Parser generators are commonly used in the syntax analysis phase of a compiler.



1. **Scanner Generators:**

Scanner generators, also known as lexical analyzer generators, automate the creation of lexical analyzers or scanners. They take regular expressions as input and generate code for recognizing tokens or lexemes in the input source code. Tools like Lex and Flex are examples of scanner generators. They are typically used in the lexical analysis phase of a compiler to tokenize the input source code.



1. **Syntax-Directed Translation Engines:**

Syntax-directed translation engines provide tools for building parsers and performing syntax-directed translation. They allow developers to define grammatical rules and associate translation actions with those rules. These translation actions perform tasks such as constructing abstract syntax trees (ASTs) or generating intermediate code directly during parsing. ANTLR (ANother Tool for Language Recognition) is a popular syntax-directed translation engine.

1. **Code Generator:**

Code generators translate intermediate representations (such as ASTs or intermediate code) into executable code for a target platform. They handle tasks such as instruction selection, register allocation, and code optimization to produce efficient machine code. Code generators are essential in the code generation phase of a compiler. Examples include GCC (GNU Compiler Collection) and LLVM's code generation backend.

1. **Data-Flow Analysis Engines:**

Data-flow analysis engines analyze the flow of data within a program to derive useful information for optimization. They perform analyses such as reaching definitions, available expressions, and live variables analysis to optimize code for performance and resource usage. Data-flow analysis engines are crucial components of optimizing compilers. LLVM and GCC incorporate data-flow analysis engines for optimization.

1. **Compiler-Construction Toolkits:**

Compiler-construction toolkits provide a comprehensive set of tools and libraries for building compilers. They typically include components for lexical analysis, parsing, semantic analysis, code generation, and optimization. Compiler-construction toolkits simplify the development of compilers by providing reusable components and infrastructure for implementing various compiler phases. Examples include LLVM, which provides a modular infrastructure for building compilers, and JavaCC (Java Compiler Compiler), which is a toolkit for building compilers and interpreters for Java-like languages.

Explain cousins of compiler.

1. Pre-processor :
   * Preprocessor produces input to compiler. They may perform the following functions :
     + Macro processing : A preprocessor may allow user to de for longer constructs.
     + File inclusion: A preprocessor may include the header file into the program text.
     + Rational preprocessor: Such a preprocessor provides the user with built in macro for construct like while statement or if statement.
     + Language extensions: this processors attempt to add capabilities to the language by what amount to built in macros.
2. Assembler :
   * Assembler is a translator which takes the assembly program as machine code as a output.
   * An Assembler is a mnemonic version of the machine code.
3. Linker
   * Linker allows us to make a single program from a several files of relocatable machine code. These file may have been the result of several different compilation, and one or more may be library files of routine provided by a system
4. Loader
   * The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at proper location

