Compiler Design
Module 1

Module 1 - Introduction

Objective: To understand the processes involved in Compiler Design.

1. Introduction :

This module starts with discussing the need for a Translator, Compiler. This module also tries to group the compiler into phases which will be discussed in the later part of this module. To begin, let us get to introduce a brief history of compilers.

1. A brief History.

In this Context, software can be defined as an essential component of the current scenario. Normally in earlier days software was written in assembly language. The instructions are written in Mnemonic code. For example, to add two numbers the following would be the assembly code.

```
MOV R1, a
MOV R2, b
ADD R1, R1, R2                                  → 1.1
```

In statement (1.1), MOV is a command that would move the value stored in variable ‘a’ to register ‘R1’, ‘b’ to R2. The command ADD then adds the contents of the registers R1 and R2 and stores the result in R1. As one could observe, these instructions are closer to the machine than to the human. The drawbacks of writing programs in assembly instructions are:

- Very difficult to remember instructions
- Benefits of reusing software on different CPUs became greater than the cost of designing compiler
- Very cumbersome to write

These drawbacks trigger the need for software that will understand human language and that is the birth of Language Processors called translators.

1.1.2. Language Processors

A **translator** is one that converts a **source program** written in one language to a **target** program in another language. This is similar to having a translator when two people who doesn’t know the other person’s language want to communicate. In the context of computer Science, a **Source program** is written in one programming language and a Target Program typically belongs to machine language. The Target language is called machine language as it is easier for the machine to understand. Some of the translators are As**sembler, Compiler and Interpreter.** Compiler converts programs written in high-level programming language to assembly language. Assemblers convert assembly language programs to machine language (object language).

The translators help programmers to write programs in a language that is easier for them to remember and understand and converts them into a language that is closer to the machine. This results in the following ways of designing software:

a. Design an interpreter / translator to convert human language to machine language

   The interpreter will have difficulties in parsing which may be ambiguous. For example, inefficient parsing would result in incorrect word boundaries during interpretation resulting in ambiguity.

b. Design a compiler that will understand high level language which is not necessarily in English but closer to English and convert that to assembly language.

   The design is complex but parsing ambiguity could be avoided. The major drawback is the mapping of the high level language to assembly language. This also necessitates the designing a compiler for every high level programming language keeping in mind the instruction set of the target assembly language.

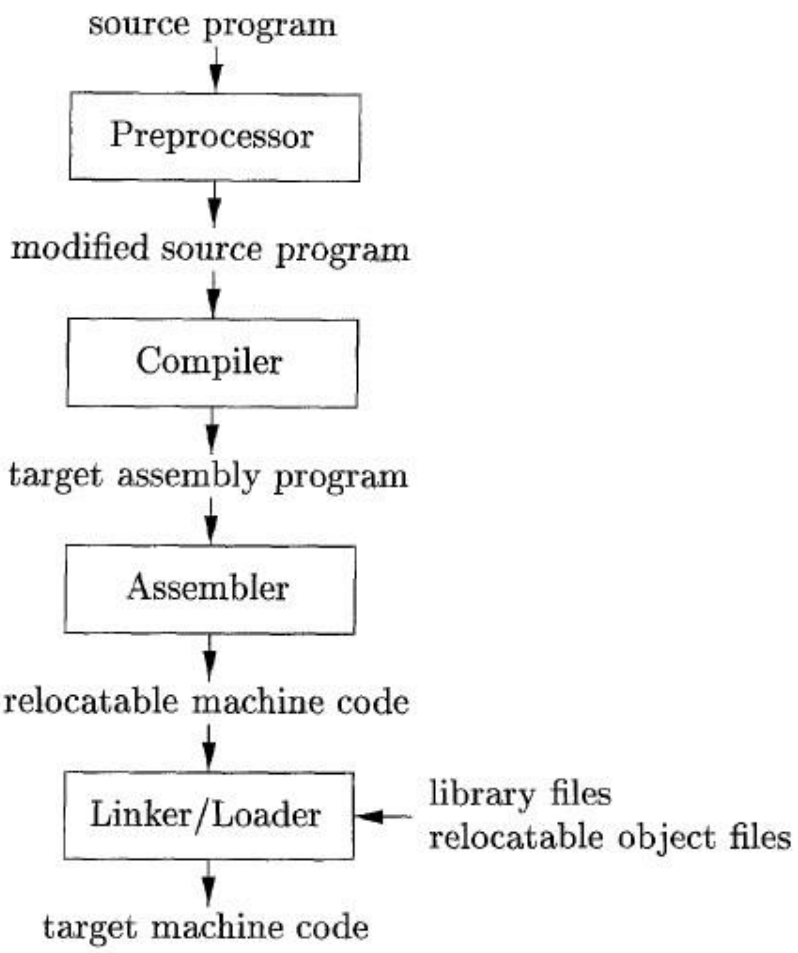
c. Design an assembler that converts assembly language to machine language

   The drawback of this is that the target language needs to be specified. Output of the various compilers to be known prior time

   So, our aim is to design a Compiler and Assembler for converting high-level language to machine language. In addition, certain other things are need for pre-processing and execution which is discussed in the next section.

1.2Language Processing System

A typical Language Processing system is given in Figure 1.1. The source program – program written in high-level programming language goes through a pre-processor. The pre-processor replaces macros and converts them into a complete code. For example, if we have a statement called #define MAX 100, in the source program, the pre-processor replaces MAX with 100 in all the places in the source program and passes it to the compiler. The compiler converts this to assembly language and the assembler converts to object language. At this point, the object language is called as the re-locatable object code. The code is re-locatable as it doesn’t have the exact address of the memory at which this code is to be loaded for execution. This re-locatable machine code is passed on to the linker. The linker will link multiple source files into one or link the current source files with the object code of the standard library and gets one object file. This file is then loaded into the main memory for execution by the loader.



Figur1.1 Language Processing System

To given an example, FORTRAN was the first real compiler to be built. It was built in the late 1950's and it required 18 person-years.

1.3 Compiler Basics

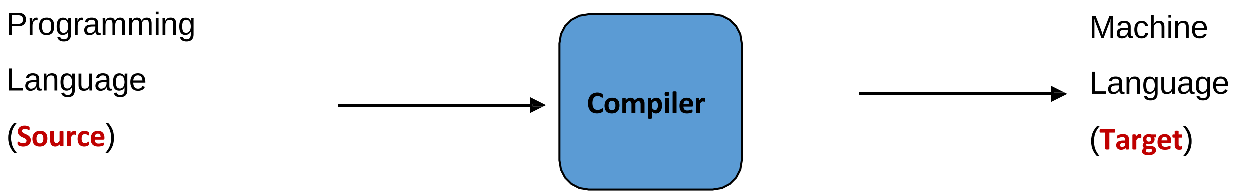


Figure 1.2 Compiler Overview

A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages and is shown in Figure 1.2. This doesn't require any concern about machine-dependent details for programmer.

1. **Types of Code**

In the process of generating assembly level code, the compiler could generate any one of the following types of codes:

a. Pure Machine Code: This refers to the set of Machine instruction which is independent of any operating system or library. These codes are typically available for the Operating Systems or Embedded Applications.

b. Augmented Machine Code: They refer to the machine instruction that has operating system routines along with run-time support routines.

c. Virtual Machine Code: These refer to the Virtual instructions that can be run on any architecture with a virtual machine interpreter or a just-in-time compiler. Ex: Java

2. **Work of a Compiler**

The Compiler has to necessarily do the following to translate high-level source code to low-level assembly code

- Processes source program
- Prompts errors in source program
- Recovers / Corrects the errors
- Produce assembly language program

After generating assembly language program, an assembler is used to convert the assembly language code into a relocatable machine code. The time of conversion from source program into object program is called compile time. The object program is executed at run time

3. Interpreter

An Interpreter is a language processor that executes the operation as specified in the source program. The inputs are supplied by the user. The interpreter processes an internal form of the source program and data at the same time (at run time) and therefore no object program is generated.

1. **Compiler vs Interpreter**

The following are some comparison between the compiler and the interpreter.

❖ For a compiler, a higher degree of machine independence exists and hence it facilitates high portability.

❖ A compiler supports dynamic execution. This helps in making modification or addition to user programs even during execution.

❖ A compiler also supports dynamic data type which helps in supporting the change in the type of object even during runtime

- An Interpreter on the other hand requires no synthesis part.
- Interpreter provides better diagnostics: more source text information available
- The machine-language target program produced by a compiler is much faster than an interpreter at mapping input to output.
- An interpreter is better with error diagnostics as it executes the source program statement by statement.

1.4 Compilation process

The process of Compilation and Interpretation is given in Figures 1.3 and 1.4 respectively.

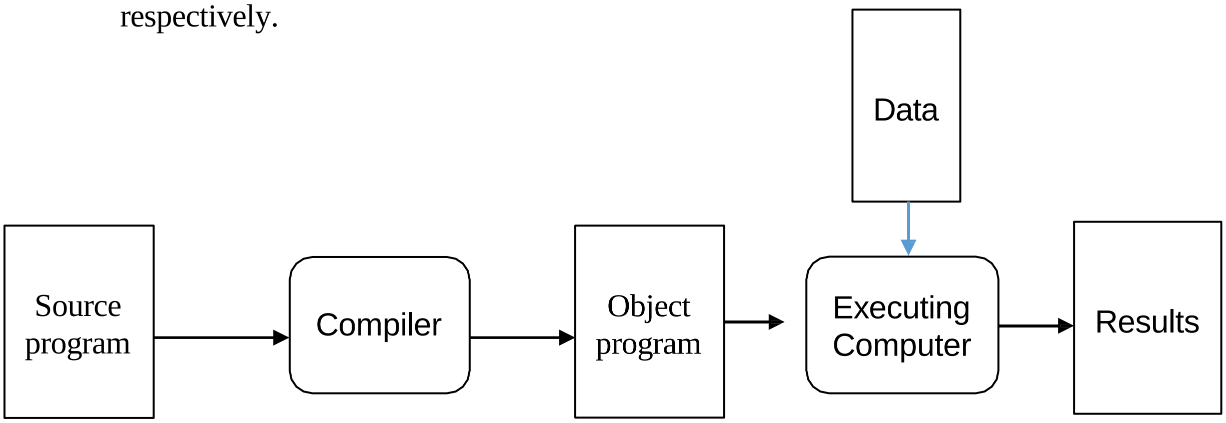


Figure 1.3: Compilation Process

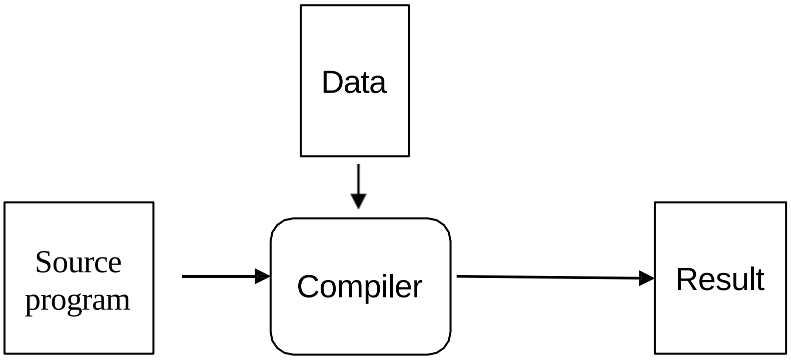


Figure 1.4 Interpretive process

As discussed, the compiler converts source program into relocatable object program, which then uses the data and executes in main memory on the other hand, the interpreter uses the source program and data and produces the execution without any intermediate object program.

1. Compiler

The compiler consists of two parts: 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.
- The synthesis of its corresponding program: 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***.

The Front End of the Compiler is typically language dependent. It depends on the source language but it does not depend on the target machine's architecture. The Back End is target dependent as it requires the instruction set of the target machine but it doesn't require information of the source language.

The Analysis and the Synthesis part of the Compiler is given in Figure 1.5. The Analysis part consists of three components while the Synthesis part consists of two components Code Generation and Optimization. In the process, it uses Error Table and a Symbol table for the generation of target code.

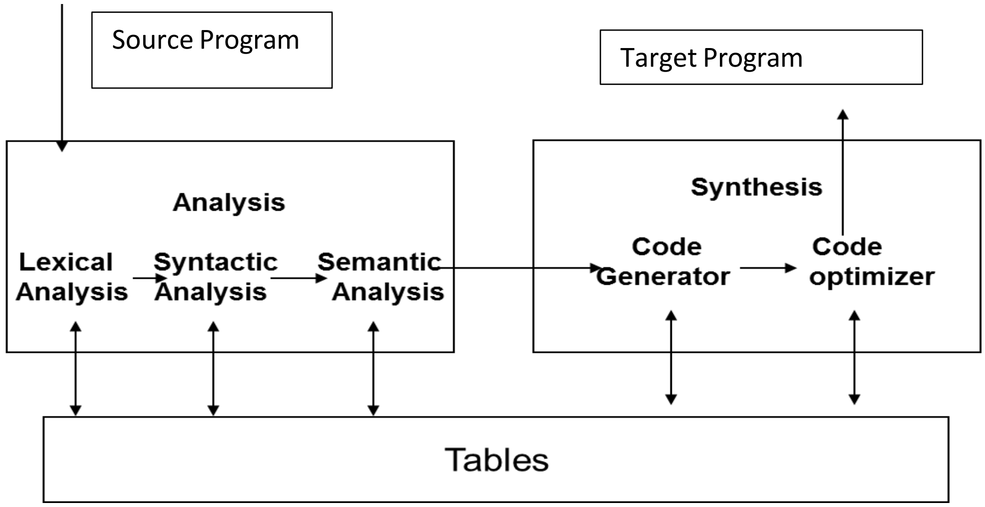


Figure 1.5 Analysis and Synthesis stages of the Compiler

1.4.2 Compiler Passes

The grouping of the work of the compiler into analysis and synthesis part poses the following questions.

- How many passes should the compiler go through?
- One for analysis and one for synthesis?
- One for each division of the analysis and synthesis?

To answer all these questions, the work done by a compiler is grouped into phases which is discussed in the next section.

1.4Phases of the Compiler

The compiler's analysis and synthesis part is grouped into 6 phases and is shown in Figure 1.6. The first three phases belong to the analysis phase and the last three phases to the synthesis phase. All the phases of the compiler interacts with the symbol table and the error handler.

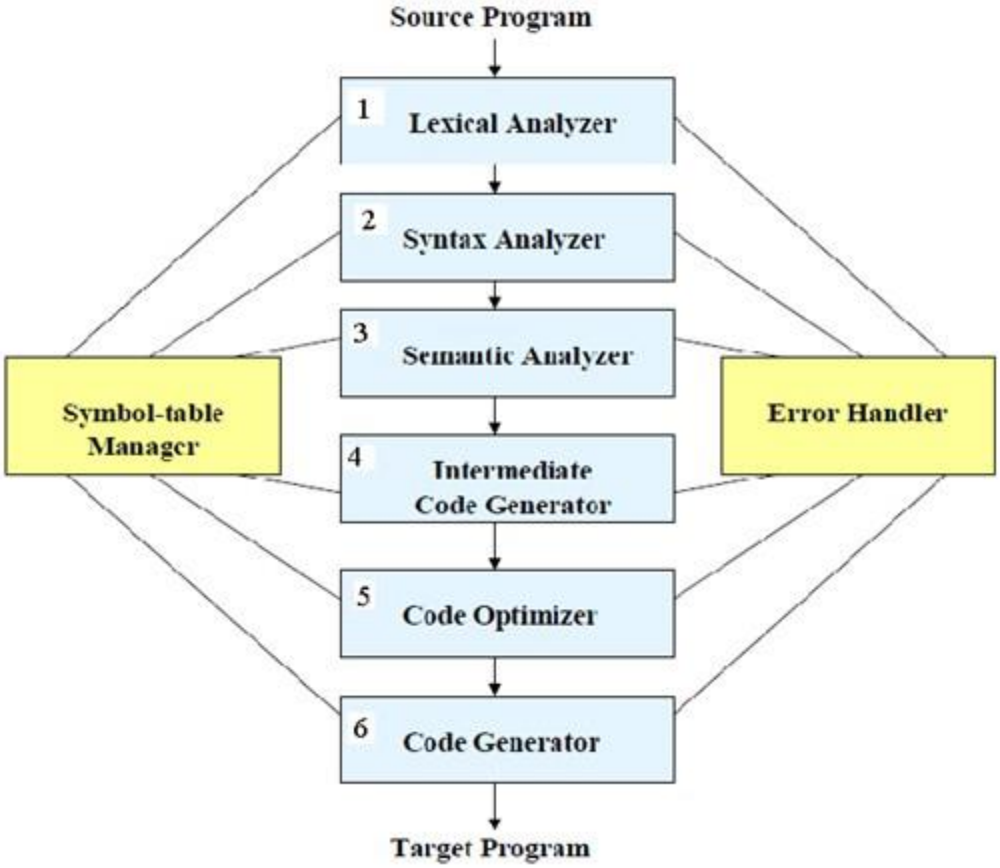


Figure 1.6 Phases of the Compiler

Lexical Phase: Lexical analyzer reads the stream of characters from the source program and combines the characters into meaningful sequences called lexeme. For every lexeme, the lexer (lexical analyser) produces a token of the form which is passed to the next phase of the compiler.

The token is of the form <token-name, attribute-value>, where token-name is an abstract symbol that is used during syntax analysis and an attribute-value: points to an entry in the symbol table for this token. During this phase, the symbol is created by the compiler, which has the information about the lexeme. The lexical analyser, typically skips all blanks, unwanted white spaces and comment lines that is being available in the source program.

Syntax Phase: The syntax phase of the compiler is the second phase. The phase is where the input from the source program is parsed and hence this phase is referred to as the Parser (parsing phase). The parser uses the tokens produced by the lexer to create a tree-like intermediate representation that verifies the grammatical structure of the sequence of tokens. 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

Semantic Phase: The semantic analyzer uses the output of the parser, which are the syntax tree and the information in the symbol table to check for semantic consistency in the source program. In this phase, the compiler gathers type information about the variables, operations, etc., and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. Type checking is done in this phase, where the compiler checks that each operator has matching operands. For example, typically an array index need to be an integer and the compiler must identify an error if a floating-point number is used to as an array index. Yet another job of the Semantic phase is type conversion, referred to as coercion. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Intermediate Code Generation: Compilers generate an explicit low-level or machine-like intermediate representation. This representation is necessary for generating assembly language. The characteristics of the intermediate representation are

- Ease of Generation
- Ease of translation to target assembly language.

The input to this phase is the syntax tree and output is intermediate code. A convention for Intermediate code generation is the three address code. The three address code has at the most three operands and 2 operators. For example,

$$x = y \textbf{ op } z$$

$$x = \textbf{op } y \qquad \rightarrow (1.2)$$

As expressed in statement 1.2, x, y, z are three operands which are typically addresses and ‘op’ refers to the operator in addition to the ‘=’ operator.

Code Optimization: This phase can operate either before or after code generation. The aim of his phase is to improve the intermediate code so that it results in better target code. This phase also aims at generating faster, shorter code, so that target code is generated that consumes less power. The important characteristic of this phase is to carry out simple optimizations that significantly improve the running time of the target program without slowing down compilation

Code Generation: This phase generation target assembly language. In this phase, the registers or memory locations are selected for each of the variables used by the program. The inputs to this phase which are the intermediate instructions are translated into sequences of machine instructions to complete an operation. One of the important consideration of code generation is the assignment of registers to hold variables as we have limited number of registers. This phase also need to decide on the choice of instructions involving registers, memory or a mix of the two.

Symbol Table: The symbol table is implemented as a data structure containing a record for each variable name, with fields for the attributes of the name. The symbol table is designed to help the compiler to identify and fetch the record for each name quickly. The symbol table has attributes that may provide information about the storage allocated for a name, its type, its scope. It also provides details on the function or procedure names, such things as the number and types of its arguments, the method of passing each argument and the return type.

Error Handler: The errors encountered in every phase are logged into the error handler for subsequent reporting to the user. The compiler however, recovers from the errors in every phase so that it can proceed with the compilation process. The compiler recovers from errors in either the panic mode of error recovery or phrase mode of error recovery.

Multi-pass Compiler: Several phases can be implemented as a single pass consist of reading an input file and writing an output file. A typical multi-pass compiler could do the following:

- First pass: preprocessing, macro expansion
- Second pass: syntax-directed translation, IR code generation
- Third pass: optimization
- Last pass: target machine code generation

1.5Summary

This module discussed need for a compiler and the various phases of the compiler.